DAOS: Data Access-aware Operating System

SeongJae Park
Amazon
sjpark@amazon.com

Madhuparna Bhowmik
University of Illinois at Urbana-Champaign
bhowmik6@illinois.edu

Alexandru Uta
Amazon
alexuta@amazon.com

ABSTRACT

In data-intensive workloads, data placement and memory management are inherently difficult: the programmer and the operating system have to choose between (combinations of) DRAM and storage, replacement policies, as well as paging sizes. Efficient memory management is based on fine-grained data access patterns driving placement decisions. Current solutions in this space cannot be applied to general workloads and production systems due to either unrealistic assumptions or prohibitive monitoring overheads.

To overcome these issues, we introduce DAOS, an open-source system for general data access-aware memory management. DAOS provides a data access monitoring framework that utilizes practical best-effort trade-offs between overhead and accuracy. The memory management engine of DAOS allows users to implement their access-aware management with no code, just simple configuration schemes. For system administrators, DAOS provides a runtime system that auto-tunes the schemes for user-defined objectives in a finite time. We evaluated DAOS on commercial service production systems as well as state-of-the-art benchmarks. DAOS achieves up to 12% performance improvement and 91% memory saving. DAOS is upstreamed and available in the Linux kernel.

CCS CONCEPTS

• **Software and its engineering** → **Secondary storage**; **Allocation / deallocation strategies**; • **Information systems** → **Information lifecycle management**.

KEYWORDS

memory management, operating systems

ACM Reference Format:
SeongJae Park, Madhuparna Bhowmik, and Alexandru Uta. 2022. DAOS: Data Access-aware Operating System. In *Proceedings of the 31st Int'l Symposium on High-Performance Parallel and Distributed Computing (HPDC '22), June 27-July 1, 2022, Minneapolis, MN, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3502181.3531466

1 INTRODUCTION

Motivation. In data-intensive workloads, applications operate on larger-than-memory datasets. This means data is frequently brought from storage to memory and vice-versa through mechanisms such as swapping. Deciding what replacement policy [19, 23, 27, 35] to apply adds significant complexity to memory managers. To ensure good application performance, modern memory managers must decide quickly which data should be kept in memory and which data can be kept on slower media [32]. Moreover, allocation granularity (e.g., page sizes) plays an important role in application performance [53, 68]. Application working sets change dynamically during runtime. Moreover, modern applications' working sets are continuously increasing [22] while DRAM capacity in a single machine is not following the same growth [50]. This implies a high probability of frequent thrashing, which can result in significant application performance degradation.

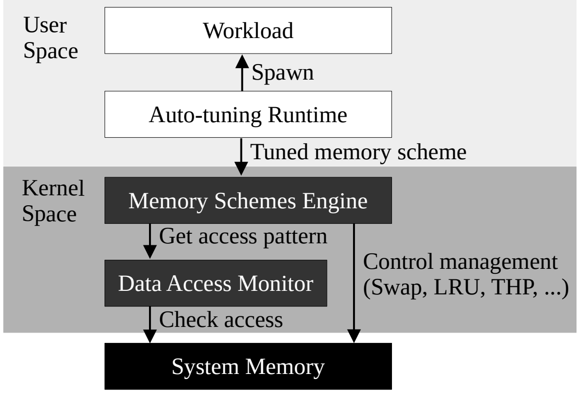

Figure 1: The overall architecture of DAOS.

Good memory managers are data-aware—driven by decisions based on fine-grained memory access patterns. We present the design, implementation and evaluation of DAOS, an open-source data-aware memory management layer which allows users to specify their own memory management policies through simple scripting. Moreover, DAOS also contains an auto-tuning component that automatically optimizes memory management policies, improving memory utilization and limiting performance loss.

State-of-the-art Limitations. Modern servers nowadays include new storage devices (including software-defined ones) that are only about one order of magnitude slower than DRAM [4, 7, 8, 33, 52]. Bridging the gap between memory and slow secondary storage, applications can take advantage of these devices to improve performance. Operating systems can utilize tiered memory constructed using DRAM and the fast storage devices. Such tiered memory should be managed as efficiently as possible, placing data based on application access patterns. Current research in this space falls short in being applicable to general systems and applications.

To offer applications good performance, memory managers need to make use of fine-grained data access information for optimizing placement. Research [13, 44, 46, 56, 63] in this area focuses on utilizing fine-grained data access information for such optimizing

HPDC '22, June 27-July 1, 2022, Minneapolis, MN, USA
© 2022 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9199-3/22/06.
https://doi.org/10.1145/3502181.3531466

memory management, and have shown impressive performance improvements and memory savings. Extraction of the access patterns is a key problem in this domain. This is usually solved using environmental assumptions such as source code availability, dedicated CPU, or limits on memory size. As a result, data access monitoring techniques are not applicable in general systems, or incur prohibitive overheads. Worse yet, because designs are usually tightly-coupled with environment assumptions, it is difficult to apply them on new systems or combine with other techniques if the assumptions conflict. As a result, most prior work failed to get adopted on general systems.

Contributions. To cover these gaps, in this paper we introduce DAOS, a novel access pattern-aware memory management system depicted in Figure 1. DAOS allows users to implement memory management schemes, which are lightweight, effective, and tuned for their given environments. DAOS alleviates users from the difficult and error-prone kernel programming. In detail, our contributions are:

1. Data Access Monitor. The core of DAOS is its data access monitoring framework. Our design minimizes and provides an upper-limit guarantee for its overhead, regardless of the size of the memory being monitored. To achieve this, the monitor trades accuracy for low-overhead, leveraging a best effort higher accuracy.

2. Data Access-aware Memory Management Schemes Engine. Using the *monitor*, we design and implement a memory management engine. The engine allows users to describe how memory should be managed for specific data access patterns in a simplified format called Memory Management Scheme. The engine repeatedly reads the current access pattern of the system via the *monitor* and manages the memory as the given scheme describes. This considerably lowers the barrier for this kind of system optimization which normally requires complex kernel programming.

3. Auto-tuning Runtime. The last DAOS component is a user-space runtime system. In addition to executing a given workload, it automatically tunes and applies a memory management scheme for the currently running system and the workload, in a user-defined time limit. For more customized tuning, it allows users to optionally describe their service level indicator model.

Evaluation using Macro-benchmarks and Production Systems. To showcase the benefits of DAOS, we benchmark it using production systems and 24 realistic workloads from the Parsec3 and Splash-2x benchmark suites. We profile data access patterns of workloads with the *access monitor* while measuring the monitoring overhead. Our results confirm that the *monitor* incurs only modest overhead and offers useful insights for understanding the DRAM-level memory behaviors. We implement the core ideas of two state-of-the-art approaches for access-aware Transparent Huge Pages (THP) [40] and page reclamation [41] as two *memory management schemes*. Using DAOS, these approaches are respectively written in only 2 lines and 1 line in our schemes engine. We apply these for 24 workloads. The *THP scheme* removes 80% of THP memory bloat while preserving 46% of THP performance gains in its best case. The *reclamation scheme* reduces the memory footprint by 91% while incurring only 0.9% runtime slowdown, in its best case. We tune the *reclamation scheme* using the *auto-tuning runtime*. Auto-tuned versions outperform manually tuned versions by up to 20%.

Limitations. At the moment, DAOS does not treat memory reads and writes differently. This might have important implications for devices in which the two operations' performance is not symmetric, e.g., NVM. We leave this feature for future versions of DAOS.

Availability. The source code for DAOS is available freely on Github[1]. The kernel parts of DAOS have already been upstreamed and integrated into the mainline Linux Kernel.

2 BACKGROUND AND RELATED WORK

We discuss necessary background knowledge that DAOS builds upon and present systems related to DAOS explaining what their downsides are.

