Software-Defined Far Memory in Warehouse-Scale Computers

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
{andreslc,junwhan,suleiman,nehaagarwal,rburny,shakeelb,jichuan,
ashwinch,dengnan,junaids,gthelen,kyurtsever,yuzhao,parthas}@google.com
Google

Abstract

Increasing memory demand and slowdown in technology scaling pose important challenges to total cost of ownership (TCO) of warehouse-scale computers (WSCs). One promising idea to reduce the memory TCO is to add a cheaper, but slower, "far memory" tier and use it to store infrequently accessed (or *cold*) data. However, introducing a far memory tier brings new challenges around dynamically responding to workload diversity and churn, minimizing stranding of capacity, and addressing brownfield (legacy) deployments.

We present a novel software-defined approach to far memory that proactively compresses cold memory pages to effectively create a far memory tier in software. Our end-to-end system design encompasses new methods to define performance service-level objectives (SLOs), a mechanism to identify cold memory pages while meeting the SLO, and our implementation in the OS kernel and node agent. Additionally, we design learning-based autotuning to periodically adapt our design to fleet-wide changes without a human in the loop. Our system has been successfully deployed across Google's WSC since 2016, serving thousands of production services. Our software-defined far memory is significantly cheaper (67% or higher memory cost reduction) at relatively good access speeds (6 µs) and allows us to store a significant fraction of infrequently accessed data (on average, 20%), translating to significant TCO savings at warehouse scale.

CCS Concepts • **Computer systems organization → Distributed architectures**; • **Software and its engineering → Memory management**.

Keywords cold data, far memory, machine learning, memory, warehouse-scale computers, zswap

ASPLOS '19, April 13–17, 2019, Providence, RI, USA

ACM ISBN 978-1-4503-6240-5/19/04.
https://doi.org/10.1145/3297858.3304053

ACM Reference Format:
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. In *2019 Architectural Support for Programming Languages and Operating Systems (ASPLOS '19), April 13–17, 2019, Providence, RI, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3297858.3304053

1 Introduction

Effectively scaling out a warehouse-scale computer (WSC) [7] requires that all resource types be scaled in a balanced manner, so that the overall resource ratios between compute, memory, storage, and networking can satisfy the aggregate workload requirements. Failing to scale one resource type causes the others to be stranded, hurting the cost per performance of the entire WSC. Consequently, scaling out WSCs is often limited by the components that have the weakest scaling behavior in terms of both performance and cost effectiveness.

In recent years, DRAM has become a critical bottleneck for scaling the WSCs. The slowdown of device-level scaling (the end of Moore's law [35]) prevents the reduction in cost per GB of DRAM [25, 27]. At the same time, the prevalence of in-memory computing, particularly for big-data workloads, has caused an explosive growth in DRAM demand. These two trends have resulted in a global DRAM supply shortage in recent years, posing serious challenges to the cost-effective scaling of WSCs.

One promising direction that has been previously proposed to reduce the cost of memory ownership is the introduction of second-tier memory or *far memory*. Far memory is a tier between DRAM and Flash that has lower cost per GB than DRAM and higher performance than Flash. By introducing far memory into the memory hierarchy and storing infrequently accessed (or *cold*) data into far memory, the system can perform the same jobs with a lower DRAM capacity or pack more jobs to each machine, both of which reduce the total cost of ownership (TCO).

Modern WSCs and applications running on them have the following characteristics that demand unique requirements when deploying second-tier memory:

Near-zero tolerance to application slowdown. WSCs are sensitive to cost per performance because of their economies of scale [7]. While adoption of far memory realizes savings in the memory aspect of WSC TCO, slowdown in applications can result in both irrecoverable Service Level Agreement (SLA) violations and the need to increase provisioned capacity to offset the performance loss due to extra time spent on far memory tier. Slowdowns of even a few percentage points introduced by the adoption of far memory can thus offset all potential TCO savings.

Heterogeneity of applications. Applications running on modern WSCs are becoming more numerous and diverse [7, 24]. Such heterogeneity makes per-application optimization for far memory impractical and necessitates a transparent yet robust mechanism for effectively utilizing far memory.

Dynamic cold memory behavior. WSCs exhibit dynamic changes in job mixes and/or utilization (e.g., diurnal patterns), introducing variability in the effective size of memory per machine that can exploit far memory technologies. Consequently, the optimal ratio between near and far memory not only depends on the workloads running at a given time but also shifts over time. Therefore, it is desirable to de-risk the TCO impact of stranded far memory capacity, or alternatively be flexible in provisioning.

In this paper, we address the aforementioned challenges to WSC cost of ownership, presenting our system design and experience in adopting far memory approach within Google's production WSCs at scale. Specifically, we make the following contributions:

- We present a fleet-wide longitudinal characterization of a real-world WSC that quantifies the large variability in the amount of available cold memory per machine. We show that cold memory ranges from 1% to 61% across different clusters and from 1% to 52% even within the same cluster based on application mix and the time of day. Such ranges warrant the need for flexible far memory provisioning instead of fixed-capacity far memory.
- We showcase a *software-defined* approach to far memory, one that is readily available, offers flexibility, and improves time to market, making memory TCO tenable. Specifically, we demonstrate that zswap [1], a Linux kernel mechanism that stores memory compressed in DRAM, can be used to implement software-defined far memory that provides *single-digit μs of latency at tail*. We also show that our proactive approach to move cold pages to slower far memory works favorably in reaping memory capacity from pages with low access rates, as opposed to a reactive approach under memory pressure on a machine.
- We discuss the design and implementation of our approach. Our control plane consists of (1) a kernel mechanism that collects memory access statistics and swaps out cold memory pages to far memory and (2) a node agent that controls the aggressiveness of the kernel mechanism based on application behavior. Our design stands on a well-defined Service Level Objective (SLO) and can be generalized to other types of far memory devices.
- We implement an autotuning system that uses machine learning to optimize the control plane based on the fleet-wide behavior. It consists of a fast far memory model estimating the far memory behavior of the entire WSC under different configurations and design space exploration guided by a machine learning algorithm called Gaussian Process (GP) Bandit [17, 21, 39]. This facilitates the whole system to adapt to long-term behavior shifts of the entire WSC.
- We present evaluation data from real-world use cases including longitudinal studies across a mix of production workloads and a case study with Bigtable [10]. Our system can migrate 20–30% of infrequently used data, facilitating 4–5% savings in memory TCO (millions of dollars at WSC scale), while having a negligible impact on a diverse mix of applications. Our machine-learning-based autotuner improves the efficiency of our system by an additional 30% relative to heuristic-based approaches.

2 Background and Motivation

2.1 Far Memory

Far memory is a tier between DRAM and Flash that provides lower cost per GB than DRAM and higher performance than Flash. In this section, we provide an overview of far memory technologies that have been studied by prior work and discuss their characteristics from the WSC perspective.

Non-volatile memory. Non-volatile memory (NVM) is an emerging memory technology that realizes higher density (thus lower cost per GB) than DRAM and persistency based on new materials. To date, relative to DRAM, most of the NVM technologies show higher latency (from hundreds of ns to tens of μs), lower bandwidth (single-digit GB/s), and read/write asymmetry (i.e., writes are slower than reads).

In terms of access interface, two types of NVM devices are available in the market: memory bus (e.g., NVDIMM-P [37], Intel Optane DC Persistence Memory [20], etc.) and PCIe bus (e.g., Intel Optane DC SSD [20], Samsung Z-SSD [38], etc.). The main difference between the two is that the former allows load/store accesses to NVM at a cache block granularity, while the latter is available through a page-granular access interface like storage devices and the data has to be copied from NVM to main memory before accessing it. Because of this, the former often provides faster access to data stored in NVM but requires hardware support from the CPU side.

