TMO: Transparent Memory Offloading in Datacenters

Johannes Weiner
Meta Inc.
Menlo Park, USA
jweiner@fb.com

Niket Agarwal
Meta Inc.
Menlo Park, USA
niketa@fb.com

Dan Schatzberg
Meta Inc.
Menlo Park, USA
dschatzberg@fb.com

Leon Yang
Meta Inc.
Menlo Park, USA
lnyng@fb.com

Hao Wang
Meta Inc.
Menlo Park, USA
haowang3@fb.com

Blaise Sanouillet
Meta Inc.
Menlo Park, USA
blez@fb.com

Bikash Sharma
Meta Inc.
Menlo Park, USA
bsharma@fb.com

Tejun Heo
Meta Inc.
Menlo Park, USA
htejun@fb.com

Mayank Jain
Meta Inc.
Menlo Park, USA
mjain8@fb.com

Chunqiang Tang
Meta Inc.
Menlo Park, USA
tang@fb.com

Dimitrios Skarlatos
Carnegie Mellon University
Pittsburgh, USA
dskarlat@cs.cmu.edu

ABSTRACT

The unrelenting growth of the memory needs of emerging datacenter applications, along with ever increasing cost and volatility of DRAM prices, has led to DRAM being a major infrastructure expense. Alternative technologies, such as NVMe SSDs and upcoming NVM devices, offer higher capacity than DRAM at a fraction of the cost and power. One promising approach is to transparently offload colder memory to cheaper memory technologies via kernel or hypervisor techniques. The key challenge, however, is to develop a datacenter-scale solution that is robust in dealing with diverse workloads and large performance variance of different offload devices such as compressed memory, SSD, and NVM.

This paper presents TMO, Meta's transparent memory offloading solution for heterogeneous datacenter environments. TMO introduces a new Linux kernel mechanism that directly measures in realtime the lost work due to resource shortage across CPU, memory, and I/O. Guided by this information and without any prior application knowledge, TMO automatically adjusts how much memory to offload to heterogeneous devices (e.g., compressed memory or SSD) according to the device's performance characteristics and the application's sensitivity to memory-access slowdown. TMO holistically identifies offloading opportunities from not only the application containers but also the sidecar containers that provide infrastructure-level functions. To maximize memory savings, TMO targets both anonymous memory and file cache, and balances the swap-in rate of anonymous memory and the reload rate of file pages that were recently evicted from the file cache.

TMO has been running in production for more than a year, and has saved between 20-32% of the total memory across millions of servers in our large datacenter fleet. We have successfully upstreamed TMO into the Linux kernel.

CCS CONCEPTS

• Software and its engineering → Operating Systems; **Memory management**; **• Hardware → Non-volatile memory**.

KEYWORDS

Datacenters, Operating Systems, Memory Management, Non-volatile Memory

ACM Reference Format:
Johannes Weiner, Niket Agarwal, Dan Schatzberg, Leon Yang, Hao Wang, Blaise Sanouillet, Bikash Sharma, Tejun Heo, Mayank Jain, Chunqiang Tang, Dimitrios Skarlatos. 2022. TMO: Transparent Memory Offloading in Datacenters. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22), February 28 – March 4, 2022, Lausanne, Switzerland.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3503222.3507731

1 INTRODUCTION

The massive growth in memory needs of emerging applications such as machine learning, coupled with the slowdown of DRAM device scaling [19, 25] and large fluctuations of the DRAM cost, has made DRAM prohibitively expensive as the sole memory capacity solution.

In recent years, a plethora of non-DRAM cheaper memory technologies such as NVMe SSDs [8, 11] and NVM [14, 17, 24, 30, 34, 37] have been successfully deployed in datacenters, or are on their way. Moreover, emerging non-DDR memory bus technologies such as

CXL [9] provide memory-like access semantics and close-to-DDR performance. The confluence of these trends enable new opportunities for memory tiering not possible in the past [12, 16, 21, 31–33, 39, 40].

With memory tiering, less frequently accessed data is migrated to slower memory. The migration process can be driven by the application itself, a userspace library [6, 12, 26, 29], the kernel, or the hypervisor. This paper focuses on kernel-driven migration, or swapping, as it can be transparently applied to many applications without requiring any application modification.

Despite its conceptual simplicity, the only known large-scale adoption of kernel-driven swapping for latency-sensitive datacenter applications, is Google's deployment [18] of zswap [43], which we call *g-swap* in the rest of this paper. As a pioneer, g-swap significantly advanced the state of the art, but still has several major limitations.

First, g-swap supports only a single slow memory tier, i.e., a compressed memory pool managed by zswap. On one hand, this simplicity avoids the difficult problem of tackling heterogeneous memory tiers that exhibit large performance variances, e.g., NVMe SSDs and NVM devices. On the other hand, it is insufficient in maximizing memory-cost savings. §2.1 shows that NVMe SSDs offer an order of magnitude cost and power savings compared with compressed memory. Moreover, some application's data are hard to compress, e.g., machine learning models with quantized byte-encoded values.

Second, g-swap relies on extensive offline application profiling, and sets a static target page-promotion rate to determine how much memory to offload. This empirical approach is not sufficiently robust because the promotion-rate metric does not directly reflect an application's sensitivity to memory-access slowdown, and does not consider the performance characteristics of the offloading device. Specifically, our evaluation in §4.3 with one of the largest applications at Meta, directly contradicts g-swap's assumption that memory offloading should be kept below a pre-configured promotion rate in order to avoid hurting the application's performance. To the contrary, with a faster offloading device, a higher promotion rate actually improves the application's performance

To address these limitations, we built TMO, a transparent memory offloading solution for containerized environments. Fundamentally, TMO needs to answer two questions: *how much memory to offload* and *what memory to offload*. To answer the first question, TMO introduces a new kernel mechanism called Pressure Stall Information (*PSI*), which directly measures in realtime the lost work due to resource shortage across CPU, memory, and I/O. PSI is reported on a per-process and per-container basis. Unlike g-swap's promotion-rate metric, PSI accounts for both the performance characteristics of the slow memory tier and the application's sensitivity to memory-access slowdown. A userspace agent called *Senpai* uses the PSI metrics to dynamically decide *how much memory to offload* without prior application knowledge while taking into account hardware heterogeneity in datacenters.

To answer the question of *what memory to offload*, we had to address several challenges. First, the Linux kernel attempted to balance memory reclamation between file cache and swap-backed anonymous memory, but skewed heavily towards file cache through several heuristics. This relegated swap to only be used as an emergency overflow for memory, which is not suitable for offloading operations. In order to offload file and anonymous pages more evenly, we modified the kernel to balance the swap-in rate of anonymous memory and the reload rate of file pages that were recently evicted from the file cache.

Second, TMO holistically identifies offloading opportunities from not only the application containers but also the sidecar containers that provide connectivity functions (e.g., routing and proxy) among microservices or infrastructure-level functions such as logging and service discovery.

Finally, as memory is distributed across complex container hierarchies and containers may have different priorities, TMO accurately monitors each container's memory needs, and considers the hierarchies and properties of containers when making offloading decisions.

Currently, TMO provides transparent memory offloading across millions of servers in our datacenters, and saves 20-32% of the total memory. About 7-19% of the savings come from the application containers, while about 13% of the savings come from the sidecar containers.

The contributions of this paper are as follows:

(1) We introduce PSI, a Linux kernel component that directly measures in realtime the lost work due to resource shortage across CPU, memory, and I/O. This is the first solution that can directly measure an application's sensitivity to memory-access slowdown, without resorting to fragile low-level metrics such as the page-promotion rate.
(2) We introduce Senpai, a userspace agent that applies a mild memory pressure to effectively offload memory across diverse workloads and heterogeneous hardware with minimal impact on application performance. Compared with g-swap, the key advantages of Senpai are that it does not require offline application profiling and supports both SSDs and zswap as slow memory tiers.
(3) TMO performs memory offloading to swap at a subliminal memory-pressure level, and the turnover is proportional to file cache. This is in contrast to the historic behavior of swapping as an emergency overflow under severe memory pressure.
(4) We report our experience of deploying TMO in production to millions of servers.
(5) We have upstreamed PSI into the Linux kernel and also made Senpai[1] open source.

