2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)

Mnemo: Boosting Memory Cost Efficiency in Hybrid Memory Systems

Thaleia Dimitra Doudali
Georgia Institute of Technology

Ada Gavrilovska
Georgia Institute of Technology

***Abstract*—For hosting data-serving and caching workloads based on key-value stores in clouds, the cost of memory represents a significant portion of the hosting expenses. The emergence of cheaper, but slower, types of memories, such as NVDIMMs, opens opportunities to reduce the hosting costs for such workloads. The question explored in this paper is how to determine adequate allocations of different memory types in future systems with heterogeneous memory components, so as to retain desired performance SLOs and maximize the cost efficiency of the memory resource. We develop Mnemo, a memory sizing and data tiering consultant, that permits quick exploration of the cost-benefit tradeoffs associated with different configurations of the hybrid memory components used by key-value store workloads. Using experimental evaluation with different workload patterns, Mnemo is able to afford applications such as Redis, Memcached and DynamoDB, with substantial reduction in their hosting costs, at negligible impact on application performance, thus improving the overall system memory cost efficiency.**

I. Introduction

Cost of Memory. In-memory key-value stores are traditionally used in data serving and caching applications, heavily deployed in native and cloud infrastructure. Such applications rely on the fast data retrieval that memory provides, compared to storage, in order to meet their clients Service Level Agreements (SLAs) and get the desired performance. In addition, they require significant memory capacity, in order to store the huge amount of data generated every minute in the Internet [17]. For this reason, major cloud providers like Amazon, Google and Microsoft Azure offer Memory Optimized Virtual Machines with significantly high capacities – in the few TBs range. They also offer Virtual Machines (VMs) with specialized support for the widely used in-memory key-value stores Redis and Memcached, like AWS ElastiCache.

In order to understand the isolated cost of memory in the cloud, across the different providers, we describe the hourly VM cost as a simplified function of the hourly cost C for a single vCPU and the hourly cost M for 1GB of memory, using the following equation:

$$\text{VM Cost} = \text{vCPU} \times C + \text{GB} \times M$$

We solve a system of equations derived from all VM instances per cloud provider, using the regression method of least squares, following the methodology described by Amur et al. [18]. More specifically, we estimate the memory cost of the `cache.r5` Memory Optimized AWS Elasticache [3] VM instance, the `n1-ultramem`, and `n1-megamem` Memory Optimized Google Compute Engine [6] VM instances and the `E`, and `M` Extreme Memory Optimized Microsoft Azure [11] VM instances.

In figure 1 we observe that the cost of memory approximately constitutes 60% to 85% of the overall VM cost, making capacity sizing decisions absolutely crucial with respect to the available cost budget and desired application performance levels.

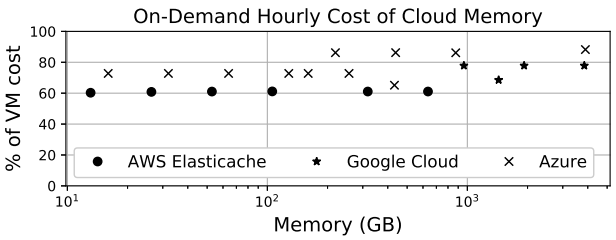


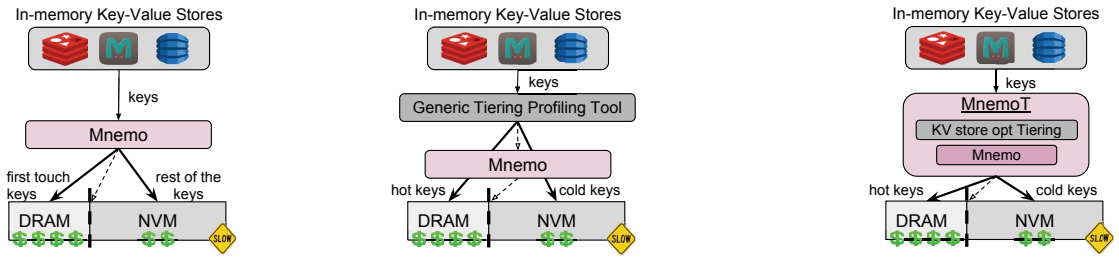
Fig. 1: Percentage of the cost of memory in select Memory Optimized Virtual Machines across major cloud providers.