Many current NVM devices are available only in fixed predetermined sizes. This can potentially lead to resource stranding in the context of WSCs.

Remote memory. Memory disaggregation [30] is an approach of using remote machines' memory as a swap device. It is implemented by utilizing unused memory in remote machines [18, 29], or by building memory appliances whose only purpose is to provide a pool of memory shared by many machines [30, 31]. Both styles of implementation reduce the need for over-provisioning the memory capacity per machine by balancing the memory usage across the machines in a cluster. Accessing a remote page takes one to tens of µs, depending on the cluster size and network fabric speed.

Remote memory has interesting challenges to be addressed in the context of WSCs before it can be deployed to realize memory TCO savings [6]. First, swapping out memory pages to remote machines expands the failure domain of each machine, which makes the cluster more susceptible to catastrophic failures. Second, pages that are being swapped out have to be encrypted before leaving the machine in order to comply with the stringent security requirements that are often set by WSC applications processing sensitive information. Third, many WSC applications are sensitive to tail latency [7], but bounding tail latency is harder for a cluster or a rack than for a machine.

2.2 Far Memory in a Real-World WSC: Opportunities and Challenges

In this subsection, we analyze the aggregate cold memory behavior of Google's WSC [24] and highlight opportunities and challenges in large-scale deployment of far memory in WSCs. This insight is critical towards designing a far memory system for WSCs because the cold memory behavior of workloads directly correlates with the efficacy of far memory. For example, applications with many cold memory pages are likely to benefit more from far memory than those with fewer cold memory pages.

There are many approaches one can take to define the *coldness* of a memory page. We focus on a definition that draws from the following two principles: (1) the value of temporal locality, by classifying as cold a memory page that has not been accessed beyond a threshold of T seconds; (2) a proxy for the application effect of far memory, by measuring the rate of accesses to cold memory pages, called *promotion rate*. These two principles are the cornerstone of our cold page identification mechanism (explained in Section 4).

Figure 1 shows the fleet-wide average of the percentage of cold memory and the promotion rate of each job running in the WSC under different T values. Since a lower T value classifies pages as cold at an earlier stage of their lifetime, it identifies more cold pages. At the most aggressive setting of $T = 120$ s, we observe that 32% of memory usage is cold on average. This large fraction of cold memory demonstrates the huge potential from far memory in real-world WSCs.

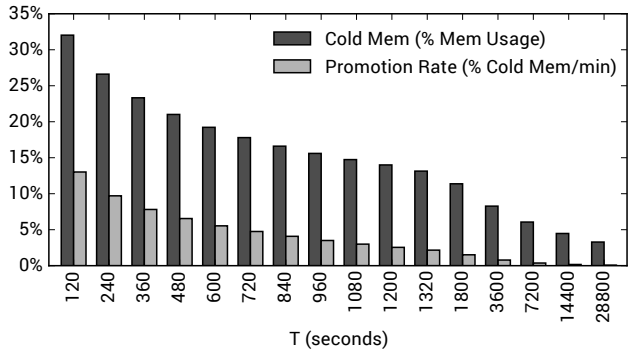


Figure 1. The average percentage of cold memory and promotion rate (y-axis; units specified in the legend) under different cold age thresholds (x-axis).

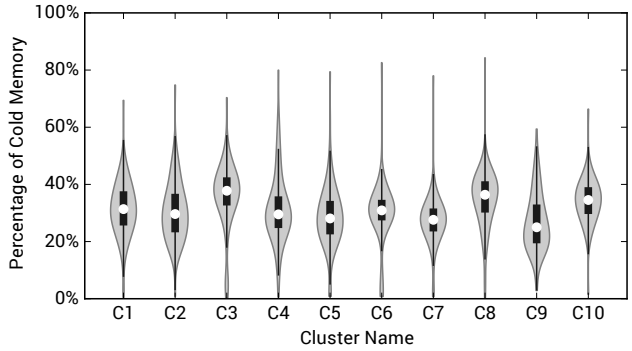


Figure 2. Cold memory variation across the machines in the top 10 largest clusters. The plot shows overall distribution (violins), median (circles), the first/third quartiles (boxes), and 1.5 interquartile range (IQR) from the first/third quartiles (whiskers) of cold memory percentage for each cluster.

On the other hand, the performance overhead of far memory increases as the system becomes more aggressive in identifying cold memory. At $T = 120$ s, applications access 15% of their total cold memory every minute on average. Depending on the relative performance difference between near memory and far memory, this access rate may noticeably degrade the application performance, offsetting the TCO savings from far memory.

This trade-off relationship between the percentage of cold memory and the performance overhead motivates the need for a robust control algorithm that can maximize the former while minimizing the latter. For the rest of this subsection, we will assume $T = 120$ s for simplicity.

Figure 2 illustrates the distribution of the percentage of cold memory per machine (i.e., total size of cold memory divided by memory usage in each machine) across 10 clusters of up to tens of thousands of machines each. We find that the percentage of cold memory varies from 1% to 52%, even

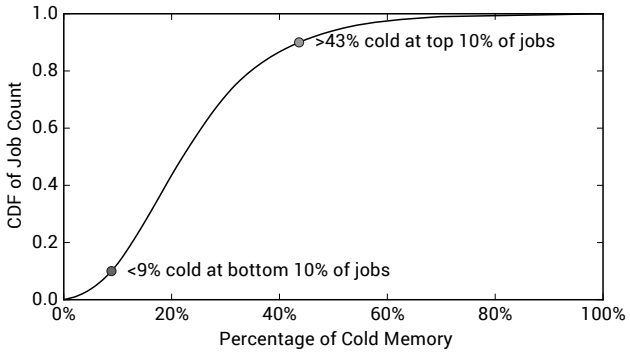


Figure 3. Cold memory variation across jobs. The plot shows the cumulative job distribution with respect to the percentage of cold memory.

within a cluster. If we provision the far memory capacity per machine to be 20% of the total memory capacity, some machines will have up to 30% more cold memory than the available far memory capacity. On the other hand, provisioning at 50% will result in insufficient near memory capacity in some machines, leading to too much performance overhead. Neither approach is favorable from a TCO perspective.

Moreover, application behavior adds another dimension of variability. Figure 3 depicts how the percentage of cold memory in each job (averaged across the job execution) is distributed. For the top 10% of jobs, at least 43% of memory usage is cold; this percentage drops to below 9% for the bottom 10% of the jobs. Such heterogeneity, along with a large number of jobs running on modern WSCs, makes per-application optimization for far memory impractical.

In summary, storing cold memory to cheaper but slower far memory has great potential of saving TCO in WSCs. But for this to be realized in a practical manner, the system has to (1) be able to accurately control its aggressiveness to minimize the impact on application performance and (2) be resilient to the variation of cold memory behavior across different machines, clusters, and jobs.

3 Software-defined Far Memory

We show a *software-defined* approach to far memory implementation, which we have adopted and deployed at Google. In particular, we propose to adopt *zswap* [1] as a far memory solution that is readily available today. In the Linux kernel (3.11+), zswap works as a swap device, which is to avoid the machine from running out of memory. Instead, we take a proactive approach towards storing cold compressed pages in memory using zswap and implement far memory in software. Compressing memory pages allows us to pack more data in memory (i.e., lower cost per GB) at a cost of increased access time. At a high level, this is no different from any other far memory implementation from a TCO perspective.

3.1 Advantages of Software-defined Far Memory

In the context of Google WSCs, zswap-based far memory provided the following benefits that were key to early deployment over other types of far memory devices.

Reliability. zswap confines failure domain within a machine, limiting catastrophic failures to a single machine, while avoiding security and reliability challenges of remote memory. Also, it does not introduce additional hardware components, which simplifies the system design, implementation, testing, deployment, and monitoring.