2 MEMORY OFFLOADING OPPORTUNITIES AND CHALLENGES IN DATACENTERS

In this section, we first present DRAM and SSD cost trends from Meta's datacenters that consist of millions of servers. Then, we showcase memory offloading opportunities across a wide range of applications running in our fleet. Furthermore, beyond workloads, we introduce datacenter and microservice memory tax and present fleet-wide characterization of offloading opportunities. Next, we

[1]https://github.com/facebookincubator/oomd

highlight the need for a memory offloading system to take into account the intricacies of the memory allocation subsystem. Finally, we show SSD heterogeneity across our datacenters and how it poses significant challenges to heterogeneous memory offloading.

2.1 Memory and SSD Cost Trends

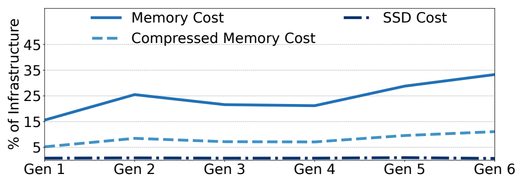

Figure 1: Cost of memory, compressed memory, and SSDs as a percentage of compute infrastructure across hardware generations.

Figure 1 shows the relative cost of DRAM, compressed memory, and SSD storage as a fraction of server cost in our datacenters. The x-axis shows different hardware generations. The compressed memory cost is estimated based on a 3x compression ratio representative of the average for our production workloads. The *Gen-1* hardware is near its end of life while *Gen-5* and *Gen-6* are expected to be deployed in the near future. The cost of DRAM, as a fraction of server cost, is expected to grow and reach 33%. While not shown in the figure, DRAM power consumption follows a similar trend and is expected to reach 38% of our server infrastructure.

Using compressed memory can reduce the cost significantly, but it is still insufficient and we need alternative memory technologies such as NVMe SSDs to further drive down the cost more aggressively. NVMe SSDs provide a much larger memory footprint per-server than DRAM, at substantially cheaper cost and lower power per-byte. We equip all our production servers with a very capable NVMe SSD. At the system level, NVMe SSDs contribute to under 3% of server cost (about 3x lower than compressed memory in our current generation of servers). Moreover, Figure 1 shows that, iso-capacity to DRAM, SSD remains under 1% of server cost across generations (about 10x lower than compressed memory in cost-per-byte!). These trends make NVMe SSDs much more cost effective compared to compressed memory for our fleet.

2.2 Cold Memory as Offloading Opportunity

Datacenter applications exhibit drastic differences in their memory behavior. To quantify the opportunity of memory offloading we characterize the memory coldness of seven large applications at Meta.

Figure 2 shows the amount of memory touched in the last x minutes, where x is 1, 2, or 5 minutes, as well as memory that remains untouched after 5 minutes. For example, for Feed, starting from the bottom, 50% of the memory is used in the last 1 minute, additional 8% in the last 2 minutes, and additional 12% in the last 5 minutes. The remaining 30% remains cold past the 5 minute interval. The memory coldness of applications vary drastically. For example, 81% of memory for Cache B is active in the last 5 minutes. By contrast, only 38% of memory for Web is actively used in the

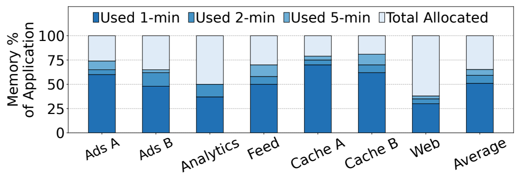

Figure 2: Application's recently used memory in 1 min, 2 min, and 5 min intervals, and memory not used in the last 5 min.

last 5 minutes. Overall, the memory offloading opportunity (i.e., fraction of cold memory) averages about 35%, but varies wildly across applications in a range of 19-62%, which emphasizes the importance of having an offloading method that is robust against application's diverse memory behaviors.

2.3 Memory Tax

To ease the operation of applications in datacenters, a significant amount of memory is used to enable microservices and provide infrastructure-level functions. We define as *datacenter memory tax* the memory required for software packages, profiling, logging, and other supporting functions related to the deployment of applications in datacenters. We further define as *microservice memory tax* all the memory required by applications due to their disaggregation into microservices, e.g., to support routing and proxy, and it is applicable uniquely to microservice architectures.

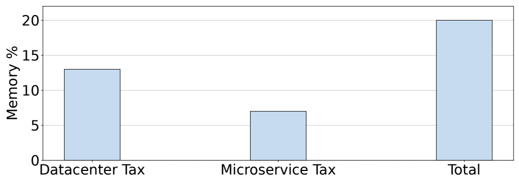

Figure 3: Datacenter and microservice memory tax.

Figure 3 shows the average memory tax as a percentage of the total server memory across all workloads at Meta. Both datacenter and microservice tax account for a significant percentage of memory usage. On average, the memory tax accounts for 20% of the total memory capacity. Datacenter memory tax is 13% and it is uniform across all workloads. Microservice memory tax accounts for 7% on average, and can vary depending on application characteristics. Notably, the performance SLA for most of the memory tax is more relaxed than that of memory directly consumed by applications. As a result, the memory tax was a prime target for memory offloading during our first production launch of TMO.

2.4 Anonymous and File-Backed Memory

Memory is separated into two main categories, anonymous memory and file-backed memory. Anonymous memory is allocated by applications and is not backed by a file or a device. File-backed memory represents allocated memory in relation to a file and is further stored in the kernel's page cache.

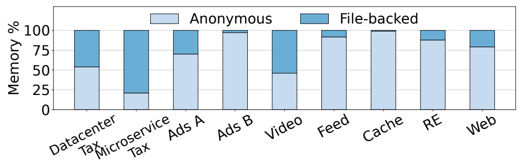

Figure 4: Anonymous and file-backed memory breakdown.

Figure 4 shows the breakdown of anonymous and file-backed memory for several large applications, datacenter memory tax, and microservice memory tax. The breakdown varies wildly across applications and memory taxes. Overall, we need to consider offloading opportunities for both anonymous and file-backed memory in order to maximize the savings.

2.5 Hardware Heterogeneity of Offload Backend

We define a memory *offload backend* as the slow-memory tier that holds offloaded memory. In our current production fleet this consists of NVMe SSDs and a compressed memory pool. In the future we expect this to include NVM and CXL devices.

NVMe SSD device heterogeneity is a significant challenge in datacenter environments. Multiple factors unavoidably create heterogeneous hardware in large-scale datacenters, including datacenter turn-ups, hardware refresh, and the need to maintain a diverse supply chain from different vendors.

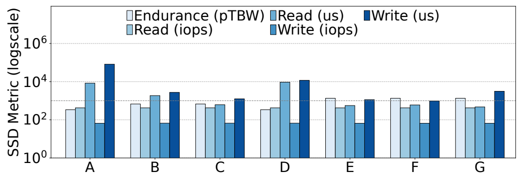

Figure 5: SSD characteristics in logscale. *A-G* **on the** *x***-axis represent different SSD devices.**

Figure 5 shows in logscale, the endurance, read and write IOPS and p99 latency across major SSD types across Meta's fleet. Newer devices are to the right of the figure. Notably, although SSD endurance has improved over SSD generations, it is still a limited resource and should be used judiciously by a memory offloading system. Furthermore, although IOPS are relatively stable across generations, read and write latency shows significant variation across generations, ranging from 9.3ms to 470us.

Besides SSDs, we also use compressed memory as an offload backend. The p90 latency of a 4KB read from compressed memory is about 40us. Compared with SSDs, compressed memory is an order of magnitude faster and with a small latency variance. Moreover, compressed memory avoids the endurance limits of SSDs. Overall, a memory offloading system needs to effectively tackle a heterogeneous fleet despite large differences in offload backends.

3 TMO DESIGN

The goal of TMO is to transparently offload memory to heterogeneous backends that offer cost-effective but slower memory accesses. TMO's workload-transparent design allows for a seamless deployment across a diverse set of applications and heterogeneous infrastructure. Fundamentally, TMO addresses the questions of *how much memory to offload* and *what memory to offload.*

3.1 Transparent Memory Offloading Architecture

Figure 6 shows the overview of the TMO architecture (left) and the memory and storage layout (right). TMO is composed of multiple pieces across userspace and the kernel. Unmodified workloads execute within a container ① and interact with the kernel memory management subsystem through system calls and paging. TMO's *Senpai* is a userspace component ② responsible for controlling the memory offloading process and deciding how much memory should be offloaded from each workload. The Senpai monitors application performance degradation in the form of pressure information that is reported by the Pressure Stall Information (PSI) component ③ that lives in the kernel. We describe PSI in §3.2 and Senpai in §3.3.