Hybrid Memory Systems. This non-trivial cost of memory, coupled with the demand for high memory capacities and the scaling limitation of traditional DRAM technologies [29], has led industry to develop new types of memory technologies. In particular, Non-Volatile Memory DIMMs (NVDIMMs), such as Intel's upcoming Optane DC Persistent Memory [7], based on Intel's 3d-XPoint technology [1], present an attractive option for system integrators, as they are expected to have an order of magnitude higher density, compared to DRAM, at a much lower per unit cost. While the concrete price point of these technologies is not presently known, industry projections have estimated that NVDIMMs will offer a 3-7x reduction in per GB cost compared to DRAM [16], [24]. Such reduction introduces a potential for a 40-67% decrease in the VM costs, given estimates of the per-VM memory costs in Figure 1.

However, NVDIMMs also have higher access latency and lower bandwidth compared to DRAM, making it unlikely that they will become just a drop-in replacement for DRAM. The co-existence of these different technologies creates *hybrid memory systems*, usually consisting of one memory component that permits fast accesses, like DRAM, and a slower one, such as NVDIMMs. Future cloud systems are expected to provide VMs comprising both traditional DRAM and NVDIMMs with even higher capacity than the existing DRAM-based Memory Optimized VMs. In fact, Google is the first cloud provider to announce the availability of VMs with up to 7 TBs of Intel's Optane DC Persistent Memory in 2019 [4].

Data Management in Hybrid Memory Systems. Applications that execute on hybrid memory systems will have to deal with a potential performance degradation from the ideal case of executing with infinite DRAM-only capacity. This is the reason why, there has been substantial research effort into building intelligent data placement and manage-

978-1-5386-5555-9/19/$31.00 ©2019 IEEE
DOI 10.1109/IPDPSW.2019.00080



(a) Stand-alone. (b) Existing Tiering Solution + Stand-alone. (c) Optimized Key-Value Store Tiering + Stand-alone.

Fig. 2: Mnemo is a memory capacity sizing consultant, complimentary to data tiering solutions. Mnemo quickly explores the trade-offs between the memory system cost and application performance, providing users with the most cost-efficient memory capacity ratio that guarantees the desired performance. Mnemo can be utilized (a) stand-alone, (b) together with existing generic tiering profiling tools or (c) MnemoT extension provides the standalone functionality together with key-value store optimized and low profiling overhead tiering decisions.

ment techniques, ranging from application- [9], [24], [30], to operating system- [25], all the way down to hardware-level solutions [20]. These solutions focus on optimizing the data placement across a hybrid memory system, so that frequently accessed data can benefit from the fast access speeds of DRAM [24], [25], or trigger data migrations that allow for maximum bandwidth utilization of all memory components [20].

Problem Statement. Data tiering solutions can reduce the application performance slowdown when executing on hybrid memory system with *fixed capacities*, by optimizing their usage efficiency. In this paper, we target the orthogonal question of *how* should one determine *the ideal capacity ratio* between the fast and slow components, so as to maximize the system's cost efficiency without significantly impacting performance. In other words, what is the minimum amount of DRAM capacity that an application requires, in order to perform sufficiently well, respecting any performance expectations or guarantees. Experimental analysis with Redis [13], Memcached [10] and DynamoDB [2], the three currently highly rated key-value stores [5], shows that there can be workloads whose access pattern allows for a substantial reduction in the amount of DRAM required, thus the total memory cost, in return for trivial application performance degradation. For example, if a workload heavily accesses 20% of the keys, then a DRAM:NVM capacity ratio of more than 20:80 will give trivial performance improvement. Also, our results demonstrate that the potential for savings is highly workload-dependent, in fact performance is tightly coupled with the key access pattern, thus a-priori knowledge of the workload can provide very accurate projections of cost-performance trade-offs.

Solution Summary. Motivated by the observed behavior of cost vs application performance for variable DRAM:NVM capacity ratios, the potential for significant cost savings and the importance of workload knowledge, we present **Mnemo** – an open-source key-value store specific profiling tool which permits exploration of the cost-benefit tradeoffs of using hybrid memory systems. More specifically, Mnemo quickly produces an accurate trendline of application performance for incremental DRAM to NVM capacity ratio, thus incremental memory system cost. We envision Mnemo to be a profiling tool that can help users who deploy key-value stores on the cloud, quickly understand what capacity sizings of VMs with DRAM and VMs with NVM [4] provide the best trade-offs between application performance and memory cost, with respect to customer Service-Level-Agreements (SLAs) and system cost budget limitations.