Time to deployment. zswap, being a software approach, can be deployed with much shorter time and lower effort, and does not require cross-vendor collaborations as there is no need of any special hardware (e.g., NVM). Moreover, some NVM technologies require deploying a new server platform (e.g., new CPUs with special memory controllers), making it impossible to retrofit far memory to older server generations, and thus, limiting their deployments to the rate of new platforms. Note that the velocity of deployment is critical for WSCs because quick deployment of a readily available technology and harvesting its benefits for a longer period of time is more economical than waiting for a few years to deploy newer platforms promising potentially bigger TCO savings.

No new hardware cost. Instead of adding hardware cost as in other far memory technologies, zswap trades off CPU cycles (for compression and decompression) with memory savings. Considering availability of idle cycles in WSCs due to large variation in CPU usage [36], such extra CPU cycles can be serviced in general for free, thereby minimizing the cost of far memory itself.

Adaptive to dynamic application behavior. Shift in job mixes and memory access patterns in WSCs (Section 2.2) result in variability of the size of cold memory per machine and across machines. zswap can adapt to this variation by dynamically resizing memory capacity available to the jobs, by compressing more/less memory. Even with such dynamism, zswap can still realize memory CapEx savings because averaging across tens of thousands of machines makes memory savings stable *at the cluster level*, which is how we provision capacity (Section 6.1). This differentiates zswap from far memory solutions whose capacity cannot be easily changed once deployed. Furthermore, it enables shorter turnaround time for experimenting with different settings without changing the hardware configuration at scale.

3.2 Challenges in Software-defined Far Memory

The control mechanism for far memory in WSCs requires (1) tight control over performance slowdowns to meet defined SLOs and (2) low CPU overhead so as to maximize the TCO savings from far memory. Although zswap exhibits

favorable properties for far memory design in WSCs, its control plane does not meet the above criteria. This is because zswap in the Linux kernel, when enabled, is triggered only on direct reclaim (i.e., when a host memory node runs out of memory) and tries to compress pages until it makes enough room to avoid out-of-memory situations, stalling application allocations. This mechanism has the following shortcomings: (1) the performance overhead due to zswap decompression is unbounded, (2) its bursty compression overhead at the last minute negatively affects the tail latencies hurting Service-Level Indicators (SLIs) of WSC applications, and (3) memory savings are not materialized until the machines are fully saturated. In fact, we did evaluate this approach during our deployment but observed noticeable degradation in application performance, negatively impacting TCO.

Additionally, not every piece of data in WSC's DRAM is amenable to savings from compression, resulting in an opportunity cost of wasted cycles when zswap chooses to compress such data. For example, video data in memory may not be as compressible as textual data.

Therefore, in this paper, we design an end-to-end warehouse-scale system that identifies cold pages and proactively migrates them to far memory while treating performance as a first-class constraint. The key question is *how cold is cold? or what is the definition of a cold page?* The quality of the cold page identification algorithm will impact both memory savings and application impact.

4 Cold Page Identification Mechanism

Our goal is to design a robust and effective control plane for large-scale deployment of zswap. As in zswap in the Linux kernel or other swap mechanisms, our system works at a OS page granularity when migrating pages between near memory (e.g., DRAM) and far memory (e.g., zswap). This enables far memory adoption with no hardware modifications.

The primary difference from the existing zswap mechanism is around *when to compress pages, or when to migrate pages from near memory to far memory.* Unlike zswap in the Linux kernel, our system identifies cold memory pages in the background and *proactively* compresses them, so that the extra free memory can be used to schedule more jobs to the machine. Once a compressed page is accessed, zswap decompresses the page and keeps it in a decompressed state from then on to avoid repeatedly incurring decompression. Such pages become eligible for compression again when they become cold in future.

The efficacy of our system heavily depends on how accurately it identifies cold pages. Below, we explain our mechanism for cold page identification.

4.1 Definition of Cold Pages

Our system identifies cold pages based on the time since the last access to each page, or simply, *age*. A page is considered cold when it has not been accessed for more than T seconds. We call T the *cold age threshold* and it determines how aggressively the system identifies cold memory pages. We base this mechanism on prior work [28, 42, 46].

The cold age threshold has a direct impact on both memory savings and performance overhead. Classifying pages as cold can prematurely map them to far memory, causing performance degradation. Our system tries to find the lowest cold age threshold that satisfies the given performance constraints in order to maximize the memory savings under a well-defined SLO, which we discuss next.

4.2 Performance SLO for Far Memory

In WSC environments, directly correlating the impact of the cold memory threshold on application performance is challenging because of the diverse nature of performance metrics across different applications, which themselves can range from latency-sensitive (e.g., user-facing web frontend) to throughput-oriented (e.g., machine learning training pipelines). Therefore, we define a low-level indicator that is easy to measure in an application-agnostic manner but still correlates with the performance overhead from far memory.

Promotion rate. The performance overhead of far memory comes from accessing pages that are stored in far memory (we call such an operation *promotion*). Thus, we define *promotion rate*, the rate of swapping in pages from far memory to near memory, and use it as an SLI for far memory performance. Since pages in far memory are migrated to near memory once they are accessed, the promotion rate is equivalent to the number of *unique* pages in far memory that are accessed in a unit time.

Target promotion rate. Different applications have different levels of performance sensitivity to promotion rate. For example, at the same level of an *absolute* promotion rate, small jobs are more likely to experience higher performance overhead than big jobs because of a potentially higher fraction of far memory accesses in the former. This necessitates a way to *normalize* the absolute promotion rate by a metric that represents how "big" each job is.

Therefore, we design our system to keep the promotion rate below $P\%$ of the application's working set size per minute, which serves as a Service Level Objective (SLO) for far memory performance. We define the working set size of an application as the total number of pages that are accessed within minimum cold age threshold (120 s in our system). The working set size per minute serves as a proxy of job's memory bandwidth usage, which, based on our evaluation, correlates with job's performance sensitivity to far memory accesses. Our SLO ensures that no more than $P\%$ of the working set of an application is from far memory, thereby limiting the performance overhead of far memory.

The exact value of P depends on the performance difference between near memory and far memory. For our deployment, we conducted months-long A/B testing at scale with production workloads and empirically determined P to be 0.2%/min. At this level of a target promotion rate, the compression/decompression rate of a job is low enough to not interfere with other colocated jobs in the same machine.

Enforcing the promotion rate to be lower than the target prevents bursty decompression from applications because it limits the rate of decompression by definition. In the rare cases where aggressive or correlated decompression bursts cause the machine to run out of memory for decompressing compressed pages, we selectively evict low-priority jobs by killing them and rescheduling them on other machines. Our WSC control plane [40] offers an eviction SLO to users, which has never been breached in 18 months in production while we realized memory savings.

4.3 Controlling the Cold Age Threshold

To determine the lowest cold age threshold that meets the promotion rate SLO, we estimate the promotion rate of an application for different cold age thresholds. For this purpose, we build a *promotion histogram* for each job in the OS kernel, where, for each cold page threshold T, we record the total promotion rate of pages that are colder than the threshold T. As an example, let's assume a case where an application has two memory pages, A and B, that were accessed 5 and 10 minutes ago, respectively, and both pages were accessed again 1 minute ago. In this scenario, the promotion histogram returns 1 promotion/min for $T = 8$ min because only B would have been considered cold under $T = 8$ min when the two pages were accessed one minute ago. Similarly, it returns 2 promotions/min for $T = 2$ min since now both A and B would have been considered cold under $T = 2$ min. We discuss our implementation in Section 5.1.