2.1 Novel Memory Layers

Modern workloads such as big data, cloud, and machine learning accelerated the increase of working set sizes [22] and data intensiveness. Due to physical scaling limits such as power consumption and heat dissipation, the size of DRAM in a single machine does not follow the same growth curve. A recent study on the trend of CPU-DRAM resource ratio on VM instances and physical machines [50] revealed DRAM capacity is relatively decreasing. This implies possible system performance degradation due to thrashing. One straightforward solution is adding layers in the memory hierarchy between DRAM and traditional storage. Both new hardware devices and software-defined devices, such as non-volatile memories [3, 4], disaggregated memory [9, 24, 50], and in-memory compressed block devices [7, 41] can be good examples of such new memory layers. On Linux, the new layer can be configured as a fast swap device [7, 8, 39], a special file system [65], or just a NUMA node [30]. Nevertheless, because the new layer will be slower than DRAM [33], optimal memory management is necessary.

Downside-1: Even though mechanisms for adding extra memory layers exist, policies to control them efficiently and in a fine-grained manner do not exist yet.

2.2 Data Access Monitoring

For effective memory management fine-grained data access pattern information is necessary. One straightforward way for data access pattern extraction is tracing each memory access [34, 37, 45, 54]. This approach usually incurs too high overhead to apply online, because it adds the instrumentation overhead to every memory access. Wang et al. [62] reduce the overhead using a static analysis technique. However, this technique requires access to source code, which is not always available in production.

Time-based sampling is periodically scanning if each memory page was accessed [31, 40, 41, 57] via scanning accessed bits in page tables. This does not require source code or binary modifications and incurs much less overhead compared to tracing. The monitoring overhead of this approach can arbitrarily increase when the number of pages to check grows. The Linux kernel has several memory management mechanisms [6, 23] which also use this approach. However, it does the scanning as rarely as possible to reduce the monitoring overhead. Because this makes only poor quality

[1] https://github.com/damonitor

access information available, Linux utilizes additional heuristics and optimizations such as working set protection [64].

Space-based sampling is an alternative. It refers to increasing the granularity of the monitoring space unit [31, 61] by splitting the monitored regions into small sub-regions that are larger than a single page. Monitoring only one page in each region can mitigate the unbounded overhead problem. This makes the monitoring overhead controllable by setting the number of the sub-regions. However, this can result in poor monitoring accuracy if the access pattern is dynamic or skewed [14, 66]. Adaptively adjusting monitoring regions with observed access frequency [55] is a recent approach that mitigates the problem.

Recency and frequency are essential high-level information for effective memory management [11, 43, 47]. None of the above-mentioned approaches directly provide those, therefore additional processing is required. For example, the Linux kernel transforms the periodic access check results to recency information using its two LRU lists mechanism [23].

Downside-2: Existing access-pattern monitoring techniques are insufficient for modern systems. They have impractical assumptions, poor accuracy, or prohibitive overheads.

2.3 Access-aware Optimization

As heterogeneous memory systems evolved, several schemes for access pattern-aware memory management emerged. These approaches commonly profile access patterns and prioritize data items according to usage. Higher priority items are placed on a different memory device or targeted by a special optimization such as transparent huge pages (THP). Several of these approaches depend on offline profiling and do static placement [20, 48, 59], thus cannot properly react to dynamic access pattern changes. Program execution context-based access profiling for the dynamic access pattern changes have been used for SSD write grouping [28, 38] and data item placement on heterogeneous memory systems [54]. For data item prioritization, recency only (LRU) [19, 23], frequency only (LFU) [27, 35], and adaptive approaches [11, 31, 43, 47, 51] have been explored.

Lagar-Cavilla et al. [41] configure heterogeneous memory systems with compressed in-memory swap devices [8], and proactively reclaim cold pages to improve memory efficiency while keeping the performance stable. This work is successfully deployed on Google's production servers, but was not accepted in the mainline Linux kernel. The community pointed out the unbounded monitoring overhead as a significant problem [18] for general systems. Early OOM [60] is a user-space approach for proactive reclamation. This scheme merely kills processes having the largest memory footprint, without access pattern awareness.

Kwon et al. [40] find that the Linux' transparent huge pages (THP) [6] subsystem aggressively promotes 4 KiB normal pages to 2 MiB huge pages. This behavior results in latency spikes due to time-consuming huge page allocations, and memory bloat due to internal fragmentation.The two problems are solved while preserving most of THP performance gains by limiting the promotions to only accessed pages, and similarly demoting the huge pages. A few more data access-aware THP optimizations [53, 68] also followed.

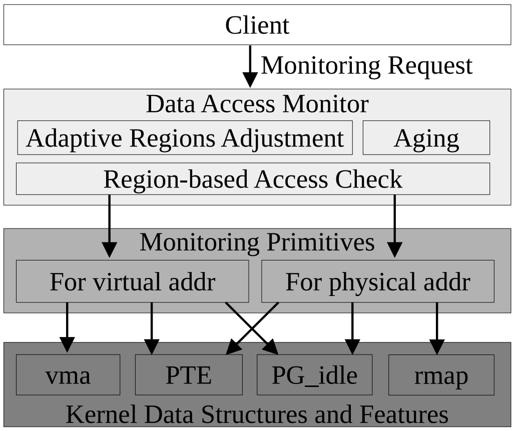

Figure 2: The structure of Data Access Monitor.

The Linux kernel fixed the latency spikes by rejecting the huge page promotion when the huge page allocation failed in the fast path [10]. Recently, proactive compaction [16, 26] is introduced to further reduce the huge page allocation failures. Nevertheless, the memory bloat is still not solved in Linux because current solutions are not access-aware. The access monitoring overhead is one of the blockers.

Downside-3: Applying one or more of the existing access-aware optimizations on general systems is challenging. Implementations are usually coupled with their target system assumptions that frequently conflict with general production systems environments.

3 DAOS DESIGN AND IMPLEMENTATION

To mitigate the downsides illustrated in Section 2, we introduce a novel system called DAOS. The overall architecture of DAOS is illustrated in Figure 1. The system provides three main subsystems, namely the *Auto-tuning Runtime*, the *Memory Schemes Engine*, and the *Data Access Monitor*.

The workflow of the system is as follows: (1) The users pass workloads to the Auto-tuning Runtime; (2) The auto-tuning runtime starts the workload and generates a data access-aware memory management scheme that is tuned for the running system and the workload; (3) The scheme describes what memory management action should be triggered for specific access patterns. (4) The tuned scheme is passed to Memory Schemes Engine in the kernel space; (5) The *engine* continuously monitors the system's access pattern online via the underlying Data Access Monitor. (6) The Monitor receives monitoring requests from Memory Schemes Engine, monitors accesses to the system memory, and returns the monitored access pattern. For each monitoring result that is returned from Data Access Monitor, the *engine* checks if the scheme it has received from *runtime* has an associated memory management action for the current access pattern. If so, it executes the management action.

3.1 Data Access Monitoring

The Data Access Monitor of DAOS is constructed in multiple layers, as depicted in Figure 2. The Data Access Monitor mitigates the monitoring overhead problem indentified in Section 2.2 by utilizing the region-based sampling and the adaptive regions adjustment mechanisms, which are implemented in the Data Access Monitor layer.

Data Access Monitor Layer. At this layer, the monitor divides the target memory region into multiple sub-regions. The sub-regions are composed of adjacent pages that are expected to have similar access frequencies. Because the monitoring target region could be dynamically updated due to certain events such as `mmap()` or memory-hotplug [29], the monitor periodically checks if such changes have been made, and updates the sub-regions accordingly. The time interval for this check is called the *regions update interval.*

The monitor then checks if each sub-region was accessed since the last check after a time interval called *sampling interval.* For this, only one randomly selected sample page in the region is checked (region-based sampling). The check result is aggregated in each region's access counter, which is reset for each time interval called *aggregation interval.* Therefore, the monitoring overhead is controllable by setting the number of sub-regions.

We know that if pages in a sub-region do not have similar access frequencies, the monitoring quality is poor. To avoid this, the monitor splits each sub-region into randomly sized smaller regions and merge adjacent regions having similar access count values after each aggregation interval (adaptive regions adjustment). To keep both the upper limit of the overhead and the lower limit of the accuracy, it avoids splits and merges respectively, if the new number of regions violates the user-defined range. In this way, the monitor provides highly accurate monitoring while incurring only lightweight and upper-bound-guaranteed overhead. The monitoring results, hence, provide fine-grained frequency information.