Based on the PSI information, Senpai drives the offload process by writing to cgroup control files ④ that trigger the kernel's memory reclamation logic. The reclamation logic determines what memory to offload. We describe our modifications to Linux kernel reclaim algorithms in §3.4. The memory management subsystem ⑤ exposes memory pressure information to the PSI module and triggers read and write operations to the offload backend ⑥ and the regular filesystem. TMO supports offloading to either a compressed memory pool based on zswap or storage devices through swap.

The right side of Figure 6 shows a high-level layout of the memory and storage in addition to their respective operations. TMO targets workload memory as well as datacenter and microservice memory tax. Furthermore, TMO is responsible for both offloading memory to the backends ⑦ by compression, swapping or discarding page cache, as well as bringing it back to main memory when needed, following the reverse operations. Finally, the figure shows TMO's currently supported offloading backends ⑧, with zswap being in DRAM and swap and filesystem in storage SSDs.

3.2 Defining Resource Pressure

In determining how much memory to offload we need a measure of the impact on application performance due to lack of memory. Devising a uniform metric for application performance across diverse workloads is by itself difficult, but even more challenging is capturing the performance impact of lack of memory specifically, excluding other unrelated issues.

The OS kernel exports various event counters that can indicate problems, such as counts of major page faults, and most prior work has used major faults as a primary indicator for approximating the impact of memory offloading to application performance. However, elevated major fault counts could be due to a workload starting up or a working set transition, and not due to a shortage of memory. Also, in a heterogeneous memory system with diversity across offloading backends, a given major fault rate could constitute a problem on a slow storage device while being insignificant on a faster one. In

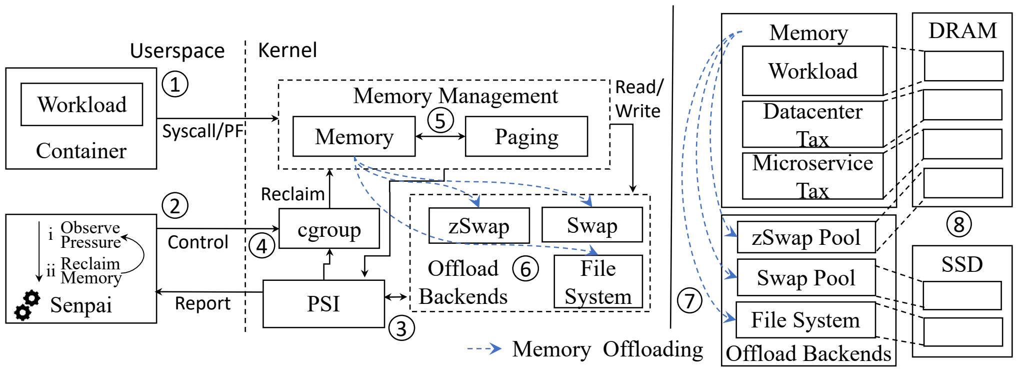

Figure 6: Overview of the TMO architecture (left) and the memory and storage layout (right).

general, it can be difficult to understand if particular kernel events are a functional problem for a workload or not. We will demonstrate in the evaluation §4.3 that major fault rate or promotion rate has limitations, especially when considering heterogeneous offloading backends with significant diversity in performance.

3.2.1 PSI Metrics. Fundamentally, PSI exposes a metric that represents the amount of lost work due to the lack of a resource. It can be measured for a single process, a container or machine-wide. PSI calculates pressure metrics by considering only non-idle processes. For each non-idle process, PSI further distinguishes between periods of time when a process is exclusively either *runnable* or *stalled* due to insufficient resources. Furthermore, it defines the compute potential as the number of non-idle processes capped at the number of CPUs. PSI is the proportion of compute potential that is unproductive due to resource stalls. It is often represented as a percentage. For containers and whole-system domains, PSI introduces two pressure indicators for each resource called `some` and `full`. The `some` metric tracks the percentage of time in which at least one process within the domain is stalled waiting for the resource. The `full` metric tracks the percentage of time in which all processes are delayed simultaneously.

Consider the example in Figure 7, which shows the execution time of two processes A and B as well as their stall time with a dotted box. The `some` stalling information is shown with a blue arrow, while `full` stalling is shown with a green arrow. The execution time is normalized to 100% and partitioned into four sections. During the first quarter, only one process stalls at a time, either process A or process B. Hence 12.5% of time is accounted for by the `some` metric. Instead, in the second quarter, for 6.25% of time, both processes stall concurrently during their execution time and hence this stalling time is accounted for by the `full` metric. In addition, 18.75% of stall time is accounted for by the `some` metric. The next two quarters show different variations of stalls and how `some` and `full` are accounted for accordingly. Overall, `some` aims to capture added latencies to individual processes due to lack of a resource while `full` indicates the amount of completely unproductive time in the container or system.

3.2.2 PSI Comparison to Other Metrics and Cost. One existing mechanism is the resident set size (RSS), which tracks the amount of main memory that belongs to a process. The main limitation of RSS is that by itself it does not capture the impact of memory, or a lack thereof, on application performance. Other metrics, like promotion rate account for the number of swap-ins per second. A drawback of the promotion rate is that it does not take into account the performance characteristics of the offloading backend. Furthermore, it fails to capture application performance improvements as more memory becomes available due to offloading.

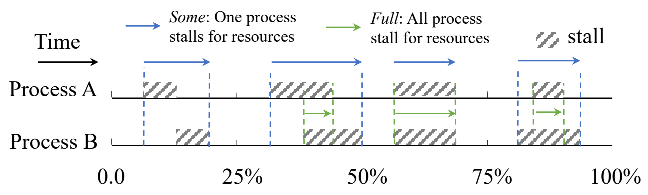

Figure 7: Kernel resource pressure metrics.

Instead, PSI directly captures the impact of memory-access slowdown to an application and further incorporates the performance and utilization characteristics of the offloading backend. The main cost of PSI is scheduling latency since some logic needs to be performed on a context switch. In real applications in our fleet the overhead is negligible. Beyond datacenters, PSI is enabled by default on all major Linux distributions including Android. We further compare PSI to other metrics in Section 4.

3.2.3 PSI Across System Resources. To track memory pressure, PSI records time spent on events that occur exclusively when there is a shortage of memory. Currently, this includes three occasions. The first occasion is when a process triggers reclaiming pages when memory is full and the process tries to allocate new pages. The second occasion is when a process needs to wait for IO for a refault, i.e., a major fault against a page which was recently evicted from the file cache. The third occasion is when a process blocks on reading a page in from the swap device.

Block IO stalls are more difficult to accurately calculate because existing hardware provides little insights into device contention. In particular, we cannot attribute a portion of a stall on block IO to the device being oversubscribed or simply expected latency of a device access. We therefore treat any process waiting on block IO completion to be stalled due to lack of IO. This has worked well for us in production across diverse workloads.

CPU stalls are accounted for as the periods of time when a process is runnable but needs to wait for an idle CPU to become available. CPU `full` pressure is only possible within a container, which occurs when none of the processes can execute either due to outside competition or due to configured limits on the cgroup's CPU cycles.

3.2.4 PSI Use Cases. By aggregating these process state times across all CPUs in the system and breaking them down per container, PSI gives an intuitive insight into the quality of resource provisioning for any given workload and container, and provides the ability to root-cause performance problems or SLO violations observed at the application level such as missed response deadlines.

The pressure metrics are aggregated in realtime, and are available at the microsecond resolution as well as running averages (10s, 1m, 5m). This enables resource management on both ends of the pressure spectrum.

Large amounts of `full` pressure can be used to detect unacceptable losses of productivity that require immediate remediation. These can be caused by overlapping peaks in a system's main workload and a system maintenance process, as well as application bugs, or errors in configuration and scheduling. Any of these situations require a timely intervention in order to prevent disruption to service quality or threats to the health of the server itself. For example, long before the kernel's out-of-memory killer triggers, applications can be functionally out of memory when the lack of it causes delays that prevent the application from meeting its SLO. Userspace out-of-memory killers can monitor `full` metrics and apply killing policies.

Additionally, `some` pressure measures the latency impact of lack of a resource and is sensitive enough to detect aggregate delays below the threshold that would cause applications to suffer meaningful performance losses. In TMO we exclusively rely on `some` metrics to measure the performance impact of memory offloading. By ensuring the value is low but non-zero, we maintain the presence of resource contention just high enough that no resources are going idle, but not so high as to disturb the nominal operation of the workload. In this pressure range, the workload is provisioned with the minimum amount of the resource it requires to function well.