While the promotion histogram lets us choose the best cold age threshold *for the past*, it is not necessarily the best threshold for the future. Ideally, the control algorithm has to give us a stable threshold over time so as to reduce unnecessary compression and decompression costs. At the same time, the system has to be responsive to sudden spikes in application activity and avoid compressing too much memory for a prolonged period of time. Thus, our system controls the threshold based on the following principles:

- It keeps track of the best cold age threshold of each 1-minute period in the past and uses their K-th percentile as the threshold for the next one minute. By doing so, it will violate the SLO for approximately $(100 - K)\%$ of the times under the steady state.
- If the best cold age threshold from the last one minute is higher than the K-th percentile from the past (i.e., jobs accessing more cold memory during the last one minute than the K-th percentile of the past behavior), we use the former so that the system can quickly react to sudden hike in application activity.
- Since the algorithm relies on per-job history, we disable zswap for the first S seconds of job execution to avoid making decisions based on insufficient information.

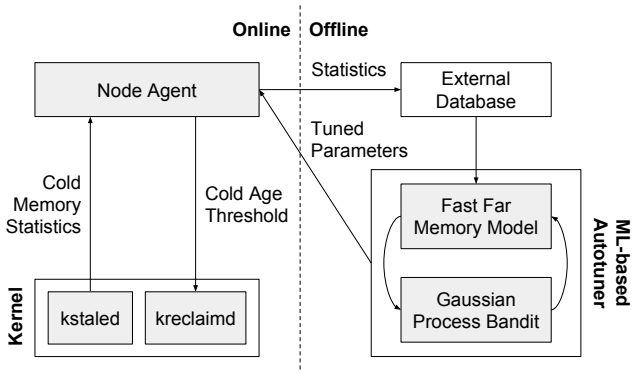


Figure 4. Overall system design.

4.4 Calculating the Size of Cold Memory

The last piece of our mechanism is estimating the size of cold memory under different cold age thresholds. For this, our system builds per-job *cold page histogram* for a given set of predefined cold age thresholds. In this histogram, for each cold age threshold T, we record the number of pages that have not been accessed for at least T seconds. This information is used to (1) estimate the working set size of a job (used to normalize the promotion rate; see Section 4.2) and (2) perform offline analysis for potential memory savings under different cold age thresholds (Section 5.3).

5 System Implementation

In this section, we discuss the implementation of our cold page identification mechanism in the OS kernel and node agent and the autotuning system for it. We present and evaluate our system design based on zswap, a readily available Linux feature, but our design can be generalized to other types of far memory technologies as our control plane is not tied to any specific far memory device.

Figure 4 illustrates the high-level diagram of our system design. On each production machine, our customized Linux kernel (Section 5.1) and a node agent (Section 5.2) adjust the cold age threshold and collect statistics for each job. Using jobs' historical data, we use machine learning to tune the parameters for node agents (Section 5.3).

5.1 Kernel

We use Linux's memory cgroup (*memcg*) [2] to isolate jobs in our WSC. The software-defined far memory is built on top of zswap. We run two machine-wide kernel daemons, namely *kstaled* and *kreclaimd*, to collect far memory statistics and

move pages to far memory. We first discuss our zswap implementation and the modifications needed for non-disruptive deployment in our production environment.

zswap. We augment upstream Linux's zswap implementation with several features tailored to WSC deployment. We use the *lzo* algorithm to achieve low CPU overhead for compression and decompression.[1]

Once zswap compresses a page, it allocates memory to store the compression payload. We use *zsmalloc* as the compressed data arena. We maintain a global zsmalloc arena per machine, with an explicit compaction interface that can be triggered by the node agent when needed. While a per-memcg zsmalloc arena seems more intuitive given that we encapsulate jobs with memcgs, it leads to varying degrees of external fragmentation in each zsmalloc arena because WSCs often pack tens or hundreds of jobs per machine. Our initial study with per-memcg zsmalloc arenas found thousands of instances per day of arenas fragmented to the point of negative gains.

Empirically, there are no gains to be derived by storing zsmalloc payloads larger than 2990 bytes (73% of a 4 KiB x86 page), where metadata overhead becomes higher than savings from compressing the page. When attempting to compress a page that yields a payload larger than that, we mark the page as *incompressible* and reject it. The incompressible state prevents zswap from attempting to re-compress that page and is cleared when *kstaled* (see below) detects any of the PTEs associated with the page have become dirty.

When a job reaches its memory limit, we turn off zswap instead of using it as a swap device. This is because WSC applications prefer "failing fast" and restarting elsewhere in the cluster by relying on their fault-tolerant nature [7], rather than wasting CPU cycles in the kernel mode trying to stave off job preemption. It also makes zswap coherent with typical behavior of the cluster-wide scheduler, which kills best-effort jobs when they run out of memory.

When a machine runs out of memory, the kernel will use direct memory reclaim in the context of a faulting process. The node agent maintains a "soft" limit for each memcg equivalent to its working set size (determined with the approach in Section 4.2) and the kernel does not reclaim below this threshold. This protects the working set of high-priority jobs and prevents reclaiming job threads from spending excessive cycles doing zswap, while reinforcing the preference of "failing-fast" low-priority jobs.

kstaled. We periodically scan the *accessed* bit present in page table entries to infer if a page is accessed in a given time period [4, 19]. We leverage *kstaled*, a kernel daemon, to track the age of all physical pages eligible for memory reclaim based on *accessed* bit information [28]. Note that *accessed* bit is set by the MMU whenever a physical page is accessed and it is up to the software to clear it [4, 19].

Across each scan period, *kstaled* walks process page tables to read the *accessed* bit for each physical page. If the *accessed* bit for a page is found to be set, *kstaled* sets the age of the corresponding page to zero; otherwise, it increases the age. It also clears the *accessed* bit to detect any future access to the page. If a physical page is mapped in multiple page tables, *kstaled* increases the age only if the *accessed* bit is not set in any of the page tables.

Our version of *kstaled* stores the age of a page in a per-page metadata structure, such that other kernel components that use page access patterns, like direct reclaim, can avail themselves of the information that *kstaled* has already reaped. We use 8 bits per page to encode the age of a page. As we pack these bits in the `struct page` metadata structure already maintained in Linux kernel, we do not incur any storage overhead for tracking the ages. We run *kstaled* at a frequency of 120 s. With 8-bit ages, we can track up to 8.5 hours (= 255×120 s) of ages.

Whenever *kstaled* updates the age of a page, it also updates two per-job histograms: (1) cold age histogram, a histogram over page ages that tracks time T for which pages have not been accessed and (2) promotion histogram, a histogram recording the age of the page when it is accessed. These histograms are exported to the node agent and are used to determine the cold page age threshold as discussed in Section 4.3.

To minimize the CPU overhead of *kstaled*, we empirically tune its scan period while trading off for finer-grained page access information. On average, *kstaled* consumes less than 11% of a single logical CPU core while running as *nice*d background task.

kreclaimd. Once the node agent consumes the age histograms built by *kstaled* and sets the cold age threshold, *kreclaimd* compares the age of each page with the cold age threshold of the job the page belongs to and reclaims all the pages whose age exceeds the threshold. We reclaim cold pages in DRAM by moving them to zswap, thus freeing up DRAM capacity for serving hot application pages. A compressed page is decompressed when accessed. *kreclaimd* runs in slack cycles not used by any other application, realizing memory gains as an unobtrusive background task.

Note that we only consider pages that are on the least recently used (LRU) list [3] to be mapped to far memory. For example, we do not map pages to far memory if they are marked unevictable or locked in memory (mlocked). This helps us prevent wasting CPU cycles on unmovable pages.