Recency information is also necessary for effective memory management. The Data Access Monitor utilizes an *Aging* mechanism called to efficiently extract the recency information from the raw monitoring results. Each region contains a counter called *age.* During merge and split, the Aging mechanism checks how the access frequency and size of each region have changed since the last check. If the change is small, *age* is increased. Otherwise, the counter is reset. When a region is split, each sub-region inherits the age of the old region. When regions are merged, the new region gets an age which is the size-weighted average of the old regions' ages. Since aging is implemented together with the adaptive regions adjustment, it incurs only minimal overhead.

Monitoring Primitives Layer. The access check method is dependent on the specific monitoring target. For instance, access checks for virtual addresses and physical addresses are different. To support variable targets, Data Access Monitor separates this part into another layer called *Monitoring Primitives.* The primitives can be implemented by using kernel data structures and features such as accessed bits in page table entries (PTE) or special hardware features like Intel CMT [49] or PML [58]. Users are allowed to implement their monitoring targets and configure Data Access Monitor to use those. By default DAOS provides two reference monitoring primitives implementations for virtual address spaces and the physical address space. The implementation for virtual address spaces uses `struct vma` to track the monitoring target regions, and PTE accessed bits to check accesses. The implementation for the physical address space also uses PTE accessed bits for access checking but it uses the mappings from physical address to virtual addresses (`rmap`) instead of `struct vma` for tracking the monitoring target pages.

The monitoring result is passed to the user by a user-registered callback that is invoked for each aggregation interval, just before resetting the access frequency counters. Users can therefore get in the callback the access frequency and recency of each region from the corresponding counters.

3.2 Memory Management Schemes Engine

Users can typically implement their data access-aware memory management optimizations by writing code in the callback, as prior work proposed. This kind of system-level coding is complex and error-prone.

However, we find that most of the prior data access-aware optimizations have common patterns and therefore can be generalized. Prior work commonly apply memory operations such as locking data in DRAM [20, 48, 54, 59], dynamic page migration [23, 31], page reclamation [41, 60], or THP promotion and demotion [40, 53, 68] based on their static or dynamic data access pattern profile results. We argue that the system can independently perform data access monitoring and triggering memory operations based on monitoring results. Hence, users can do data access-aware memory management optimizations without complex kernel programming. We propose text-based scheme descriptions for triggering memory management conditions. These can be implemented by non-kernel programmers.

Based on this idea, we implement a kernel subsystem called the *Memory Management Schemes Engine* on top of the Data Access Monitor. The engine defines the monitoring result-based memory operations trigger conditions as a simple configuration called a *memory management scheme.* A scheme is constructed with 3 conditions (*min/max size of the target region*, *min/max access frequency of the target region*, and *min/max age of the target region*) and a memory operation action such as *paging out* and *THP promotion.* Users fill the seven values for the memory management scheme and pass it to the engine. The engine starts monitoring accesses, finds memory regions fulfilling the conditions based on the monitoring results, and applies the action of the scheme to the regions. Table 1 describes the actions that DAOS supports. We plan to support more actions in the future.

Listing 1 shows example schemes implementing the main ideas of two state-of-the-art data access-aware optimizations. The first scheme in line 4 is implementing a proactive reclamation [41] by asking the engine to find and page out memory regions that are not accessed for two or more minutes. Lines 8 and 12 are implementing a data access-aware THP [40]. Line 8 asks the engine to make memory regions having 2 MiB or larger size and 80% or more access rate for one or more minutes to use 2 MiB hugepages, while the scheme at line 12 asks the engine to make memory regions having 5% or less access rate for one or more minutes to use 4 KiB regular pages.

3.3 Searching Optimal Operation Schemes

If thresholds for target memory conditions of monitoring-based operation schemes are not tuned for the given workload and the

Table 1: The actions supported by the DAOS Scheme Engine.

Action	Description
WILLNEED, COLD	Asks the kernel to expect the given region will be accessed/not accessed soon.
HUGEPAGE, NOHUGEPAGE	Asks the kernel to do THP promotions/demotions for the given region.
PAGEOUT	Immediately page out the memory region.
STAT	Count the total number and size of memory regions fulfilling the conditions. Can be used for estimating working set size and scheme tuning.

system, it could result not only in insufficient benefit but also significant degradations. The schemes engine provides the STAT action for such tuning. The tuning finds six optimal threshold values which depend on the data access pattern and CPU usage of the workload as well as the hardware characteristics of the server. For example, CPU speed and DRAM latency, bandwidth, capacity. This is a complex problem, and therefore tuning schemes could be difficult and time-consuming even for experts.

The problem can be redefined in a simpler form: finding the appropriate aggressiveness of the given memory operation action that achieves the best performance and memory efficiency of the target workload. This makes sense as the thresholds control the aggressiveness of the action. Changing memory management affects the performance and memory efficiency. Users typically have unique preferences and service level agreement (SLA) for performance and efficiency. For such cases, we further unify these two goal metrics to a single score value that is calculated from the two metrics by a user-defined formula.

For example, if a user values the performance and the memory efficiency equally, but has an SLA that promises no more than 10% performance drop, Listing 2 can be a reasonable score function. Memory footprint (resident set size) is represented as rss in the function. If the SLA is not violated (line 3), the function returns an even-weighted sum of performance improvement and memory saving (line 4). Otherwise, the worst score ever calculated is returned (line 7).

A straightforward solution to this simplified problem will be trying every possible aggressiveness and picking one that achieves the best score. This solution will explore a large search space, which is inevitable if the relation between the aggressiveness and the score is random. In practice we find that this is not the case. Empirically we find only a limited number of possible patterns with which we experiment in this paper and we describe below. We encourage practitioners to explore other scoring functions with different behaviors. It is outside the scope of this article to explore other scoring functions.

```
# size       frequency   age        action
# page out memory regions not accessed
# >=2 minutes
min  max     min  min    2m  max    page_out

# Use THP for >=2MiB regions having
# >=80% frequency ratio for >=1 minute
2MB  max     80%  max    1m  max    thp

# Do not use THP for regions having
# <=5% frequency ratio for >=1 minute
min  max     min  5%     1m  max    nothp
```

Listing 1: Example memory management schemes.

We find 6 patterns are expected, as depicted in Figure 3. For example, as we apply PAGEOUT more aggressively, the performance will degrade only gradually at the beginning, then becomes steep after the first inflection point (starts thrashing). The performance will decrease gradually again after the second inflection point (thrashing nearly saturated). The behavior is opposite for memory efficiency. Therefore, the score, which is proportional to the performance and memory efficiency, has one of six patterns (see Figure 3-right): 1) continuously increases (the memory efficiency dominates), 2) first increases and then decreases but still better than no action (dominated by the efficiency at first, then by the performance, but eventually by the efficiency again), and 3) first increases and then decreases, eventually worse than no action. Three more complementary patterns also exist.

3.4 Metric Validation

To empirically show the theory is valid, we measure and calculate the metrics for several workloads. The behavior and patterns vary for different hardware. We set up three different Amazon AWS EC2 bare metal instances. These are tuned for I/O intensive workloads (i3.metal [1]), compute-, memory-, and network-balanced workloads (m5d.metal [12]), and compute-intensive workloads (z1d.metal [2]), described in Table 2. On each server we launch one QEMU/KVM [5] virtual machine running the DAOS-ported v5.10 Linux kernel. On the virtual machines, we run 24 realistic workloads from Parsec3 and Splash-2x benchmark suites [67]. The memory scheme we use asks the engine to page out memory regions that are not accessed for *min_age* or more seconds. We vary the min_age from 0 seconds to 60 seconds in 1-second granularity. For each run, we measure runtime and memory footprint, then calculate the score using the score function in Listing 2. Each measurement is performed three times. Figure 4 shows the average scores and error bars show the standard deviations of the runs.

```
1 pscore = -1 * (runtime / orig_runtime - 1)
2 mscore = -1 * (rss / orig_rss - 1)
3 if pscore > -0.1:
4     score = 0.5 * pscore + 0.5 * mscore
5     prev_scores.append(score)
6     return score
7 return min(prev_scores)
```

Listing 2: Example score function for same preference on performance and memory saving, with SLA promising no more than 10% performance drop. rss is resident set size.