Before PSI, operators relied on correlating indirect metrics such as kernel time, variations in application throughput, event counters for reclaim activity, file re-reads, and swap-ins, in order to estimate the resource health of workloads. This required an intuitive understanding of the storage hardware device characteristics and kernel behavior. For example, the file read-ahead algorithm may shield the application to varying degrees from the effects of recorded cache re-reads. PSI on the other hand measures productivity losses from qualifying stall events directly at the process level. It takes into account differences in underlying hardware, the effectiveness of the kernel's memory management algorithms, and even the internal concurrency (i.e., `some` vs. `full` stalls) of the workload.

3.3 Determining Memory Requirements of Containers

Senpai is a userspace tool responsible for driving memory offload, using PSI metrics to determine how much memory can be moved out.

Estimating the memory a workload requires is challenging. Along with their workingsets, applications also make allocations or instantiate file cache that is rarely used - or even used just once - but the kernel only identifies and reclaims such cold pages when free memory is low. This leads to applications' memory footprints being larger than the number of pages they would need resident in order to function normally. In practice, most developers notice when their applications run out of memory, but are rarely made aware when memory is (vastly) overprovisioned.

Senpai continuously engages the kernel's reclaim algorithm, using PSI as feedback on workload health to modulate its aggressiveness, to identify the share of allocated pages that the workload requires to function, and offload any beyond that. This proactive approach ensures that available offloading capacity is utilized optimally on one hand, and simultaneously provides an accurate workingset profile of the application over time. This allows application developers to more precisely provision memory capacity for their workloads.

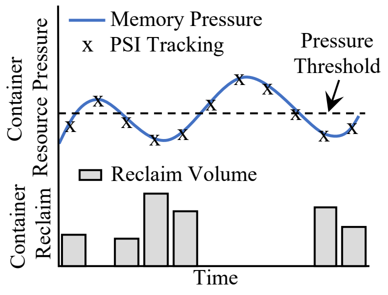

Figure 8: Example of Senpai PSI tracking and reclaim volume tuning.

Figure 8 shows a high-level overview of Senpai's operations. Once every few seconds, Senpai calculates for each cgroup the amount of memory to reclaim as follows:

$$reclaim\_mem = current\_mem \times reclaim\_ratio \times max(0, 1 - \frac{PSI_{some}}{PSI_{threshold}})$$

PSI_{some} is the cgroup's `some` PSI metric. $PSI_{threshold}$ and *reclaim_ratio* are configurable parameters. *current_mem* is the current memory footprint of the cgroup. No memory is reclaimed when PSI_{some} is above $PSI_{threshold}$. Otherwise, Senpai asks the kernel to reclaim *reclaim_mem* from the cgroup. As PSI_{some} approaches $PSI_{threshold}$, Senpai gradually reclaims less memory in order to achieve a mild steady-state memory pressure.

Senpai uses the cgroup interface to direct the kernel's reclaim algorithm. In an early version, Senpai would continuously adjust the memory limit of the workload's cgroup. Lowering the limit below the current size causes the kernel to reclaim the difference. Instead, increasing the limit relieves pressure and allows the workload to expand. However, the statefulness of the memory limit combined with dynamic workloads can cause problems in some situations. For example, if a container is under rapid memory growth and its size is actively expanding, it may become blocked until Senpai can raise its limit further.

To address this problem, we added a stateless `memory.reclaim` cgroup control file to the kernel. This knob allows Senpai to ask the kernel to reclaim exactly the calculated memory amount without applying any limit, thus avoiding the risk of blocking expanding workloads.

As PSI is effective in capturing the impact of memory offloading on diverse applications, we strive for using a single globally optimal Senpai configuration to support all applications. We studied the performance sensitivity of applications related to file cache and anonymous memory, and iteratively arrived at the current configuration that is used in production for all applications, specifically, *reclaim_ratio*=0.0005 and *PSI_threshold*=0.1%. In production, reclaim is performed every six seconds. We set this value empirically to leave enough time to measure the delayed impact (refaults) of reclaimed memory. The step size of how much memory is reclaimed (*reclaim_mem*) depends on how far or how close the observed pressure (PSI_{some}) is to the target threshold ($PSI_{threshold}$). The maximum is 1% of the total workload size in each reclaim period. As a result, reaction time to extreme contraction tends to be minutes. Adaptation to workload expansion, on the other hand, is immediate.

On one hand, TMO's effectiveness is not very sensitive to these parameters. On the other hand, our experiments demonstrate that certain workloads (e.g., batch workloads with less stringent SLOs) can tolerate more memory pressure, which provides opportunities for offloading more memory. We leave it as future work to perform automated or online tuning of these parameters to maximize savings.

Senpai has additional mechanisms to modulate reclaim in certain events such as SSD write endurance thresholds being exceeded or swap space exhaustion. Besides the memory PSI metrics, Senpai also monitors the IO PSI metrics, because the memory PSI metrics alone are insufficient to fully capture the performance impact of memory offloading. In some cases, refaults induced by Senpai might not impact the workload in the form of fault latencies, but might slow down the storage device enough to impact the workload's operation indirectly. In theory, the same idea might apply to CPU contention, but in practice, we have not found the CPU cycles spent on refaults to be significant.

Advantages of a userspace Senpai.

Senpai is implemented in userspace on top of exported kernel interfaces for PSI and cgroup memory control. This has several advantages over a hypthetical in-kernel implementation. First, userspace has full access to floating point units and can therefore perform calculations more effectively. Second, release cycles for userspace components tend to be significantly faster than for the kernel. We have repeatedly iterated through Senpai variables over experimental and deployment phases, and have also deployed different settings across applications for extended periods of time. In the future we have plans to exploit distinct Senpai configurations across workloads with different performance SLO thresholds.

3.4 Kernel Optimizations for Memory Offloading

Senpai relies on the kernel to reclaim cold memory pages from cgroups. Instead of using expensive full page table scans to determine which memory pages are cold, Senpai lets the kernel's reclaim algorithm choose the pages to offload. This algorithm operates by maintaining a pair of active/inactive page LRU lists for both file and swap-backed pages and by reclaiming colder pages first. This mechanism is production-tested for identifying less-used memory pages with relatively low CPU cost and high accuracy. Using it in Senpai greatly simplifies our implementation. In production, reclaim driven by Senpai consumes 0.05% of all CPU cycles, a negligible amount.

Historically, the kernel has been very conservative in using swap. In the era of slow, spinning disks, the semi-random IO patterns produced by paging would impose substantial latencies on the workload due to seek overheads of rotational media. Workloads typically could not sustain the IO overhead of constant paging, so supporting working sets larger than available memory was not considered a case to optimize for. Rather, the kernel focused on simply reclaiming memory that would never be accessed again, specifically, file cache from previous reads. The kernel's logic for balancing scans of file cache against swap-backed memory would attempt to balance reclaim success rate between the two pools of memory, but skewed heavily towards file cache through a number of different heuristics. This relegated swap to only be used as an emergency overflow for memory. While developing TMO, we noticed that substantial portions of a workload's file cache would be reclaimed before the kernel began to consider swapping out cold swap-backed memory.

Fundamentally, at the time the algorithm was conceived, the kernel lacked knowledge of when reclamation of file cache was resulting in recently resident file pages (part of the working set) needing to be read in from storage. We augmented the kernel with non-resident cache tracking. Whenever a file cache page is evicted from memory, a counter that is maintained per cgroup is incremented and the current value is stored in a shadow entry that replaces the page. When a file page is faulted in, the kernel can determine the reuse distance [7], i.e., the difference between the current page fault count and its value in the shadow entry. If the reuse distance is smaller than the size of resident memory, the fault is considered a refault.

TMO uses this refault-detection mechanism in two ways. For one, we repurpose it to calculate memory PSI, by tracking stalls due to recently evicted file cache, and excluding stalls due to first-time-accessed file cache.

For the purpose of offloading to swap backends, we modify the kernel reclaim algorithm to exclusively reclaim from file cache so long as no refaults occur. As soon as refaults begin to occur, the kernel now balances reclamation of file cache and swap based on the refault rate and swap-in rate respectively. With this new reclaim algorithm, swap occurs as soon as the file cache's working-set begins to be reclaimed, i.e., refaults begin to occur. This approach more equally offloads file-backed and swap-backed cold memory, and minimizes the aggregate amount of paging. Our changes to the Linux kernel reclaim algorithm have been upstreamed.

3.4.1 Transparent Zswap Support. As described in §2.5, using compressed memory as an offload backend has substantial performance advantages. It provides a memory offload solution in the absence of SSDs, and avoids the endurance limits imposed by SSDs.