Figure 2 illustrates the possible deployment scenarios of Mnemo as a key-value store workload profiling tool, for the purpose of memory capacity sizing.

1. **Stand-alone** (Figure 2a). In this configuration Mnemo calculates application performance estimates for incremental sizing of DRAM with the keys as they get accessed (touched) by the workload access pattern. Mnemo gets the necessary performance baselines by actual workload execution and provides the estimate via a simple yet extremely accurate analytical model.
2. **Existing Tiering Solution + Stand-alone** (Figure 2b). In this configuration, the user first utilizes existing tiering solutions, that are generic for any application type not just key-value store workloads. The tiering solution will provide the user with the DRAM key allocations that optimize perfomance in a hybrid memory system. Mnemo will then calculate performance estimates for incremental DRAM sizing following the tiered key ordering. In this way, Mnemo provides users with the most performance optimized *and* cost efficient tiering of the key space.
3. **Optimized Key-Value Store Tiering + Stand-alone** (Figure 2c). This is an extended version of the tool, *MnemoT*, that includes a custom tiering solution optimized for key-value store workloads with respect to the profiling overhead. MnemoT is now both a tiering and capacity sizing profiling tool, that provides users with quick and accurate tiering and cost efficient static placement decisions.

II. Experimental Setup

Testbed. Due to restricted access to commercially available hybrid memory platforms, we use native hardware to emulate

Node	FastMem	SlowMem
Factor	B:1 L:1	B:0.12 L:3.62
Latency (ns)	65.7	238.1
BW (GB/s)	14.9	1.81

TABLE I: Testbed Bandwidth and Latency values for DRAM (B:1 L:1) and emulated NVM (B:x L:y) of x times reduced bandwidth and y times increased latency.

Runtime	FastMem	SlowMem	Cost Reduction
Best Case	C bytes	0 bytes	0
In between	F bytes	S bytes	p = 0.2
Worst Case	0 bytes	C bytes	1

TABLE II: Short description of the performance baselines together with the corresponding capacity sizings and memory cost reduction factors. Cost reduction is captured as a factor of the FastMem-only cost.

a system with two memory components, one that is of high bandwidth and low latency (i.e., DRAM), referenced throughout the paper as **FastMem**, and one of significantly lower bandwidth and higher latency (i.e., NVDIMM), referenced as **SlowMem**. Our testbed consists of a 12-core, dual-socket Xeon platform, with two 4 GB DDR3 memory nodes and a 12 MB shared Last Level Cache. We emulate SlowMem via throttling the DRAM node of one socket, according to prior research [23], [25], [26]. The other socket remains unmodified and corresponds to FastMem. Table I summarizes the latency and bandwidth values of our testbed. We do assume that SlowMem is used as an extension of the flat memory address space, in other words FastMem does not serve the purpose of caching for SlowMem.

Server Configuration. We deploy Redis, Memcached and DynamoDB locally, on our native hardware testbed, using their default configuration for in-memory functionality. We do not make any modifications in the source code of the key-value stores. For this reason, we can set up a local deployment of the closed-source DynamoDB [14]. Since these key-value stores are designed for DRAM-only systems and in order to allow for variability in the data allocations across FastMem and SlowMem, we run two server instances on the same testbed. At time of deployment, we use the *numactl* command-line Linux utility, in order to bind the execution of the server processes to the CPU cores of the FastMem socket, and their memory allocations to one memory node, either FastMem or SlowMem exclusively. In this way, we will be able to establish performance baselines, where both server instances allocate all data in FastMem or SlowMem, respectively. These baselines will be critical for the design of Mnemo, as explained in Section IV.

Client Configuration. We use the Yahoo! Cloud Serving Benchmark [21], in order to capture server performance from a client perspective. The client is configured to run on the same testbed as the server, without interference, so as to reduce any network-related noise and to truly evaluate the impact of the hybrid memory on the server. The core module of the client is modified, so it can redirect requests across the two server instances.

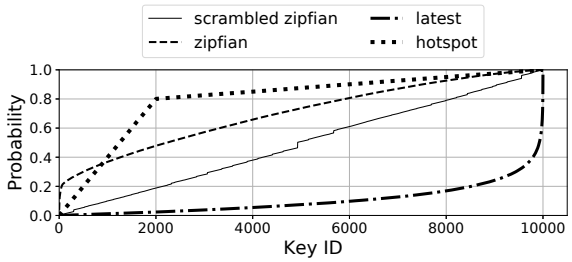


Fig. 3: CDF of the key space across different request pattern distributions. Shows the probability for a key ID to be requested throughout the workload.