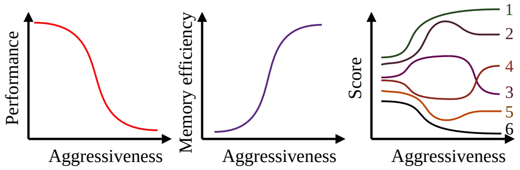

Figure 3: Patterns for performance, memory efficiency, and a unified score for varying aggressiveness of PAGEOUT.

Each workload on each machine shows one of the 6 expected patterns. The patterns are not only dependent on workloads but also on the hardware. For example, scores for parsec3/canneal on i3.metal and z1d.metal show the third pattern but the sixth pattern is shown on m5d.metal. The measurement result also shows random score variations. The scores of several workloads including parsec3/canneal, parsec3/streamcluster, and parsec3/x264 vary too much so that it is hard to recognize the pattern.

Conclusion-1: For the score function defined in Listing 2, the 6 patterns identified in Figure 3 are found in practice for Parsec3 and Splash-2x workloads (Figure 4).

3.5 Auto-tuning Runtime

Based on the theory and observations, we design the autotuning runtime system that finds and applies the best memory management schemes for the given workload and the running system, within a user-defined time limit. The runtime estimates the relationship between the aggressiveness of the memory operation and the score, with a limited number of samples. The runtime adheres to the user-specified time limit by controlling the number of samples for the relation estimation. Finally, the runtime finds the best aggressiveness from the estimated relation.

Inputs. The runtime system asks users to provide the basic memory management scheme to tune, the workload to apply the tuned scheme, in form of a pid or a command, and finally the time limit for the tuning. For more customized tuning, users can optionally ask the system to use specific metrics for *performance* and *memory efficiency*. For the former, runtime is used by default while for the latter the default is RSS. Alternatively, users can define a new score function. The function illustrated in Listing 2 is used by default. For custom metrics, the minimal time to wait for stabilized value (*unit work time*) is also required.

Sampling. The runtime system first calculates the number of available samples (nr_samples) by dividing the total limit time by unit work time. Then, it randomly picks nr_samples combinations of the scheme thresholds having different aggressiveness. Further, the system sequentially applies each of the combinations to the workload, measures metrics, and calculates the score. To get more effective samples, the system first randomly picks only 60% of nr_samples samples to explore the global parameter space and picks the remaining 40% samples near the parameters which have shown the highest scores for a localized search around the best points.

Estimation vs. best scheme searching. To get the relationship while mitigating the random score noise, we use polynomial curve fitting [21, 25, 42]. The degree is set as nr_samples/3 to avoid over-fitting. On the fitted curve, the system finds peaks using gradients [36] and finally applies the configuration of the peak having the highest score.

Table 2: AWS EC2 instance types used in experiments.

Instance type	CPU	DRAM
i3.metal	3.0 GHz x 36 vCPUs	128 GiB
m5d.metal	3.1 GHz x 48 vCPUs	96 GiB
z1d.metal	4.0 GHz x 24 vCPUs	96 GiB

Figure 5 is showing an example of this process with data from parsec3/raytrace assuming only 10 samples are allowed for the tuning process. The *Measured* line shows the second-granularity measurement results that show trend with significant noise. Because only 10 samples are allowed, the system randomly collects 60% of allowed samples (for *min_age* of 5, 17, 23, 36, 47, and 55 seconds). Because the sample of 17 seconds *min_age* shows the highest score, it picks the remaining 40% of allowed samples (*min_age* of 13, 15, 18, 19 seconds) near it. Curve fitting is applied to the 10 samples defining the *Estimated* line. The highest peak at is at 16 seconds *min_age* and therefore the schemes are tune to use this value.

3.6 Implementation

The kernel space subsystems of DAOS, namely the Data Access Monitor and Memory Schemes Engine are implemented in the Linux kernel. The implementations were easily ported to a wide range of kernel versions from v4.4 to v5.10 due to their modular design. For these kernel components, we also provide unit tests and black box tests made on top of the kunit [17] and the kselftest [15] frameworks, respectively.

We use a pseudo-file system of the Linux kernel, namely debugfs as a user-space interface for the Memory Schemes Engine. That is, the Auto-tuning Runtime in user-space passes memory schemes to the engine by writing it to a file on the debugfs. The runtime also uses another Linux kernel pseudo-file system called procfs to read the memory footprint of the target workload.

The Auto-tuning Runtime is implemented in user space as a set of bash and python scripts. We do not make performance optimizations for the runtime internal logic, such as the curve fitting and highest peak searching, because the auto-tuning time is mostly consumed in collecting samples.

4 DAOS IN PRACTICE

We showcase our experience with DAOS on both state-of-the-art benchmarks and a commercial production use-case from a large cloud provider. We track the overhead DAOS adds and the improvements it can bring in terms of both memory usage and performance. Using DAOS we mimick the access-aware optimization proposed in previous research [40, 41].

In our experiments, we use AWS EC2 baremetal instances of different types, as described in Table 2. On each instance, we run a QEMU/KVM [5] guest virtual machine that utilizes half the CPUs and a quarter of the memory.

Access-aware Optimization Policies. To evaluate the accuracy of the Data Access Monitor and the effectiveness of the Memory