5.2 Node Agent

The node agent running on each machine (called *Borglet* in our cluster management system [40]) dynamically controls

[1] We compared several compression algorithms, including lzo, lz4, and snappy, and concluded that lzo shows the best trade-off between compression speed and efficiency.

the cold age threshold in a per-job basis. Using the algorithm described in Section 4.3, it builds a pool of the best cold age thresholds in the past by reading the kernel statistics every minute and calculating the smallest cold age threshold for the past one minute that does not violate the target promotion rate. Then, it enables zswap S seconds after the beginning of each job execution by setting the threshold to the K-th percentile from the pool for the next one minute. The node agent periodically exports every job's cold memory statistics from the kernel to an external database in order to facilitate offline analysis and monitoring.

5.3 ML-based Autotuner

Our system exposes K and S, discussed before, as tunable parameters to adjust the aggressiveness of the control plane. However, manual one-off tuning of these parameters involves many iterations of trial and error with A/B testing in production systems, which is risky, time-consuming, and susceptible to workload behavior shift over time.

In order to address this challenge, we design a pipeline that autotunes such parameters based on the application behavior measured across the entire WSC and automatically deploys new parameters to the production system. The key ideas behind our autotuner are (1) a fast far memory model, which facilitates what-if analysis of memory savings and performance SLO under different parameter configurations, and (2) design space exploration based on machine learning, which makes it feasible to identify fleet-wide optimal parameter values in less than a day. The following explains the key components of our autotuner in more detail.

Problem formulation. Finding the optimal values for parameter K and S is an optimization problem where the goal is to maximize memory savings while meeting the performance SLO. In order to express this problem in a way that can be calculated based on our cold page identification mechanism, we formulate our problem as maximizing the size of cold memory across the fleet (Section 4.4), while ensuring that the fleet-wide promotion rate at the 98th percentile is below the target SLO (Section 4.2). This allows us to maximize memory savings while not hurting the application performance *at tail*, which is critical for WSC applications.

Importantly, our problem formulation facilitates *offline* what-if analysis under different parameter values. Our cold page identification mechanism relies only on the working set size, the promotion histogram, and the cold page histogram of each job over time, all of which are exported to an external database by the node agent. From this information, we can calculate the size of cold memory and the promotion rate not only under the actual cold age threshold used in the fleet *but also for any possible threshold* by leveraging the histograms that contain information about all possible thresholds. In other words, we can emulate our control algorithm offline with any parameter configuration and estimate the size of cold memory and the promotion rate over time under that configuration.

Fast far memory model. For offline what-if analysis of parameter values, we build a fast far memory model, a MapReduce-style [9, 11] distributed pipeline that consumes far memory traces and runs the control algorithm in Section 4.3 with a given parameter configuration. Each far memory trace entry includes job's working set size, promotion histogram, and cold page histogram, aggregated over a 5-minute period. The traces are collected by an existing telemetry infrastructure [36]. The pipeline reports the size of cold memory and 98th percentile fleet-wide promotion rate under a given configuration, which serve as the objective and the constraint of our parameter tuning as explained above.

Our implementation provides excellent scalability because replaying traces from different jobs is independent, and hence can be parallelized, and multiple parameter configurations can be modeled in parallel as well. Consequently, our model is capable of *modeling one week of the entire WSC's far memory behavior* in less than an hour, facilitating rapid exploration without introducing risks to production systems.

ML-based autotuning. Even with our fast far memory model, manually exploring the parameter values is still challenging as there are hundreds of valid configurations and the search space grows exponentially as we add more parameters to the system.

In order to address this challenge, we use a state-of-the-art machine learning algorithm for black-box optimization called *Gaussian Process (GP) Bandit* [17, 21, 39]. GP Bandit learns the shape of search space and guides parameter search towards the optimal point with the minimal number of trials. GP Bandit has been effective in solving complex black-box optimization problems, e.g., optimizing hyperparameters for deep learning models [17]. To the best of our knowledge, this is the first use of GP Bandit for optimizing a WSC.

Our pipeline explores the parameter search space by iterating on the following three steps:

1. Run GP bandit over the existing observations and obtain the parameter configurations to be explored.
2. Run the far memory model with a one week trace from the entire WSC and estimate the size of cold memory and the promotion rate under each configuration.
3. Add new observations to the pool and go back to Step 1 until the maximum number of iterations is reached.

The best parameter configuration found by the pipeline is periodically deployed to the entire WSC. The deployment happens in multiple stages from qualification to production with rigorous monitoring at each stage in order to detect bad configurations and roll back if necessary before causing a large-scale impact.

The key advantage of using machine learning for parameter autotuning is its adaptability. As we highlighted in Section 4, the efficacy of far memory depends heavily on workload characteristics and underlying far memory implementations. This implies that parameter re-tuning is needed for any change in WSCs, such as deploying different types of far memory devices, adding more parameters, or fleet behavior shifts over time. The ML's capability of learning the problem with zero guidance greatly simplifies this continuous innovation without worrying about the effort and complexity of re-tuning the system, which is hard to achieve with manual tuning or handcrafted heuristics.

6 Evaluation

We deployed our far memory system implementation to Google's WSC, spread geographically across multiple data centers serving hundreds of thousands of production services, and measured the metrics to study far memory performance and overheads. Our typical machine configuration is described in Chapter 3 of [7]. For example, an Intel 18-core Haswell based 2-socket server has 16 DIMM slots and supports up to two DIMMs per memory channel.

6.1 Cold Memory Coverage

In this section, we show *cold memory coverage* as a metric to represent the efficacy of our system. We define cold memory coverage as the total size of memory that is stored in far memory (i.e., compressed) divided by the total size of cold memory under the lowest possible cold age threshold (120 s in our system). Conceptually, this is the percentage of cold memory that is stored in far memory and implies how close our system is from the upper bound where all pages that have not been accessed for 120 s or longer can be stored in far memory with no performance degradation.

Figure 5 shows the fleet-wide average of cold memory coverage over time, with relevant timeline annotated. The first stage of the roll-out (A to B) deployed zswap with static parameter values informed by a limited set of small-scale experiments. Then, in the second stage (C to D), we rolled out our autotuner and its parameter value suggestions.

After the initial roll-out, zswap with manually tuned parameters (between B and C) achieved 15% of stable cold memory coverage. On top of that, the ML-based autotuner increased the cold memory coverage to 20%, which corresponds to a 30% increase from the initial stage. This improvement showcases the effectiveness of our autotuner as a system for optimizing WSC configurations without a human in the loop.

Figure 6 shows the distribution of cold memory coverage across the machines in the top 10 largest clusters. As in the cold memory analysis in Section 2.2, we observe a wide range of cold memory coverage across different machines, even within the same cluster. This demonstrates the advantage of zswap's flexibility in terms of far memory capacity.

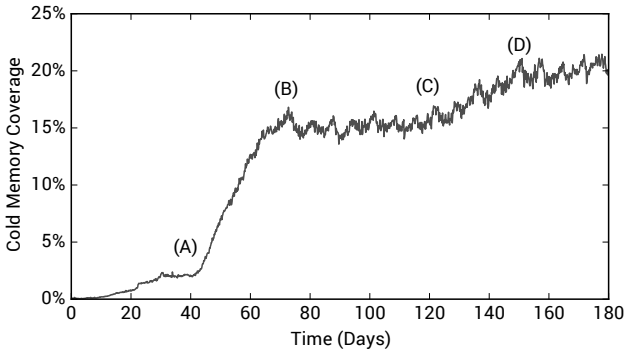


Figure 5. Cold memory coverage over time. zswap with hand-tuned parameters was rolled out during (A) to (B); the autotuner was rolled out during (C) to (D).