As a new Linux component, *zswap* allows the kernel to store anonymous memory compressed in RAM. Instead of writing pages to a swap partition on disk, the kernel compresses the page and stores them in RAM. Accessing the original anonymous memory page will still result in a page fault, but this time no disk IO needs to be performed, the page can be decompressed and made resident. Zswap fault latencies can be substantially faster than loading from a block device, but the per-page memory savings depend on the compressibility of the application's cold anonymous memory.

With Senpai and PSI, we can deploy swap or zswap with no additional modifications to the application or Senpai, since Senpai automatically adjusts the reclaim rate based on the offload backend's performance. In the case of faster storage or zswap, more pages can be offloaded due to lower latency per-page swap-in.

4 EVALUATION

TMO has been running in production for more than a year leading to significant memory savings across Meta's fleet. Our evaluation focuses on showcasing different aspects of TMO. Specifically, it answers the following questions:

(1) How much memory can TMO save?
(2) How does TMO impact memory-bound applications?
(3) Are PSI metrics more effective than the promotion-rate metric?
(4) How to tune TMO's configurable parameters?
(5) Can TMO avoid SSD wear-out due to offloading writes?

4.1 Fleet-Wide Memory Savings

We break TMO's memory savings into savings from applications, datacenter memory tax, and application memory tax, respectively.

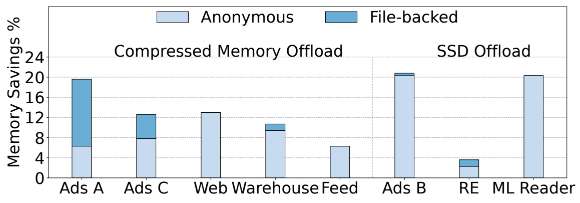

Figure 9: Memory savings across eight applications normalized to their resident memory size.

Application savings. Figure 9 shows the relative memory savings achieved by TMO for eight representative applications using different offload backends, either compressed memory or SSDs. Using a compressed-memory backend, TMO saves 7-12% of resident memory across five applications.

Multiple applications' data have poor compressibility and their memory offloading is more effective with a SSD backend. For those applications, Figure 9 shows that TMO achieves significant savings of 10-19% with a SSD backend. Such savings that do not rely on compression would be unattainable with previous approaches [18]. Specifically, machine learning models used for Ads prediction commonly use quantized byte-encoded values that exhibit a compression ratio of 1.3-1.4x, leading to poor memory savings through a compressed memory offloading. Instead, for such applications SSD offloading provides a more cost-effective solution. Overall, across compressed-memory and SSD backends, TMO achieves significant savings of 7-19% of the total memory, without any noticeable application performance degradation.

Datacenter and application memory tax savings. Beyond regular workload memory, TMO further targets datacenter and application memory tax.

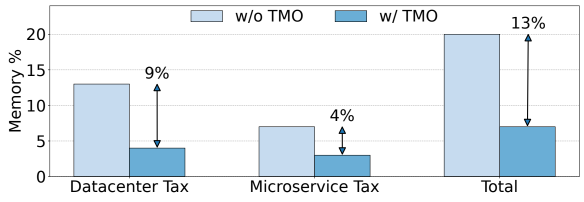

Figure 10: Datacenter and application memory tax savings normalized to a server's total memory.

Figure 10 shows the relative memory savings by offloading memory tax across Meta's fleet. When it comes to datacenter tax, TMO saves on average 9% of the total memory within a server. Application tax savings account for 4%. Overall, TMO achieves on average 13% of memory tax savings, in addition to actual workload savings. This is a significant amount of memory at the scale of Meta's fleet.

4.2 Performance Impact on Memory-Bound Applications

In this section we demonstrate how TMO helps improve the performance of memory-bound applications. Specifically, we perform an analysis of the Web application, which is one of the largest applications at Meta. Our production load-testing framework provides high fidelity A/B tests and we use it to guide our hardware and software optimizations across Meta's fleet. The hardware used for each of the experimental machine pools in the experiments below are production Skylake 64GB hosts (representative of what is deployed in our fleet). We chose enough machines (often 10s of hosts) to have statistically accurate A/B comparisons. In this section we will extensively demonstrate results from Web due to its excellent testing framework, but most of the conclusions are representative across our fleet for other large applications. The performance metric is requests per second (RPS) with a predefined target tail latency. Each server automatically throttles its RPS in order to meet the tail latency.

The memory profile of Web works as follows. It starts by loading up the entire file system cache into memory and then lazily loads anonymous memory as Web requests arrive over time. As Web hosts get closer to their memory limits, servers self-regulate and throttle the processed requests per second in order to avoid running out of memory. The baseline tier in Figure 11(a) illustrates that behavior,

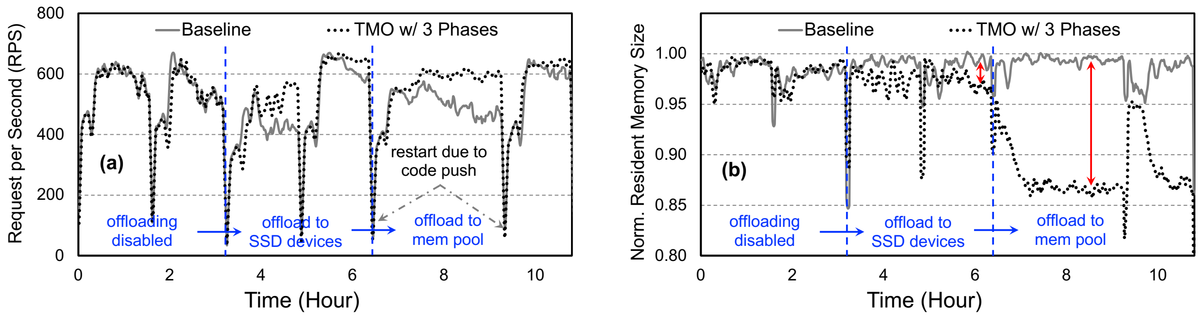

Figure 11: Performance improvement and memory saving for the Web application on memory-bound hosts. We set up two tiers: started with identical configuration without any swap enabled and later enabled SSD & then compressed memory offloading on the 2nd tier.

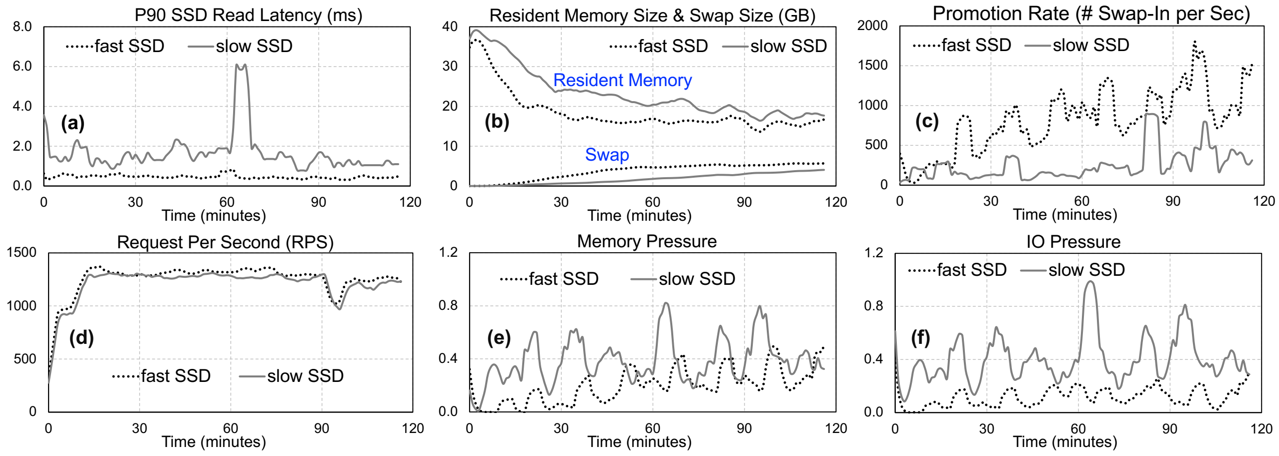

Figure 12: The Web application under the control of TMO with fast and slow SSDs. The solid-gray configuration has lower RPS/performance but actually lower promotion/swap-in rate; while its PSI metrics are indeed higher.

i.e., as the server gets memory bound, the RPS gradually goes down. Note that the loss can be more than 20% over a two-hour period.