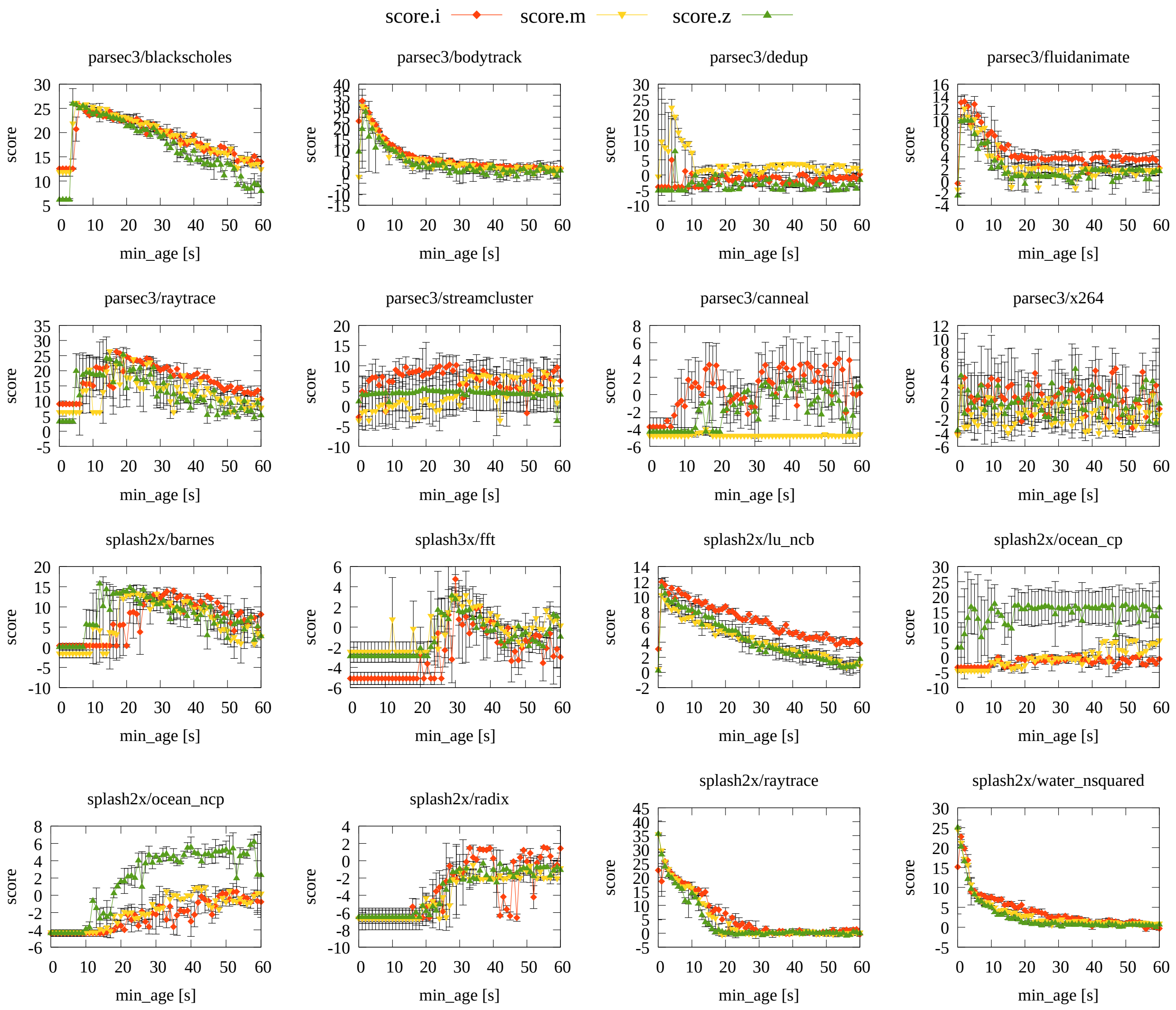

Figure 4: Scores of the proactive reclamation schemes for varying aggressiveness on different workloads and machines. The scores from `i3.metal`, `m5d.metal`, and `z1d.metal` are labeled as *score.i*, *score.m*, and *score.z*. Note that aggressiveness increases from right to left. We ran a total of 24 benchmarks, but plot only 16 due to space constraints.

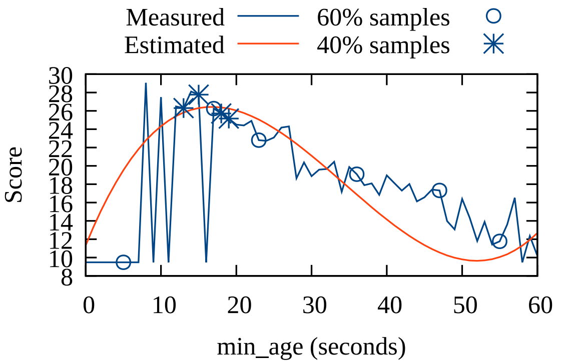

Figure 5: The trend estimation for parsec3/raytrace.

```
1  # size      frequency    age        action
2  min  max    5  max       min  max   hugepage
3  2M   max    min  min     7s   max   nohugepage
4
5  4K   max    min  min     5s   max   pageout
```

Listing 3: Monitoring-based memory management schemes for evaluations. Lines 2 and 3 are implementing *ethp*, while line 5 is for *prcl*.

Schemes Engine, we implement the core ideas of two state-of-the-art data access-aware optimization works for THP [40] and proactive reclamation [41] in two simple memory management schemes called *ethp* and *prcl*, respectively. The two schemes are shown in

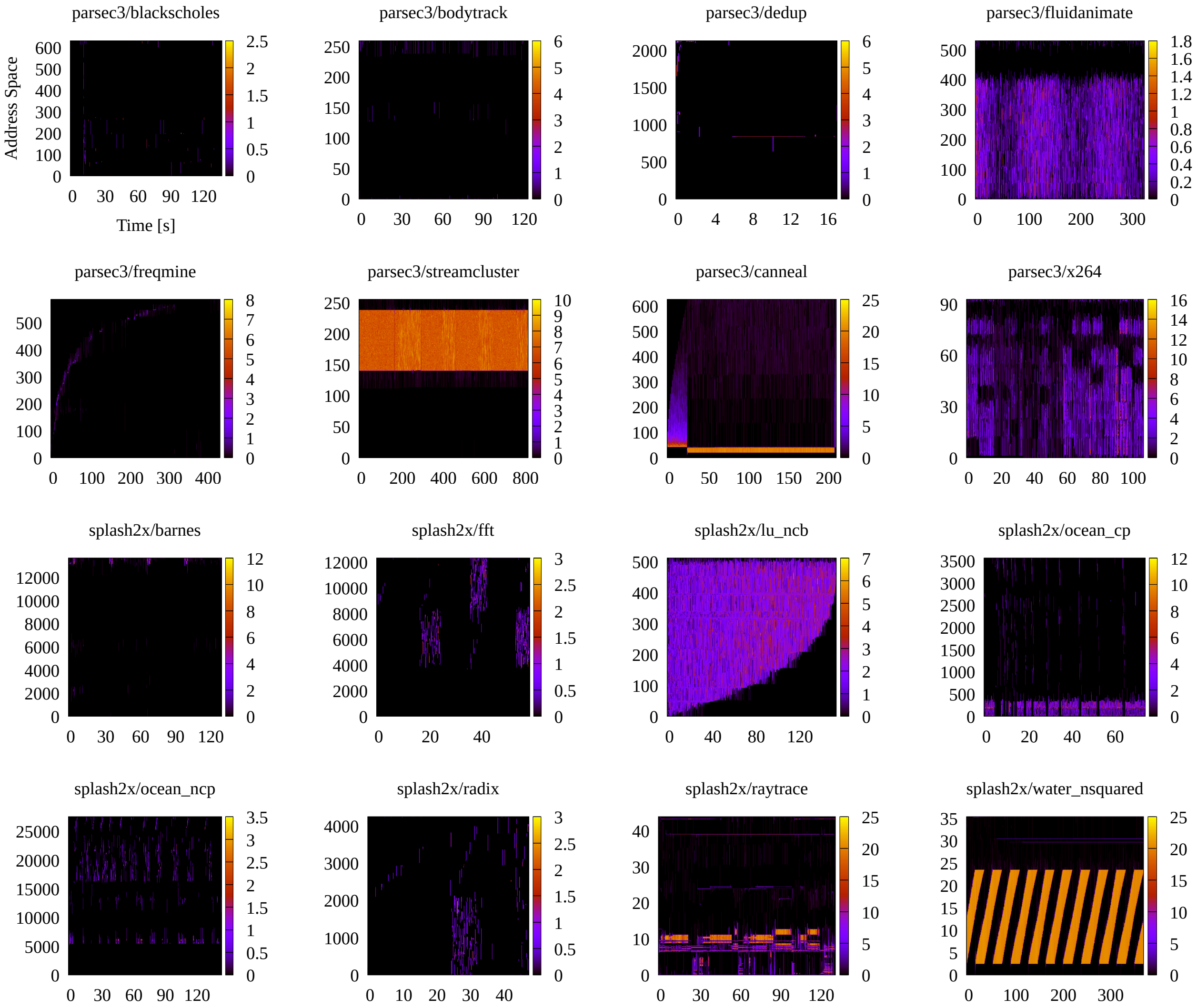

Figure 6: Data access patterns of the workloads in heatmap format. Each heatmap shows when (x-axis, in seconds) what memory regions of specific address (y-axis, in MiB) is how frequently (color) accessed.

Listing 3. Lines 2 and 3 are for *ethp* and line 5 is for *prcl*. The thresholds are manually optimized by our experience on the i3.metal guest.

Workloads. On the virtual machines, we run 24 realistic workloads from state-of-the-art Parsec3 and Splash-2x benchmark suites [67]. Each workload is run against six different system configurations called *baseline*, *rec*, *prec*, *thp*, *ethp*, and *prcl*. **Baseline** runs DAOS-ported v5.10 Linux kernel but disables DAOS features, turns off THP, and utilizes a 4 GiB Zram [7] swap device. Other configurations are the same as the *baseline* but have below differences. **Rec** and **prec** run Data Access Monitor to monitor and record the access patterns in the virtual address space of the workload and the entire physical address space of the guest machine, respectively. **Thp** turns THP on. **Ethp** and **prcl** apply *ethp* and *prcl* memory schemes, respectively.