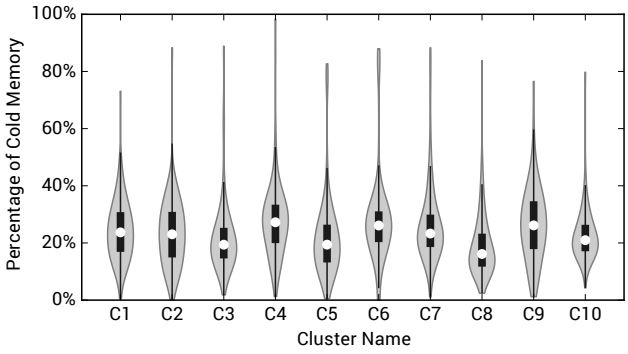


Figure 6. The distribution of the cold memory coverage across the machines in the top 10 largest clusters.

While cold memory coverage varies over machine and time, the cluster-level ratio has been stable, which allows us to convert zswap's cold memory coverage into lower memory provisioning. With 20% of cold memory coverage, 32% for the upper bound for cold memory ratio (Figure 1), and 67% cost reduction for compressed pages (Section 6.3), our system achieves a 4–5% reduction in DRAM TCO in a transparent manner. These savings are realized with *no difference in performance SLIs*, which we discuss next.

6.2 Performance Impact

We measure the performance impact of our system with two metrics: the promotion rate and the CPU overhead. The former is our performance SLI for far memory (Section 4), which can be generalized to other types of far memory devices. The CPU overhead shows the the cycles consumed using zswap as far memory. Additionally, we also monitored other application-level performance metrics and did not detect any statistically meaningful difference before and after the deployment of our system.

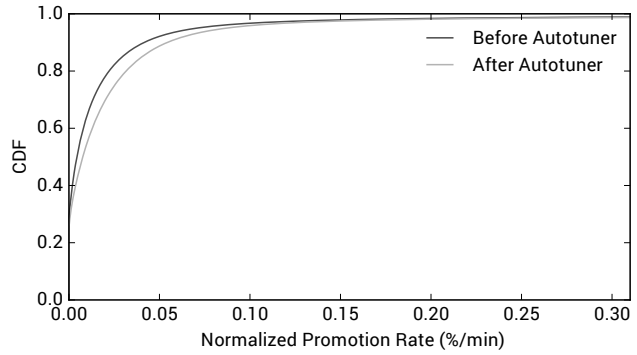


Figure 7. The fleet-wide distribution of the normalized promotion rate before/after applying the ML autotuner.

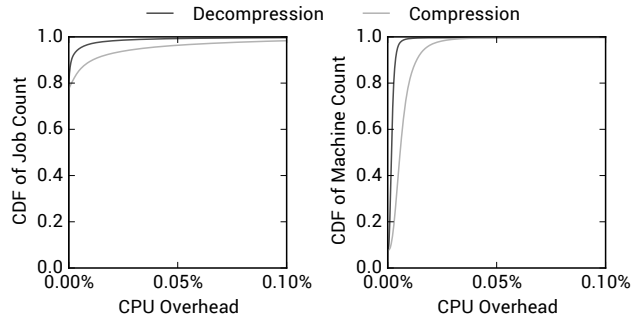
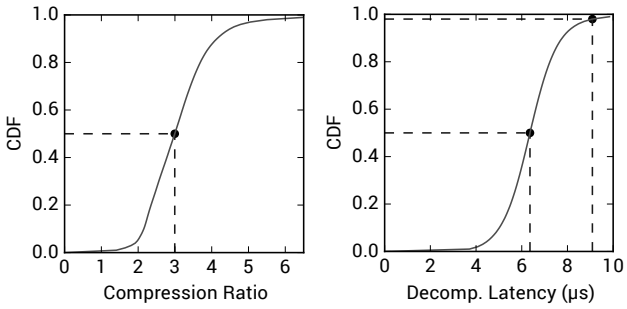


Figure 8. The cumulative distribution of per-job (left) and per-machine (right) CPU overhead as the percentage of CPU cycles spent on compression and decompression.

Figure 7 shows the distribution of the promotion rate of each job normalized to its working set size, before and after deploying ML-based autotuner. Prior to deploying autotuner, we manually determined the values of K and S with months-long A/B experiments of a few candidate configurations from our educated guess. We observe very low promotion rates in both the cases; the 98th percentile of the promotion rate is less than 0.2% of the working set size per minute. This demonstrates that our cold page identification mechanism accurately classifies infrequently accessed pages to be effective candidates for far memory.

Also, Figure 7 shows that the autotuner does not violate the performance SLO, which is defined as the promotion rate at tail. Moreover, the autotuner slightly increased the promotion rate around 25th to 90th percentiles, that is, found a configuration that pushes harder only when the performance SLO has enough margin to increase cold memory coverage.

Figure 8 shows the distribution of the per-job CPU overhead of our system, which is defined as the CPU cycles spent on compressing and decompressing pages, normalized to the CPU usage. These cycles also include those spent on unsuitable incompressible cold pages. For 98% of the jobs, 0.01% and 0.09% of job's CPU usage is spent on compressing cold pages and decompressing them on demand, respectively.



(a) Compression ratio. **(b)** Average decompression latency.

Figure 9. Fleet-wide compression characteristics.

Figure 8 also indicates that the CPU overhead of our system at the machine level is very low as well, with 0.001% and 0.005% for decompression and compression, respectively, at median. From the TCO perspective, this CPU overhead is negligible compared to the average cold memory coverage of 20% across the WSC (Figure 5).

6.3 Compression Characteristics

zswap incurs two types of additional cost to the system. First, compressed pages are still stored in DRAM, which makes the actual memory savings depend on the compression ratio of data. Second, compressed pages are decompressed on demand, which incurs a performance overhead when accessing compressed pages. This subsection quantifies these two aspects based on the statistics collected from the entire WSC.

Figure 9a presents the distribution of average compression ratio of compressed pages in each job, excluding incompressible pages, which are 31% of cold memory on average. Even though zswap uses a lightweight compression algorithm to minimize its CPU overheads, it achieves a 3x compression ratio (i.e., 67% memory savings) at median. The compression ratio varies from 2–6x, which depends on application characteristics. For example, multimedia data and encrypted end-user content are incompressible even when cold.

Figure 9b shows the distribution of average decompression latency per page, which is measured by aggregating the total CPU time spent on decompression for each job divided by the number of decompressed pages during a 5-minute interval. The decompression latency of zswap is measured to be 6.4 µs at the 50th percentile and 9.1 µs at the 98th percentile. Achieving such latencies for zswap-based far memory makes it competitive to alternative far memory technologies available today (e.g., tens of µs latency is common for both the fastest SSDs [38] and remote memory [30]).

6.4 Case Study with Bigtable

Lastly, we present a case study to quantify the impact of our far memory system on application-level performance metrics (e.g., instructions per cycle [45]). Our target application is

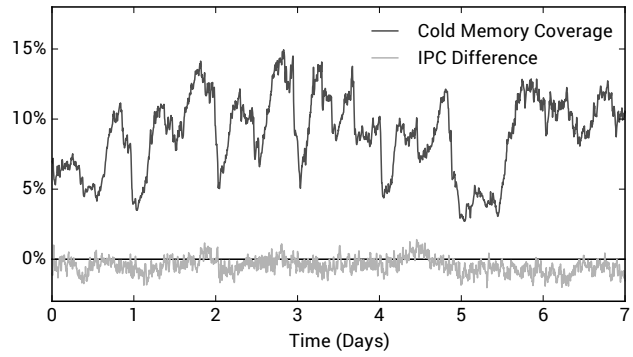


Figure 10. Cold memory coverage and performance overhead of zswap for Bigtable. Negative IPC difference indicates lower performance in machines with zswap enabled.

Bigtable [10], one of our biggest consumers of DRAM, which stores petabytes of data in storage with in-memory caching and serves many production services at a rate of millions of operations per second.