Workloads. We further adapt the default YCSB workloads, so that they can represent a broad range of request distributions and data sizes of modern data serving application scenarios. Table III summarizes the characteristics of the workloads we define, together with representative use cases that correspond to common workload patterns in the widely-used social media platform, Facebook. The matching of Facebook actions to request distributions and operation ratios is done with respect to information provided by [19], [21]. Figure 3 visualizes the distributions included in the custom workloads. The `scrambled zipfian` distribution contains some hot (heavily accessed) keys scrambled across the key space, whereas the `zipfian` maps these keys at the beginning of the key range. Concerning data sizes, we try to capture both text and photo use cases. Publicly available "social media cheat sheets" [8], [15], include the sizes of photos in pixels and size of text posts in character length for all widely used social media platforms. We infer the size distributions in the average case for specific data use cases, as depicted in Figure 4. We limit the exploration to these use cases, and do not include bigger data sizes, such as videos, due to restricted memory capacity of our native testbed.

System Cost Baselines. Table II summarizes the performance baselines in a hybrid memory system, the corresponding memory capacity sizings, and the cost reduction factor. If we assume that SlowMem is p times cheaper, per byte, than FastMem, then $R(p)$ will be the cost reduction factor of the total memory system, according to the following model:

$$R(p) = \frac{H_{cost}}{F_{cost}} = \frac{F \times F_{cost/byte} + S \times S_{cost/byte}}{C \times F_{cost/byte}}$$

$$R(p) = \frac{F + (C - F) \times p}{C}, \quad \text{where } 0 < p < 1$$

Throughout the paper we fix **p = 0.2**, based on price estimates used in prior research [24]. In real usage scenarios, this price factor can be derived from actual memory hardware cost, or the pricing of Virtual Machine instances with a choice of memory technology.

Workload	Distribution	Read:Write ratio	Record Size Type	Use Case
`Trending`	hotspot	100:0 readonly	thumbnail (≈100 KB)	Read Facebook short Trending News.
`News Feed`	latest	100:0 readonly	thumbnail (≈100 KB)	Read Facebook News Feed.
`Timeline`	scrambled zipfian	100:0 readonly	thumbnail (≈100 KB)	Read Facebook user's Timeline.
`Edit Thumbnail`	scrambled zipfian	50:50 updateheavy	thumbnail (≈100 KB)	Edit Profile Photo - Add filter/frame.
`Trending Preview`	hotspot	100:0 readonly	thumbnail (≈100 KB) text post (≈10 KB) photo caption (≈1 KB)	Scroll through Facebook Trending News. Preview the news photo thumbnail, caption and news summary.

TABLE III: Custom YCSB workloads adapted to modern use case scenarios of social media platforms, capturing a broad range of request distributions, operations ratio and size of data typically serviced by in-memory key-value stores. Number of keys is 10,000 and number of requests 100,000.

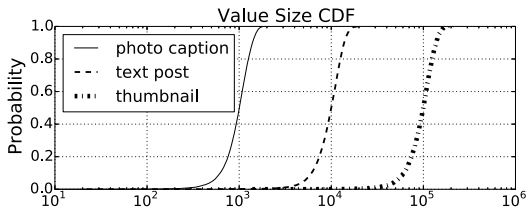


Fig. 4: CDF of common data sizes used across social media platforms. Horizontal axis depicts size (Bytes) in logarithmic scale.

III. Motivation

The experimental analysis that follows aims to capture the workload parameters that influence the performance of a key-value store on a hybrid memory system, in order to highlight the scope of potential memory cost reduction. Figure 5 shows the behavior of the Redis client runtime performance as a function of the memory cost, when increasing the FastMem to SlowMem capacity ratio. We describe in more detail the observations regarding the parameters that influence application performance of an in-memory key-value store.

Key distribution. Figure 5a shows that the throughput of Redis is significantly improved when using FastMem, up to 40% compared to the case where all data is allocated in SlowMem. Looking back to Figure 3 we can reason that the throughput improvement of all workloads seems to follow their corresponding key access distribution pattern, as indicated by the estimate lines, that will be further explained in Section IV. It is important to notice here that in workloads like `trending` which have a small set of frequently accessed keys, there is a potential of significantly reducing the memory cost in return for trivial performance loss. For example, sizing FastMem such that it only holds the hot keys will reduce the system's memory cost to be only 36% of the cost of using only FastMem, in return for 31% throughput improvement from the SlowMem-only case, and only 10% less throughput than the ideal case of FastMem-only allocations. Therefore, the knowledge of the access distribution across the key space, can facilitate optimal sizing decisions, and thus enable significant cost reduction for minimal performance degradation, depending on the workload.