We set the sampling interval, the aggregate interval, the regions update interval, the minimum number of regions, and the maximum number of regions of Data Access Monitor as 5 milliseconds, 100 milliseconds, 1 second, 10, and 1,000, respectively. Note that the minimum sampling interval of the original proactive reclamation [41] is 24,000 times larger (2 minutes) than ours due to the monitoring overhead problem, which DAOS alleviates.

4.1 DAOS with Manual Control

To show if the Data Access Monitor exhibits good accuracy, we visualize the monitoring results recorded with *rec* configuration in heatmap format in Figure 6. The color denotes access frequency, the x-axis represents time, and the y-axis encodes the region's memory address. In practice, the virtual address space has two large gaps between stack, `mmap()`-ed areas, and heap. Since visualizing the gaps makes the heatmap mostly blank, we find and visualize the biggest subspace of each workload that shows active access patterns.

Each workload shows its unique data access patterns. Small hot memory regions are easily identifiable (canneal, dedup) and dynamic pattern changes are effectively captured (fft, raytrace, and water_nsquared of splash-2x). The patterns are also similar to those reported by prior work [55]. We conclude our Data Access Monitor provides monitoring results that are accurate enough for DRAM-level profiling.

Conclusion-2: DAOS is able to accurately identify the hot memory regions and capture the different access patterns exhibited by varied workloads (Figure 6).

4.2 DAOS Overhead & Benefits

To show how much monitoring overhead DAOS incurs and how much improvement the monitoring-based memory management schemes provide, we measure the performance and memory efficiency of each workload that is normalized to those of *baseline*, for the five configurations. We measure the runtime and resident set size (RSS) of each workload for performance and memory footprint, respectively. We show only the results on i3.metal here, because we manually optimized *ethp* and *prcl* schemes on the machine. The results on the other instance types are similar. The results are plotted in Figure 7.

Monitoring overhead. While running the *rec* and *prec* configurations, the monitoring thread is mostly in a sleep state waiting for the next sampling stage. On average, the thread consumes 1.37% and 1.46% of a single CPU time. The average of the normalized performance of the workloads under *rec* and *prec* are both 0.99. Those in the worst cases are 0.97 and 0.96. This means the monitoring incurs about 1% on average and up to 4% slowdown, which is modest. Note that *prec* monitors much larger target memory (entire memory of the machine) compared to *rec* (only the virtual address space of each workload) but shows similar overhead because the Data Access Monitor overhead control mechanism works regardless of the monitoring target memory size.

Conclusion-3: DAOS overhead is minimal, incurring on average 1% CPU utilization on a core, and at most 4% slowdown (Figure 7).

Effects of *ethp*. *Thp* shows how much performance improvement and memory bloat the Linux-original THP incurs. Because THP benefits depend on the access pattern, not every workload gets clear performance improvement. Splash-2x/ocean_ncp is the best case showing a 27.54% performance gain. However, it also shows the biggest memory efficiency drop of 82.18%. In contrast, the monitoring-based THP scheme of DAOS (*ethp*) provides 12.67% performance improvement and 16.3% memory efficiency drop for the workload. In other words, it reduces 80.16% of memory overhead while preserving 46% of the performance improvement. On average, *thp* makes 5.23% performance improvement and 5.51% memory overhead. *Ethp* preserves 39% of the performance improvement and removes 64.28% of the memory overhead on average.

Effects of *prcl*. Applying the *prcl* results in 37.10% memory saving and 13.66% performance drop on average. In the case of parsec3/freqmine, it achieves 91.34% memory saving with only a 0.91% slowdown. However, the slowdown is significant for several workloads. In the worst case (i.e., splash-2x/ocean_ncp), it incurs a 78.16% slowdown though it also generates a 36.29% memory saving.

Conclusion-4: DAOS is lightweight and accurate. The Memory Schemes Engine is effective in implementing general DRAM-level performance optimizations. The significant performance drop of *prcl* in the worst-case shows tuning the schemes is necessary for each workload (Figure 7).

4.3 Auto-tuning Runtime

Using the Auto-tuning Runtime System, we find the best *prcl* scheme for each workload on the three instance types. We set the maximum number of samples for the relation estimation to 10 and use the score function described in Listing 2. Therefore, we give the same priorities to performance and memory efficiency gains but avoid significant performance drops. Figure 8 shows the results of the auto-tuned versions.

The manually-written scheme incurs 13.65%, 13.45%, and 9.54% performance drops on average. The auto-tuner reduces the performance loss to 0.91%, 2.04%, and 0.59%. **DAOS auto-tuning removes 93.33%, 84.83%, and 93.81% of the performance drops** of the manually-written schemes. In the worst case (parsec3/canneal on z1d.metal), the manual version shows up to 78.18% performance degradation for splash2x/ocean_ncp workload on i3.metal, but the auto-tuned version shows only 14.61% performance drop.

The average memory savings of the auto-tuned versions (24.97%, 24.73%, and 25.10%) are lower than those of the manual version (37.10%, 35.21%, and 33.22%). This is because the Auto-tuning Runtime System uses less aggressive schemes for several workloads that can incur significant performance drops otherwise. Still, several auto-tuned versions also show better memory savings with several workloads including raytrace, volrend and water_nsquared of splash-2x.

In total, the average scores on the three machines are 9.99, 10.09, and 11.35 with the manual version, but the auto-tuner increases the score to 12.08, 11.01, and 12.06. The auto-tuner obtains score improvements of 20.02%, 6.16%, and 6.25%.

Conclusion-5: The DAOS auto-tuner is able to significantly reduce performance degradations compared to manual optimizations at a cost of slightly smaller memory improvements (Figure 8).

4.4 DAOS in Production Systems

We have experienced running DAOS on production systems. For the purpose of this paper we considered a large-scale serverless production system, which runs on baremetal machines. For the experiment, we set up several test machines that closely replicate the settings and workloads of the serverless production system. The production system is composed of several processes running to serve client requests. The measured memory overhead of this service is relatively large, with a difference between resident sets and working sets of approximately 90%.

Therefore, DAOS is a good candidate for trimming down the memory bloat using its page-reclaiming scheme. We hand-crafted a

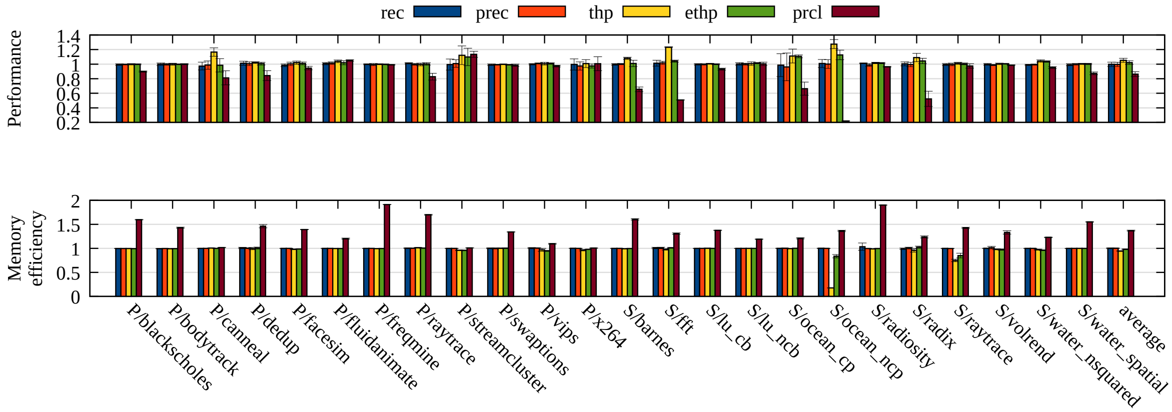

Figure 7: Normalized performance and memory efficiency of workloads for monitoring (*rec* and *prec*), original THP (*thp*), and monitoring-based memory management schemes (*ethp* and *prcl*). The monitoring overhead and memory improvement of the schemes. Prefixes of *P/* and *S/* mean the workload belongs to parsec3 and splash-2x, respectively.