Figure 10 shows results from A/B testing between machines with zswap disabled (control group) and enabled (experimental group). These two groups were constructed by randomly sampling machines in a cluster. We show two metrics: cold memory coverage and user-level IPC. User-level IPC does not count kernel-level instructions, so that instructions executed by zswap can be excluded from IPC calculation and comparison. Also, site reliability engineers [8] monitored other application-level performance metrics (e.g., queries per second) and observed no violations of application-defined SLOs.

Figure 10 shows that the IPC difference between the control and the experimental groups is within noise. Note that, even with cluster-level A/B testing, noisiness is expected in WSCs because of different job instances serving different queries, machine-to-machine variation, etc. We conclude that our system does not incur noticeable degradation in application performance.

Also, zswap achieves 5–15% of the cold memory coverage for Bigtable, which aligns with the fleet-wide average. Even within the same application, we observe ~3x difference in the cold memory coverage over time. Some of this variation comes from the diurnal nature of the load (e.g., lower request rate during nights resulting in more cold memory), but the behavior is not always predictable. This again demonstrates the complexity of provisioning the far memory capacity.

7 Related Work

To the best of our knowledge, we present the first end-to-end system for software-defined far memory deployed at scale. One of the key related ideas on memory compression was proposed by Ekman and Stenstrom [15]. They argue that performing decompression in software is not fruitful since decompression latencies will always be significant compared to the application runtime. However, we show that compression/decompression in software can be done while not impacting application runtime, translating to substantial TCO savings. Additionally, we use machine learning to automatically determine aggressive cold page criteria to increase memory savings while meeting our SLOs.

Memory compression. Memory compression is a well-known technique that has been used for more than 20 years in desktop systems to over-commit memory capacity [16, 33, 43]. Memory capacity generated through compression in desktops can be used when the system is low on memory. Such a reactive technique can cause severe performance degradation in WSCs, which we observed during our initial deployment phase (Section 3.2). On the other hand, in WSCs, memory savings from proactively compressing cold pages translate to cluster-level memory capacity savings, and hence reducing DRAM CapEx (Section 6.1).

Software-managed far memory. Xue et al. [44] proposed an adaptive persistent memory scheme for augmenting memory capacity when machine is under memory pressure. In our work, however, we designed a proactive far memory usage scheme to mindfully avoid OS memory reclaim slow paths during memory allocation requests, which otherwise can hurt tail behavior in WSCs. Agarwal et al. [5] proposed Thermostat, a cold page classification mechanism designed specifically for huge page 2 MB mappings. It introduces extra page faults on a sample of randomly selected cold pages to measure the performance impact of mapping cold page into far memory. In contrast, our technique to estimate performance impact is based on promotion histograms derived from PTE *accessed* bits that covers both huge and regular pages (critical for production systems where fragmentation can limit huge pages). Remote memory is another approach to software-managed far memory [30, 31]. However, remote memory deployment faces many unsolved challenges as discussed by Aguilera et al. [6] that need to be solved before realizing memory TCO savings in practice.

Hardware-based far memory. Several researchers have proposed and studied hardware-based far memory [22, 23, 26, 34]. However, adoption of such proposals at WSC scale is slow and challenging as they often come with the requirement of developing, testing, qualifying, and deploying a new server platform. For example, Intel Optane DC Persistence Memory [20] is compatible only with new Intel CPUs equipped with special memory controllers. Furthermore, the adoption rate of far memory can be limited if it cannot be retrofit to older machines, which often form a large fraction of the production server fleet [7]. Our software-defined far memory solution instead works well with existing server platforms. As hardware-based far memory devices become readily available with a spectrum of latency, bandwidth, and

density characteristics, they can augment memory TCO savings in addition to our zswap-based software approach.

Cold data detection. *Accessed*-bit-based cold/stale page detection technique is a well-known idea [28, 42]. We build our mechanisms on top of such proposals and track the per-page age of all eligible pages at runtime with no storage overhead and tolerable CPU consumption.

Application-driven far memory usage. Eisenman et al. [14] presented an application-driven approach to use far memory in a real datacenter environment. However, with the diversity of applications that run in a WSC, it is not practical to modify each of them individually. This limits wider far memory adoption. Our application-agnostic system design enables us to reap TCO savings without involving any of our customers. Other approaches based on application profiling, user-level APIs for far memory allocation and migration pose similar customer adoption challenges [12, 13, 41].

8 Conclusion

WSC designers today face a significant memory wall that motivates new designs and optimizations: *how do we get increased memory capacity at lower costs without impacting performance and reliability?* In this paper, we discussed one such new design, an approach to creating a *software-defined far memory tier*, that addresses this challenge.

We showed that a proactive approach to identify cold pages can effectively utilize far memory. Our software-defined far memory solution built by compressing cold pages on top of the readily available Linux zswap mechanism can compress memory by 3x or more, effectively creating a new far memory tier that is cheaper by 67% or more than existing memory, with access times in single-digit µs for decompression. Our end-to-end system design uses a new model for identifying cold pages under well-defined performance SLOs and features new mechanisms in the kernel and the node agent to implement such a model at warehouse scale. We also propose a new approach to leveraging machine learning to autotune our system without a human in the loop.

Our system has been in deployment in Google's WSC for several years and our results show that this far memory tier is very effective in saving memory CapEx costs without negatively impacting application performance. But perhaps more importantly, it achieves these benefits transparently to the application developers and can be deployed both on existing and new machine generations, both important considerations for real-world production systems. Additionally, since this approach is software-based, it can be tuned dynamically to respond to the churn and diversity in WSC workloads.

Looking ahead, there are several areas of future work. The cold memory coverage discussed in the paper, although substantial at warehouse scale, is still very conservative. Our SLOs were very stringent to avoid impacting application performance and to minimize cycles spent on compression. Going beyond the promotion rate to looking at end-application performance metrics in our control plane presents significant opportunities for additional savings. Similarly, more aggressive ML-based tuning can provide additional benefits. Furthermore, hardware support for compression, such as through a tightly-coupled accelerator, can increase both the number of pages compressed and the compression ratio (through more complex compression algorithms), correspondingly increasing both coverage and cost savings dramatically.

Finally, the interaction between hardware and software far memory is an interesting area. Our software-defined far memory based on compression is surprisingly competitive in performance and price-performance ratio to newer non-volatile storage solutions already available in the market, such as Intel Optane SSD [20] or Samsung Z-SSD [38]. Emerging hardware technologies such as Intel Optane DC Persistent Memory [20] have sub-µs latencies [32], but the price-performance trade-offs of deploying such technologies is yet to be explored. The lower access latencies of these latter technologies have the potential to further relax the definition of cold memory, potentially targeting more infrequently accessed memory, but with better performance trade-offs. Ultimately, an exciting end state would be one where the system uses both hardware and software approaches and multiple tiers of far memory (sub-µs tier-1 and single-µs tier-2), all managed intelligently with machine learning and working harmoniously to address the DRAM scaling challenge.

Acknowledgments

The results in this paper build on the work of a great larger team at Google and we are grateful to all our colleagues involved in the design and operation of the systems we discuss in this paper. In particular, we would like to thank Hugh Dickins, David Rientjes, Richard Otap, and Rohit Jnagal for their input. We would also like to thank Liqun Cheng, Urs Hölzle, Konstantinos Menychtas, and Amin Vahdat and the anonymous reviewers, as well as our shepherd Marcos K. Aguilera, for their feedback on the paper.

References

[1] [n. d.]. Linux zswap. Retrieved July 31, 2018 from https://www.kernel.org/doc/Documentation/vm/zswap.txt
[2] [n. d.]. Memory Resource Controller. Retrieved July 31, 2018 from https://www.kernel.org/doc/Documentation/cgroup-v1/memory.txt
[3] [n. d.]. Page Frame Reclamation. Retrieved July 31, 2018 from https://www.kernel.org/doc/gorman/html/understand/understand013.html
[4] Advanced Micro Devices Inc. 2018. AMD64 Architecture Programmer's Manual Volume 2: System Programming. Retrieved July 30, 2018 from https://support.amd.com/TechDocs/24593.pdf

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

[6] 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 Symposium on Cloud Computing*.