Figure 11(b) shows the resident memory size when offloading to SSD devices (2nd phase in the middle) and a compressed memory pool (3rd phase on the right), respectively, compared with a baseline without offloading (1st phase on the left). The figure shows that once TMO is enabled, it is able to offload a significant fraction of system memory to the heterogeneous backends and the RPS drop is eliminated over time. This leads to 20% of capacity savings for Web. Because the Web application's data has a high compression ratio of 4x, compressed-memory offloading is effective and saves about 13% of Web memory at the peak. By contrast, SSD offloading saves only about 4% memory in the best case, because Web is sensitive to memory-access slowdown.

4.3 Comparing PSI and Promotion Rate

TMO relies on PSI to report the lost work due to lack of resources such as CPU, memory, and IO. Compared with counting page-promotion events [18], PSI naturally factors in an application's sensitivity to memory-access slowdown as well as the performance and utilization aspects of a given offload backend. These cannot be achieved by low-level metrics such as the promotion rate.

In general, a slower backend directly impacts the PSI memory pressure and IO pressure, since a page fault takes longer to resolve, which leads to lower memory savings. Figure 12 presents various metrics to compare TMO with two different SSD backends, when running the Web application under load tests, with a Senpai configuration to keep the memory and IO pressure below a predefined threshold. The "fast SSD" and "slow SSD" in Figure 12 refer to SSD C and B in Figure 5, respectively.

Figure 12(a) shows that the P90 read latency of the slow SSD is significantly worse than that of the fast SSD. Figure 12(b) shows that TMO with the fast SSD backend exhibits more aggressive swapping, with a higher swap size and a lower resident memory size. Note the consistent 1.6-3GB difference between the amount of anonymous memory offloaded across the two SSDs. This corresponds to a significant difference of 10-15% of the resident set size.

Figure 12(c) further shows that the promotion rate (i.e., swap-in per second) is higher for the fast SSD. Combined with Figure 12(d), we see that the host with a higher promotion rate actually processes more requests per second. This illustrates the flaw in simply using promotion rate as a proxy for application performance. First, it

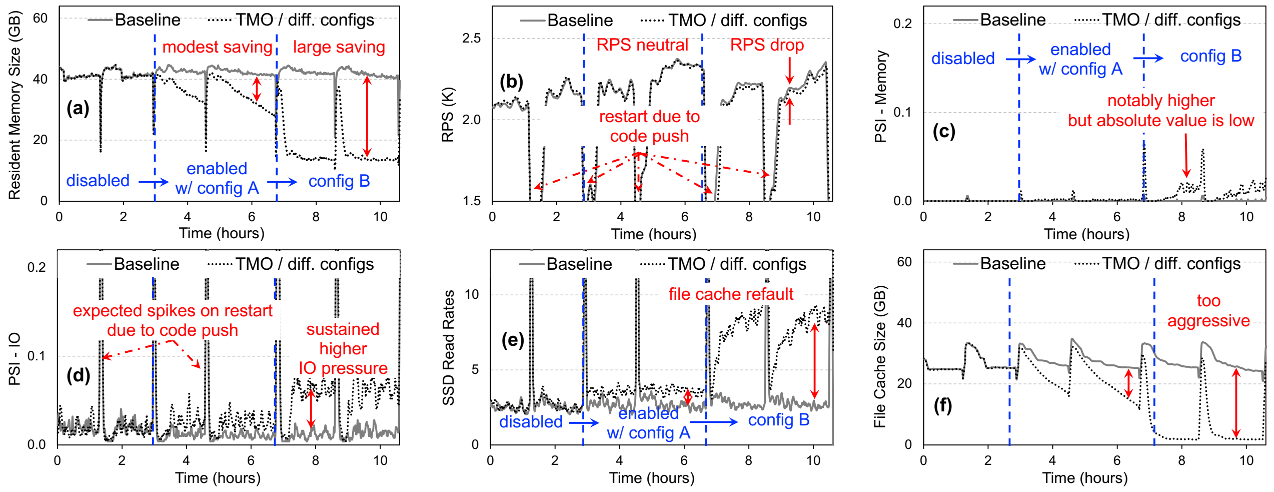

Figure 13: Memory savings, RPS, observed memory and IO PSI metrics, SSD Read Rate, and File Cache Size for the Web application on non-memory-bound hosts. We set up two tiers: the 2nd tier started with TMO disabled and later enabled compressed memory offloading with 2 different configs.

ignores the fact that the performance characteristics of the offload backend impacts the offloading decision. Second, it ignores the fact that some applications actually gain better performance as more memory becomes available due to more aggressive offloading. The observed behavior in this experiment demonstrates that promotion rate is not a robust metric to drive offloading decisions with not only heterogeneous offload backends, but also the same offload backend with device-to-device performance variance in the fleet, e.g., slow SSD vs. fast SSD as shown above.

Unlike the promotion rate, the PSI memory and IO pressures in Figures 12(e) and (f) adapt well to maintain the application-level performance. It offloads more memory for a better-performing offload backend, while keeping the memory pressure within the target threshold. Overall, the figure shows that Senpai naturally adapts its behavior based on the real-time characteristics of individual backends and applications.

4.4 Impact of Senpai Configuration Tuning and Production Selection Process

In order to select the production Senpai configurations for memory and IO PSI metrics, and the maximum reclaim step, we performed tuning of the parameters across many production workloads and finally selected one global configuration. Our goal was to select a configuration that 1) avoids application end-to-end performance SLA regressions when compared to a system without offloading, and 2) provides maximum memory savings.

Figure 13 compares the Web application across 2 different Senpai configurations, *Config A* and *Config B*. *Config A* has more mild settings compared to *Config B*. Figures 13(a) and (b) show the memory savings and RPS across the two configs. While the aggressive *Config B* leads to larger memory savings compared to *Config A*, which still saves significant memory for Web, it comes at the cost of significant RPS regression, violating our design goal.

We are currently running our production fleet with *Config A* and it works well across different offload backends. In order to arrive at this config, we had to understand the sensitivity of applications to various Senpai configuration parameters. Figures 13(c) and (d) show that across the two configs, memory PSI pressure was similar to the baseline tier without TMO, while IO PSI pressure with *Config B* was noticeably higher to *Config A*, which itself tracked baseline pretty well. Recall that memory PSI pressure tracks stalls due to shortage of memory, while IO PSI pressure tracks Block IO stalls coming from SSD reads. In this experiment we are using compressed memory pool as the offload backend, and so Block IO stalls should be coming from increased file cache faults. Figures 13(e) and (f) confirm that we are experiencing higher IO PSI pressure in file cache since we see that SSD read rates are higher, while resident file cache is significantly lower, with *Config B*.

A higher IO PSI pressure doesn't always direclty translate into application performance loss unless the data is critical. Digging deeper and tracing the SSD reads we found that SSD reads are contributed primarily by application bytecode which is directly loaded from filesystem and cached in the file cache. In our fleet, Web is bound in the cpu front-end (fetching instructions/code) and increased IO PSI pressure caused by higher application bytecode misses in the file cache directly impacted the RPS with *Config B*. This tuning experiment, yet again, highlights the importance of holistically tuning for both file cache and anonymous memory in TMO.

4.5 Handling SSD Endurance

SSDs have limited write endurance. To avoid premature SSD wearout, TMO regulates writes during memory offloading to SSDs. Figure 14 shows memory offloading for the Ads B application with and without the regulation mechanism. Based on a fleet-wide analysis, we have identified that 1MB per-second is a safe threshold for the SSDs in our fleet. The figure shows that Senpai accurately

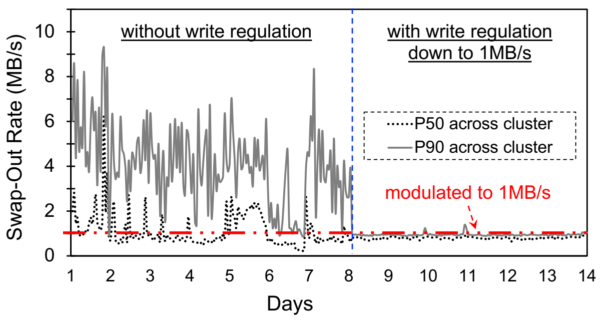

Figure 14: Swap rate with and without write regulation.

modulates the write rate of memory offloading, which gives us the confidence to deploy TMO across Meta's fleet.

5 DISCUSSION

5.1 Production Deployment Experience

Since PSI is already in the upstream Linux kernel and Senpai is also open source, it is feasible for other organizations to deploy the full TMO solution. We describe our deployment experience as a reference for others.