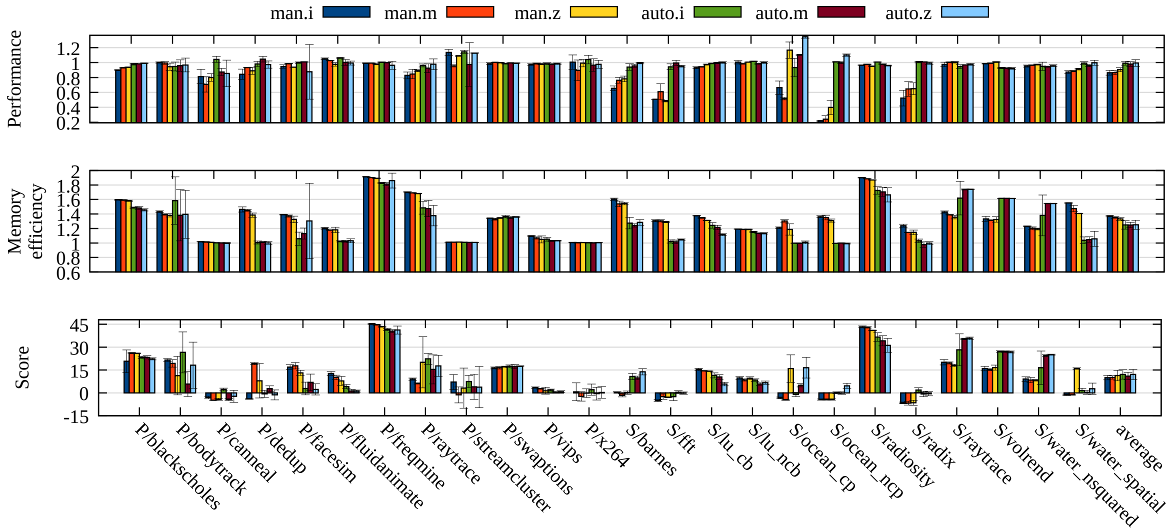

Figure 8: Comparisons of the manually optimized scheme and automatically tuned schemes. Auto-tuning removes 90% of slowdown while preserving 70% of memory savings on average. The results from the manually optimized *prcl* scheme are labeled with *man* prefix while Auto-tuned schemes are labeled with *auto* prefix. The suffixes of *.i*, *.m*, and *.z* represent the types of the running host instances, i3.metal, m5d.metal, and z1d.metal, respectively. Prefixes of *P/* and *S/* mean the workload belongs to parsec3 and splash-2x, respectively.

scheme to page-out to either ZRAM-based swap or file-based swap all the pages that are not touched for 30 seconds. We let DAOS run for several minutes and inspected the amount of RSS reported by the server. The results are plotted in Figure 9. Our experience shows that DAOS is able to reduce significantly the memory overhead – by 80% in the ZRAM case and by 90% in the file-based case. All this advantage comes at a modest cost of at most 2% CPU overhead. The applications running on the production system report negligible performance overhead.

Conclusion-6: DAOS is able to significantly reduce the memory overhead of a serverless production workload while adding modest CPU overhead (Figure 9).

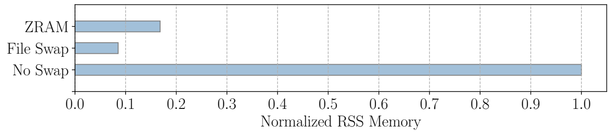

Figure 9: DAOS significantly reduces memory bloat by reclaiming pages.

5 CONCLUSION

We introduced DAOS, a system for general data access-aware memory management optimizations. DAOS provides a highly accurate data access monitoring capability incurring only lightweight and upper-bound-limited overhead. Furthermore, DAOS provides an abstraction for general access-aware memory management so that users can implement their optimizations with no code, but rather simple user-space scripting. Finally, DAOS provides a runtime system for auto-tuning the user-defined access-aware optimizations. In our experience running DAOS on state-of-the-art benchmarks and real-world production systems, we confirmed DAOS is lightweight and achieves performance and memory efficiency improvements of up to 12% and 91%, respectively. DAOS is open-source and already contributed to the Linux kernel.

REFERENCES

[1] Amazon ec2 i3 instances - high i/o compute instances. https://aws.amazon.com/ec2/instance-types/i3/.

[2] Amazon ec2 z1d instances - high frequency compute instances. https://aws.amazon.com/ec2/instance-types/z1d/.

[3] Intel Memory Drive Technology. https://www.intel.com/content/www/us/en/software/intel-memory-drive-technology.html.

[4] Intel Optane Persistent Memory. https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html.

[5] Qemu. https://www.qemu.org/.

[6] Transparent hugepage support. https://www.kernel.org/doc/Documentation/vm/transhuge.txt.

[7] zram: Compressed ram based block devices. https://www.kernel.org/doc/Documentation/blockdev/zram.txt.

[8] zswap. https://www.kernel.org/doc/html/latest/vm/zswap.html.

[9] Marcos K. Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Stanko Novaković, Arun Ramanathan, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, and Michael Wei. Remote regions: a simple abstraction for remote memory. In *2018 USENIX Annual Technical Conference (ATC)*, pages 775–787, Boston, MA, July 2018. USENIX Association.

[10] Vlastimil Babka. [rfc 06/13] mm, thp: remove __gfp_noretry from khugepaged and madvised allocations. https://lkml.org/lkml/2016/5/10/105, 2016.

[11] Sorav Bansal and Dharmendra S Modha. Car: Clock with adaptive replacement. In *3rd USENIX Conference on File and Storage Technologies (FAST)*, volume 4, pages 187–200, 2004.

[12] Jeff Barr. Amazon ec2 m5 instances - general purpose compute instances. https://aws.amazon.com/ec2/instance-types/m5/, 2017.

[13] Wen-ke Chen, Sanjay Bhansali, Trishul Chilimbi, Xiaofeng Gao, and Weihaw Chuang. Profile-guided proactive garbage collection for locality optimization. In *ACM SIGPLAN Notices*, volume 41, pages 332–340. ACM, 2006.

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

[15] Jonathan Corbet. A kernel self-testing update. https://lwn.net/Articles/737893/, 2017.

[16] Jonathan Corbet. Proactive compaction. https://lwn.net/Articles/717656/, 2017.

[17] Jonathan Corbet. A kernel unit-testing framework. https://lwn.net/Articles/780985/, 2019.

[18] Jonathan Corbet. Proactively reclaiming idle memory. https://lwn.net/Articles/787611/, 2019.

[19] Asit Dan and Don Towsley. An approximate analysis of the lru and fifo buffer replacement schemes. *SIGMETRICS Perform. Eval. Rev.*, 18(1):143–152, April 1990.

[20] Subramanya R Dulloor, Amitabha Roy, Zheguang Zhao, Narayanan Sundaram, Nadathur Satish, Rajesh Sankaran, Jeff Jackson, and Karsten Schwan. Data tiering in heterogeneous memory systems. In *Proceedings of the 11th European Conference on Computer Systems (EuroSys)*, page 15. ACM, 2016.

[21] S. A. Dyer and Xin He. Least-squares fitting of data by polynomials. *IEEE Instrumentation Measurement Magazine*, 4(4):46–51, 2001.

[22] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. Clearing the clouds. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems*, volume 47 of *ASPLOS*, page 37, New York, New York, USA, 2012. ACM Press.

[23] Mel Gorman. *Page Frame Reclamation*. Prentice Hall Upper Saddle River, 2004. https://www.kernel.org/doc/gorman/html/understand/understand013.html.

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

[25] Philip George Guest and Philip George Guest. *Numerical methods of curve fitting*. Cambridge University Press, 2012.

[26] Nitin Gupta. Proactive compaction for the kernel. https://lwn.net/Articles/817905/, 2020.

[27] Vishal Gupta, Min Lee, and Karsten Schwan. Heterovisor: Exploiting resource heterogeneity to enhance the elasticity of cloud platforms. In *Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '15, page 79–92, New York, NY, USA, 2015. Association for Computing Machinery.

[28] Keonsoo Ha and Jihong Kim. A program context-aware data separation technique for reducing garbage collection overhead in nand flash memory. *Proc. 7th IEEE SNAPI*, 2011.