[7] Luiz André Barroso, Urs Hölzle, and Parthasarathy Ranganathan. 2018. *The Datacenter as a Computer: Designing Warehouse-Scale Machines*. Morgan & Claypool Publishers.

[8] Betsy Beyer, Chris Jones, Jennifer Petoff, and Niall Richard Murphy. 2016. *Site Reliability Engineering: How Google Runs Production Systems*. O'Reilly Media.

[9] Craig Chambers, Ashish Raniwala, Frances Perry, Stephen Adams, Robert R. Henry, Robert Bradshaw, and Nathan Weizenbaum. 2010. FlumeJava: Easy, efficient data-parallel pipelines. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*.

[10] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson Hsieh, Deborah Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert Gruber. 2006. Bigtable: A distributed storage system for structured data. In *Proceedings of the Symposium on Operating Systems Design and Implementation*.

[11] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified data processing on large clusters. In *Proceedings of the Symposium on Operating System Design and Implementation*.

[12] Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. 2014. System software for persistent memory. In *Proceedings of the European Conference on Computer Systems*.

[13] 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 European Conference on Computer Systems*.

[14] Assaf Eisenman, Darryl Gardner, Islam AbdelRahman, Jens Axboe, Siying Dong, Kim Hazelwood, Chris Petersen, Asaf Cidon, and Sachin Katti. 2018. Reducing DRAM footprint with NVM in Facebook. In *Proceedings of the European Conference on Computer Systems*.

[15] Magnus Ekman and Per Stenstrom. 2004. A case for multi-level main memory. In *Proceedings of the Workshop on Memory Performance Issues*.

[16] Adam Engst. 1996. RAM Doubler 2. Retrieved October 17, 2018 from https://tidbits.com/1996/10/28/ram-doubler-2/

[17] Daniel Golovin, Benjamin Solnik, Subhodeep Moitra, Greg Kochanski, John Elliot Karro, and D. Sculley. 2017. Google Vizier: A service for black-box optimization. In *Proceedings of the International Conference on Knowledge Discovery and Data Mining*.

[18] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang Shin. 2017. Efficient memory disaggregation with Infiniswap. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation*.

[19] Intel Corporation. 2016. Intel® 64 and IA-32 Architectures Software Developer's Manual. Retrieved July 30, 2018 from https://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-manual-325462.html

[20] Intel Corporation. 2018. Intel Newsroom. Reimagining the Data Center Memory and Storage Hierarchy. Retrieved July 30, 2018 from https://newsroom.intel.com/editorials/re-architecting-data-center-memory-storage-hierarchy/

[21] Hugo Larochelle Jasper Snoek and Ryan P Adams. 2012. Practical Bayesian optimization of machine learning algorithms. In *Advances in Neural Information Processing Systems*.

[22] Youngbin Jin, Shihab Mustafa, and Myoungsoo Jung. 2014. Area, power, and latency considerations of STT-MRAM to substitute for main memory. In *Proceedings of the Memory Forum*.

[23] Ju-Yong Jung and Sangyeun Cho. 2013. Memorage: Emerging persistent RAM based malleable main memory and storage architecture. In *Proceedings of the International Conference on Supercomputing*.

[24] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. 2015. Profiling a Warehouse-scale Computer. In *Proceedings of the International Symposium on Computer Architecture*.

[25] Uksong Kang, Hak-Soo Yu, Churoo Park, Hongzhong Zheng, John Halbert, Kuljit Bains, S. Jang, and Joo Sun Choi. 2014. Co-architecting controllers and DRAM to enhance DRAM process scaling. *Presented at the Memory Forum*.

[26] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. 2009. Architecting phase-change memory as a scalable DRAM alternative. In *Proceedings of the International Symposium on Computer Architecture*.

[27] Seok-Hee Lee. 2016. Technology scaling challenges and opportunities of memory devices. In *Proceedings of the International Electron Devices Meeting*.

[28] Michel Lespinasse. 2011. Idle page tracking / working set estimation. Retrieved July 31, 2018 from https://lwn.net/Articles/460762/

[29] 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 *Proceedings of the International Conference on Cluster Computing*.

[30] 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. In *Proceedings of the International Symposium on Computer Architecture*.

[31] Kevin Lim, Yoshio Turner, Jose Renato Santos, Alvin AuYoung, Jichuan Chang, Parthasarathy Ranganathan, and Thomas F. Wenisch. 2012. System-level implications of disaggregated memory. In *Proceedings of the International Symposium on High-Performance Computer Architecture*.

[32] Allyn Malventano. 2018. Intel's Optane DC Persistent Memory DIMMs Push Latency Closer to DRAM. Retrieved December 15, 2018 from https://www.pcper.com/news/Storage/Intels-Optane-DC-Persistent-Memory-DIMMs-Push-Latency-Closer-DRAM

[33] Tom Nelson. 2018. Understanding Compressed Memory on the Mac. Retrieved October 17, 2018 from https://www.lifewire.com/understanding-compressed-memory-os-x-2260327

[34] 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 International Symposium on Computer Architecture*.

[35] Parthasarathy Ranganathan. 2017. More Moore: Thinking outside the (server) box. *Keynote at the International Symposium on Computer Architecture*.

[36] Charles Reiss, Alexey Tumanov, Gregory R. Ganger, Randy H. Katz, and Michael A. Kozuch. 2012. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proceedings of the ACM Symposium on Cloud Computing*.

[37] Arthur Sainio. 2016. NVDIMM – Changes are here so what's next? *Presented at the In-Memory Computing Summit*.

[38] Samsung Electronics. 2017. Ultra-Low Latency with Samsung Z-NAND SSD. Retrieved July 31, 2018 from https://www.samsung.com/us/labs/pdfs/collateral/Samsung_Z-NAND_Technology_Brief_v5.pdf

[39] Niranjan Srinivas, Andreas Krause, Sham Kakade, and Matthias Seeger. 2010. Gaussian process optimization in the bandit setting: No regret and experimental design. In *Proceedings of the International Conference on Machine Learning*.

[40] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. 2015. Large-scale cluster management at Google with Borg. In *Proceedings of the European Conference on Computer Systems*.

[41] Haris Volos, Andres Jaan Tack, and Michael M. Swift. 2011. Mnemosyne: Lightweight persistent memory. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*.

[42] Carl A. Waldspurger. 2002. Memory resource management in VMware ESX server. In *Proceedings of the Symposium on Operating Systems Design and Implementation*.

[43] Paul R. Wilson, Scott F. Kaplan, and Yannis Smaragdakis. 1999. The case for compressed caching in virtual memory systems. In *Proceedings of the USENIX Annual Technical Conference*.

[44] Dongliang Xue, Chao Li, Linpeng Huang, Chentao Wu, and Tianyou Li. 2018. Adaptive memory fusion: Towards transparent, agile integration of persistent memory. In *Proceedings of the International Symposium on High Performance Computer Architecture*.

[45] Xiao Zhang, Eric Tune, Robert Hagmann, Rohit Jnagal, Vrigo Gokhale, and John Wilkes. 2013. CPI^2: CPU performance isolation for shared compute clusters. In *Proceedings of the European Conference on Computer Systems*.

[46] Pin Zhou, Vivek Pandey, Jagadeesan Sundaresan, Anand Raghuraman, Yuanyuan Zhou, and Sanjeev Kumar. 2004. Dynamic tracking of page miss ratio curve for memory management. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*.