We first deployed TMO to tackle datacenter memory and microservice memory tax, as they had more relaxed performance constraints compared with user-facing workloads such as our Web application. Previous measurements of tax memories were *ad hoc*, reliant on measuring how much memory was used before hitting out-of-memory or unreliable per-process metrics such as RSS. With TMO, we were able to measure and attribute memory consumption to specific processes and proactively act on regressions. This improved observability by itself was very valuable and helped us accurately repurpose tax memories for application workloads.

After the success with offloading datacenter and microservice memory tax, we then deployed TMO in file-only mode (i.e., without swapping) for all applications. The improved observability helped accurately setting the memory size for application containers. It also helped determine how much file cache each application needed. In one case, TMO helped detect that an application unexpectedly consumed a large amount of file cache due to its repeated execution of a self-extracting binary. Most methods for memory-usage accounting would not detect this as they discount file cache as negligible or reclaimable. We changed the application to extract the binary ahead of time, which resulted in 70% memory savings for the application!

Finally, we switched TMO from file-only mode to include swapping for several of the largest applications at Meta. The performance SLO of our user-facing applications is much more stringent than that of the system-tax components. As PSI works well to capture the impact of memory offloading on diverse applications, we were able to use a single globally-optimal Senpai configuration to support all applications (§3.3).

In order to get the best performance and compression ratio for zswap, we experimented with many compression algorithms, including lzo, lz4, and zstd [42], and finally chose zstd as it provided a good compression ratio with a low overhead. We also experimented with various zswap memory pool allocators, including Z3fold, Zbud, and Zsmalloc [43], and finally chose Zsmalloc as it provided the most efficient memory pool and gave the biggest memory savings.

5.2 Limitations and Future Work

Currently, we manually choose the offload backend between zswap and SSD-backed swap depending on the application's memory compressibility and sensitivity to memory-access slowdown. Although we could develop tools to automate the process, a more fundamental solution is for the kernel to manage a hierarchy of offload backends, e.g., automatically using zswap for warmer pages and using SSD for colder or less-compressible pages, as well as including NVM and CXL devices into the memory hierarchy in the future. The kernel reclaim algorithm should dynamically balance across these pools of memory. We are actively working on this architecture.

5.3 Hardware for Memory Offloading

Hardware support can make memory offloading more efficient and effective. Specifically, zswap can benefit from hardware-assisted memory compression and decompression. Currently, we rely on sampling in software to maintain the LRU ordering for memory reclaim and offloading, and the overhead scales with the targeted paging rate. With upcoming bus technologies such as CXL that provide memory-like access semantics, hardware assistance for estimating not only cold memory but also warm memory will help TMO-like techniques work more effectively.

6 RELATED WORK

The work from Lagar-Cavilla et al. [18] is the closest to ours. Their approach swaps cold pages into a compressed in-memory pool, enforces a target rate of swapping-in pages, and uses machine learning to tune the key parameters. The fundamental difference that distinguishes our work lies in the way we measure application performance degradation. Instead of building heuristics around low-level performance indicators based on offline profiling, our approach relies on high-level, real-time pressure metrics that naturally factors in memory access pattern, sensitivity to page faults and hardware characteristics. As a result, TMO is applicable to a much wider variety of production environments.

Remote Memory. Using remote machine's memory to offload cold local memory has been studied before [2, 22, 23]. Infiniswap [13] reduces remote CPU usage and addresses fault tolerance concerns. Fastswap [4] uses a far-memory-aware scheduler to optimize total workload throughput. These systems do not provide fine-grained control over performance degradation and hence are unsuitable for latency-sensitive workloads.

DRAM-NVM hybrid memories and Page Migration Optimizations. The emerging NVM technologies attract many studies on using NVM to extend DRAM with several optimizations [3, 12, 16, 21, 31–33, 36, 38]. Techniques such as prefetching, kernel-object migrations, multi-threaded and huge page migrations, batching and asynchronous memory management operations, as well as compiler support can further improve the overall system performance and enable even more aggressive memory management. Persistent memory file systems [5, 15, 27, 35, 40] often cope with similar problems as TMO in determining how often to migrate data to slower storage mediums and which data to migrate. We believe that PSI could be used in this domain to better account for the performance implications of data placement in such tiered storage systems.

Cold page detection. Some earlier work [20, 41] classifies cold/hot pages based on the accessed bit in page table entry (PTE), which is also used by the kernel's reclaim logic. Idle memory tracking [10] avoids this conflict by introducing an additional page flag. G-swap improves accessed bit accuracy by additionally tracking page age histograms and feeding them to machine learning [18]. [28] samples accessed bit in dynamic sized regions that are subdivided when getting too hot. Our work instead relies on the kernel LRU algorithm to estimate page hotness, which is more efficient than unoptimized or unsorted linear sampling methods. Most previous work uses access rate for a hot/cold page cutoff decision. Thermostat [1] uses estimated access latency based on access rate to better quantify performance impact. Our use of pressure metrics takes a step further by directly measuring performance loss, which is more adaptive to different hardware and workloads.

7 CONCLUSION

This paper introduces TMO, Meta's transparent memory offloading solution for heterogeneous datacenter environments. TMO is composed of multiple components across userspace and the kernel. The Pressure Stall Information (PSI) in the OS kernel tracks the amount of lost work due to the lack of CPU, memory, and I/O. Senpai is a userspace component responsible for controlling how much memory to offload given the workload behavior and hardware characteristics. We further presented a set of kernel optimizations for efficient memory offloading for both anonymous and file-backed memory. Finally, we described our production deployment experience of TMO and future work.

ACKNOWLEDGMENTS

Dimitrios Skarlatos at CMU is funded by NSF under grant CNS 2107307 and a Meta Faculty Award.

REFERENCES

[1] Neha Agarwal and Thomas F Wenisch. 2017. Thermostat: Application-transparent page management for two-tiered main memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems.* 631–644.

[2] Marcos K Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, and Michael Wei. 2017. Remote memory in the age of fast networks. In *Proceedings of the 2017 Symposium on Cloud Computing.* 121–127.

[3] Hasan Al Maruf and Mosharaf Chowdhury. 2020. Effectively prefetching remote memory with leap. In *2020 {USENIX} Annual Technical Conference ({USENIX}{ATC} 20).* 843–857.

[4] Emmanuel Amaro, Christopher Branner-Augmon, Zhihong Luo, Amy Ousterhout, Marcos K Aguilera, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. 2020. Can far memory improve job throughput?. In *Proceedings of the Fifteenth European Conference on Computer Systems.* 1–16.

[5] Thomas E. Anderson, Marco Canini, Jongyul Kim, Dejan Kostić, Youngjin Kwon, Simon Peter, Waleed Reda, Henry N. Schuh, and Emmett Witchel. 2020. Assise: Performance and Availability via Client-local NVM in a Distributed File System. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20).* USENIX Association, 1011–1027. https://www.usenix.org/conference/osdi20/presentation/anderson

[6] Benjamin Berg, Daniel S. Berger, Sara McAllister, Isaac Grosof, Sathya Gunasekar, Jimmy Lu, Michael Uhlar, Jim Carrig, Nathan Beckmann, Mor Harchol-Balter, and Gregory R. Ganger. 2020. The CacheLib Caching Engine: Design and Experiences at Scale. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation.*

[7] Kristof Beyls and Erik D'Hollander. 2001. Reuse distance as a metric for cache behavior. In *Proceedings of the IASTED Conference on Parallel and Distributed Computing and systems*, Vol. 14. Citeseer, 350–360.

[8] Danny Cobb and Amber Huffman. 2012. Nvm express and the pci express ssd revolution. In *Intel Developer Forum*, Vol. 2012. Intel.

[9] Compute Express Link. [n. d.]. https://www.computeexpresslink.org/.

[10] Vladimir Davydov. 2015. Idle memory tracking. https://lwn.net/Articles/639341/. Accessed: 2021-07-19.

[11] Jaeyoung Do, Sudipta Sengupta, and Steven Swanson. 2019. Programmable Solid-State Storage in Future Cloud Datacenters. *Commun. ACM* 62, 6 (may 2019), 54–62. https://doi.org/10.1145/3286588

[12] Subramanya R Dulloor, Amitabha Roy, Zheguang Zhao, Narayanan Sundaram, Nadathur Satish, Rajesh Sankaran, Jeff Jackson, and Karsten Schwan. 2016. Data tiering in heterogeneous memory systems. In *Proceedings of the Eleventh European Conference on Computer Systems.* 1–16.

[13] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G Shin. 2017. Efficient memory disaggregation with infiniswap. In *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17).* 649–667.