[29] Dave Hansen. Memory hotplug. https://lwn.net/Articles/124226/, 2005.

[30] Dave Hansen. Allow persistent memory to be used like normal ram. https://lwn.net/Articles/776921/, 2019.

[31] T. Heo, Y. Wang, W. Cui, J. Huh, and L. Zhang. Adaptive page migration policy with huge pages in tiered memory systems. *IEEE Transactions on Computers*, pages 1–1, 2020.

[32] Călin Iorgulescu, Florin Dinu, Aunn Raza, Wajih Ul Hassan, and Willy Zwaenepoel. Don't cry over spilled records: Memory elasticity of data-parallel applications and its application to cluster scheduling. In *2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17)*, pages 97–109, 2017.

[33] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. Basic performance measurements of the intel optane DC persistent memory module. *CoRR*, abs/1903.05714, 2019.

[34] Aamer Jaleel. Memory characterization of workloads using instrumentation-driven simulation. http://www.jaleels.org/ajaleel/publications/SPECanalysis.pdf, 2007.

[35] George Karakostas and D Serpanos. Practical lfu implementation for web caching. *Technical Report TR-622–00*, 2000.

[36] H Jerome Keisler. *Foundations of infinitesimal calculus*, volume 20. Prindle, Weber & Schmidt Boston, 1976.

[37] Changsu Kim, Juhyun Kim, Juwon Kang, Jae W Lee, and Hanjun Kim. Context-aware memory profiling for speculative parallelism. In *2017 IEEE 24th International Conference on High Performance Computing (HiPC)*, pages 328–337. IEEE, 2017.

[38] Taejin Kim, Duwon Hong, Sangwook Shane Hahn, Myoungjun Chun, Sungjin Lee, Jooyoung Hwang, Jongyoul Lee, and Jihong Kim. Fully automatic stream management for multi-streamed ssds using program contexts. In *17th USENIX Conference on File and Storage Technologies (FAST)*, pages 295–308, 2019.

[39] Sohyang Ko, Seonsoo Jun, Yeonseung Ryu, Ohhoon Kwon, and Kern Koh. A new linux swap system for flash memory storage devices. In *International Conference on Computational Sciences and Its Applications*, pages 151–156. IEEE, 2008.

[40] Youngjin Kwon, Hangchen Yu, Simon Peter, Christopher J Rossbach, and Emmett Witchel. Coordinated and efficient huge page management with ingens. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 705–721, 2016.

[41] 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. Software-defined far memory in warehouse-scale computers. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, pages 317–330, New York, NY, USA, 2019. ACM.

[42] EC Levy. Complex-curve fitting. *IRE transactions on automatic control*, (1):37–43, 1959.

[43] Zhan-sheng Li, Da-wei Liu, and Hui-juan Bi. Crfp: a novel adaptive replacement policy combined the lru and lfu policies. In *2008 IEEE 8th International Conference on Computer and Information Technology Workshops*, pages 72–79. IEEE, 2008.

[44] Youyou Lu, Jiwu Shu, Weimin Zheng, et al. Extending the lifetime of flash-based storage through reducing write amplification from file systems. In *11th USENIX Conference on File and Storage Technologies (FAST)*, volume 13, 2013.

[45] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Acm sigplan notices*, volume 40, pages 190–200. ACM, 2005.

[46] Jaydeep Marathe and Frank Mueller. Hardware profile-guided automatic page placement for ccNUMA systems. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 90–99. ACM, 2006.

[47] Nimrod Megiddo and Dharmendra S Modha. Arc: A self-tuning, low overhead replacement cache. In *2nd USENIX Conference on File and Storage Technologies (FAST)*, volume 3, pages 115–130, 2003.

[48] Gaku Nakagawa and Shuichi Oikawa. Data placement based on data semantics for nvdimm/dram hybrid memory architecture. *CLOUD COMPUTING 2017*, page 109, 2017.

[49] Khang T Nguyen. Intel's cache monitoring technology: Use models and data. https://software.intel.com/content/www/us/en/develop/blogs/intels-cache-monitoring-technology-use-models-and-data.html, 2016.

[50] Vlad Nitu, Boris Teabe, Alain Tchana, Canturk Isci, and Daniel Hagimont. Welcome to zombieland: practical and energy-efficient memory disaggregation in a

datacenter. In *Proceedings of the 13th European Conference on Computer Systems (EuroSys)*, page 16. ACM, 2018.

[51] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. The lru-k page replacement algorithm for database disk buffering. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, SIGMOD '93, page 297–306, New York, NY, USA, 1993. Association for Computing Machinery.

[52] Young Paik. Developing extremely low-latency nvme ssds. *Flash Memory Summit*, 2017.

[53] Ashish Panwar, Sorav Bansal, and K. Gopinath. Hawkeye: Efficient fine-grained os support for huge pages. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, page 347–360, New York, NY, USA, 2019. Association for Computing Machinery.

[54] SeongJae Park, Yunjae Lee, Moonsub Kim, and Heon Y. Yeom. Automating context-based access pattern hint injection for system performance and swap storage durability. In *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19)*, Renton, WA, July 2019. USENIX Association.

[55] SeongJae Park, Yunjae Lee, and Heon Y. Yeom. Profiling dynamic data access patterns with controlled overhead and quality. In *Proceedings of the 20th International Middleware Conference Industrial Track*, Middleware '19, page 1–7, New York, NY, USA, 2019. Association for Computing Machinery.

[56] Karl Pettis and Robert C Hansen. Profile guided code positioning. In *ACM SIGPLAN Notices*, volume 25, pages 16–27. ACM, 1990.

[57] Mike Rapoport. Idle page tracking. https://www.kernel.org/doc/html/latest/admin-guide/mm/idle_page_tracking.html, 2018.

[58] Janis Schoetterl-Glausch. Intel page modification logging for lightweight continuous checkpointing. *Bachelor thesis, Operating Systems Group, Karlsruhe Institute of Technology (KIT), Germany, October31*, 2016.

[59] Harald Servat, Antonio J Peña, Germán Llort, Estanislao Mercadal, Hans-Christian Hoppe, and Jesús Labarta. Automating the application data placement in hybrid memory systems. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 126–136. IEEE, 2017.

[60] Jakob Unterwurzacher. earlyoom(1). https://manpages.debian.org/testing/earlyoom/earlyoom.1.en.html, 2020.

[61] Carl Waldspurger, Trausti Saemundsson, Irfan Ahmad, and Nohhyun Park. Cache modeling and optimization using miniature simulations. In *2017 USENIX Annual Technical Conference (ATC)*, pages 487–498, Santa Clara, CA, 2017. USENIX Association.

[62] Haojie Wang, Jidong Zhai, Xiongchao Tang, Bowen Yu, Xiaosong Ma, and Wenguang Chen. Spindle: Informed memory access monitoring. In *2018 USENIX Annual Technical Conference (ATC)*, pages 561–574, Boston, MA, 2018. USENIX Association.

[63] Yijian Wang and David Kaeli. Profile-guided I/O partitioning. In *Proceedings of the 17th annual international conference on Supercomputing*, pages 252–260. ACM, 2003.

[64] Johannes Weiner. mm: thrash detection-based file cache sizing. https://lwn.net/Articles/552327/, 2013.

[65] Dan Williams. Filesystem-dax. Oakland, CA, February 2018. USENIX Association.

[66] Juncheng Yang, Yao Yue, and K. V. Rashmi. A large scale analysis of hundreds of in-memory cache clusters at twitter. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 191–208. USENIX Association, November 2020.

[67] Xusheng Zhan, Yungang Bao, Christian Bienia, and Kai Li. Parsec3.0: A multicore benchmark suite with network stacks and splash-2x. *ACM SIGARCH Computer Architecture News*, 44(5):1–16, 2017.

[68] Weixi Zhu, Alan L. Cox, and Scott Rixner. A comprehensive analysis of superpage management mechanisms and policies. In *2020 USENIX Annual Technical Conference (ATC)*, pages 829–842. USENIX Association, July 2020.