[14] Intel. [n. d.]. https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html.

[15] Rohan Kadekodi, Saurabh Kadekodi, Soujanya Ponnapalli, Harshad Shirwadkar, Gregory R. Ganger, Aasheesh Kolli, and Vijay Chidambaram. 2021. WineFS: A Hugepage-Aware File System for Persistent Memory That Ages Gracefully. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21).* Association for Computing Machinery, New York, NY, USA, 804–818. https://doi.org/10.1145/3477132.3483567

[16] Sudarsun Kannan, Yujie Ren, and Abhishek Bhattacharjee. 2021. *KLOCs: Kernel-Level Object Contexts for Heterogeneous Memory Systems.* Association for Computing Machinery, New York, NY, USA, 65–78. https://doi.org/10.1145/3445814.3446745

[17] Emre Kültürsay, Mahmut Kandemir, Anand Sivasubramaniam, and Onur Mutlu. 2013. Evaluating STT-RAM as an energy-efficient main memory alternative. In *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS).* IEEE, 256–267.

[18] Andres Lagar-Cavilla, Junwhan Ahn, Suleiman Souhlal, Neha Agarwal, Radoslaw Burny, Shakeel Butt, Jichuan Chang, Ashwin Chaugule, Nan Deng, Junaid Shahid, Greg Thelen, Kamil Adam Yurtsever, Yu Zhao, and Parthasarathy Ranganathan. 2019. Software-Defined Far Memory in Warehouse-Scale Computers *(ASPLOS '19).* 14 pages.

[19] Seok-Hee Lee. 2016. Technology scaling challenges and opportunities of memory devices. In *2016 IEEE International Electron Devices Meeting (IEDM).* IEEE, 1–1.

[20] Michel Lespinasse. 2011. Idle page tracking / working set estimation. https://lwn.net/Articles/460762/. Accessed: 2021-07-19.

[21] Yang Li, Saugata Ghose, Jongmoo Choi, Jin Sun, Hui Wang, and Onur Mutlu. 2017. Utility-based hybrid memory management. In *2017 IEEE International Conference on Cluster Computing (CLUSTER).* IEEE, 152–165.

[22] Shuang Liang, Ranjit Noronha, and Dhabaleswar K Panda. 2005. Swapping to remote memory over infiniband: An approach using a high performance network block device. In *2005 IEEE International Conference on Cluster Computing.* IEEE, 1–10.

[23] Kevin Lim, Jichuan Chang, Trevor Mudge, Parthasarathy Ranganathan, Steven K Reinhardt, and Thomas F Wenisch. 2009. Disaggregated memory for expansion and sharing in blade servers. *ACM SIGARCH computer architecture news* 37, 3 (2009), 267–278.

[24] Tz-Yi Liu, Tian Hong Yan, Roy Scheuerlein, Yingchang Chen, Jeffrey KoonYee Lee, Gopinath Balakrishnan, Gordon Yee, Henry Zhang, Alex Yap, Jingwen Ouyang, Takahiko Sasaki, Sravanti Addepalli, Ali Al-Shamma, Chin-Yu Chen, Mayank Gupta, Greg Hilton, Saurabh Joshi, Achal Kathuria, Vincent Lai, Deep Masiwal, Masahide Matsumoto, Anurag Nigam, Anil Pai, Jayesh Pakhale, Chang Hua Siau, Xiaoxia Wu, Ronald Yin, Liping Peng, Jang Yong Kang, Sharon Huynh, Huijuan Wang, Nicolas Nagel, Yoichiro Tanaka, Masaaki Higashitani, Tim Minvielle, Chandu Gorla, Takayuki Tsukamoto, Takeshi Yamaguchi, Mutsumi Okajima, Takayuki Okamura, Satoru Takase, Takahiko Hara, Hirofumi Inoue, Luca Fasoli, Mehrdad Mofidi, Ritu Shrivastava, and Khandker Quader. 2013. A 130.7mm<sup>2</sup> 2-layer 32Gb ReRAM memory device in 24nm technology. In *2013 IEEE International Solid-State Circuits Conference Digest of Technical Papers.* 210–211. https://doi.org/10.1109/ISSCC.2013.6487703

[25] Chris A. Mack. 2011. Fifty Years of Moore's Law. *IEEE Transactions on Semiconductor Manufacturing* 24, 2 (2011), 202–207. https://doi.org/10.1109/TSM.2010.2096437

[26] Sara McAllister, Benjamin Berg, Julian Tutuncu-Macias, Juncheng Yang, Sathya Gunasekar, Jimmy Lu, Daniel S. Berger, Nathan Beckmann, and Gregory R. Ganger. 2021. Kangaroo: Caching Billions of Tiny Objects on Flash. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21).* Association for Computing Machinery, New York, NY, USA, 243–262. https://doi.org/10.1145/3477132.3483568

[27] Ian Neal, Gefei Zuo, Eric Shiple, Tanvir Ahmed Khan, Youngjin Kwon, Simon Peter, and Baris Kasikci. 2021. Rethinking File Mapping for Persistent Memory. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. USENIX Association, 97–111. https://www.usenix.org/conference/fast21/presentation/neal

[28] SeongJae Park, Yunjae Lee, and Heon Y Yeom. 2019. Profiling Dynamic Data Access Patterns with Controlled Overhead and Quality. In *Proceedings of the 20th International Middleware Conference Industrial Track*. 1–7.

[29] Persistent Memory Development Kit. [n. d.]. https://pmem.io/pmdk/.

[30] Moinuddin K Qureshi, Vijayalakshmi Srinivasan, and Jude A Rivers. 2009. Scalable high performance main memory system using phase-change memory technology. In *Proceedings of the 36th annual international symposium on Computer architecture*. 24–33.

[31] Luiz E Ramos, Eugene Gorbatov, and Ricardo Bianchini. 2011. Page placement in hybrid memory systems. In *Proceedings of the international conference on Supercomputing*. 85–95.

[32] Amanda Raybuck, Tim Stamler, Wei Zhang, Mattan Erez, and Simon Peter. 2021. HeMem: Scalable Tiered Memory Management for Big Data Applications and Real NVM. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 392–407. https://doi.org/10.1145/3477132.3483550

[33] Thomas Shull, Jian Huang, and Josep Torrellas. 2019. AutoPersist: An Easy-to-Use Java NVM Framework Based on Reachability. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 316–332. https://doi.org/10.1145/3314221.3314608

[34] Zixuan Wang, Xiao Liu, Jian Yang, Theodore Michailidis, Steven Swanson, and Jishen Zhao. 2020. Characterizing and Modeling Non-Volatile Memory Systems. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 496–508. https://doi.org/10.1109/MICRO50266.2020.00049

[35] Jian Xu and Steven Swanson. 2016. NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*. USENIX Association, Santa Clara, CA, 323–338. https://www.usenix.org/conference/fast16/technical-sessions/presentation/xu

[36] Zi Yan, Daniel Lustig, David Nellans, and Abhishek Bhattacharjee. 2019. Nimble page management for tiered memory systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 331–345.

[37] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. 2020. An Empirical Guide to the Behavior and Use of Scalable Persistent Memory. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. USENIX Association, Santa Clara, CA, 169–182. https://www.usenix.org/conference/fast20/presentation/yang

[38] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steven Swanson. 2020. An Empirical Guide to the Behavior and Use of Scalable Persistent Memory. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies* (Santa Clara, CA, USA) *(FAST'20)*. USENIX Association, USA, 169–182.

[39] Wangyuan Zhang and Tao Li. 2009. Exploring phase change memory and 3D die-stacking for power/thermal friendly, fast and durable memory architectures. In *2009 18th International Conference on Parallel Architectures and Compilation Techniques*. IEEE, 101–112.

[40] Shengan Zheng, Morteza Hoseinzadeh, and Steven Swanson. 2019. Ziggurat: A Tiered File System for Non-Volatile Main Memories and Disks. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*. USENIX Association, Boston, MA, 207–219. https://www.usenix.org/conference/fast19/presentation/zheng

[41] Pin Zhou, Vivek Pandey, Jagadeesan Sundaresan, Anand Raghuraman, Yuanyuan Zhou, and Sanjeev Kumar. 2004. Dynamic tracking of page miss ratio curve for memory management. *ACM SIGPLAN Notices* 39, 11 (2004), 177–188.

[42] Zstandard. [n. d.]. https://en.wikipedia.org/wiki/Zstandard.

[43] zswap. [n. d.]. https://www.kernel.org/doc/html/latest/vm/zswap.html.