Read:Write ratio. Similarly, Figure 5b shows the relationship between performance and cost for different amounts of read and write requests in the workload. We observe that write heavy workloads, such as `edit thumbnail` are less impacted by the heterogeneity of the memory subsytem, rather than read heavy ones such as `timeline`. Read requests are more likely to translate to an actual memory access, and thus are more exposed to any latency slowdown of the memory. In the context of our analysis, this observation shows the importance of capturing and differentiating the write and read requests of a workload.

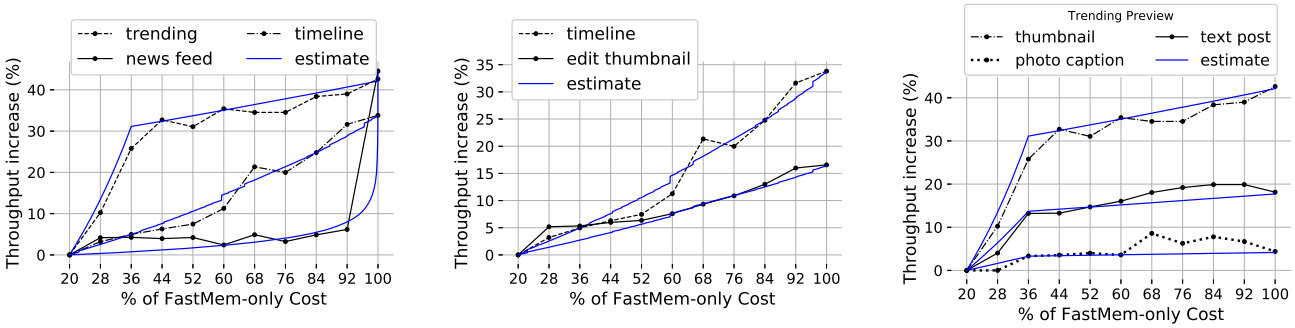
Record size. Figure 5c shows the impact of the record size on application performance. Big record sizes influence performance in a much more significant way than small record values (the knee of the line is bigger). This further shows that in such workloads, where data of different granularities need to be fetched, it is more important for the large records to be allocated in FastMem, compared to small objects, which can reside in SlowMem without impacting application runtime.

Takeaways. First, we observe that, when right-sizing the memory components, there is scope to significantly reduce the memory cost of a system while triggering minimal performance slowdown, especially for workloads with small sets of frequently accessed keys. Second, we see that application performance is tightly coupled with the key access pattern, the read:write ratio and record size. Therefore, *knowledge of these workload characteristics can facilitate the exploration of performance and cost tradeoffs*, when deciding about the capacity ratio of a hybrid memory system.

IV. Design and Implementation

Design Motivation. The observations regarding workload behavior on hybrid memory systems from Section III, and the need to reduce the system's memory cost given the FastMem pricing references in Section I, make a case for Mnemo. We argue that the presence of heterogeneous memory components will open new opportunities for workloads and platform operators to leverage a new cost-performance tradeoff in terms of the sizing of the different memory capacities. With that in mind, we design and build Mnemo– an application profiling tool that can provide insights into this tradeoff for data serving applications, an important class of memory intensive workloads.

Mnemo Overview. Mnemo is an open-source, easy to setup tool, designed for capacity sizing analysis of key value stores on hybrid memory systems. It permits its users, application or infrastructure operators, to understand the impact on cost and performance from different distributions of a workload's



(a) Varying key distribution.

(b) Varying read:write request ratio.

(c) Varying record size.

Fig. 5: Application performance of Redis for incremental FastMem to SlowMem capacity ratio (left to right). As the capacity of FastMem increases, so does the memory cost (x-axis) as well as the application throughput (y-axis), compared to the case of using only SlowMem (left bottom point). Reported values are the mean of multiple experiments runs. The solid blue line corresponds to Mnemo's performance estimate described in Section IV. We report the measurement variability through the estimate's accuracy in Section V.

memory capacity across different types of memory components. The output are cost estimation curves, similar to the graphs plotted via the measurements in Section III.

Mnemo does not require any application level modifications and profiles the application quickly, as it does not perform fine-grained execution monitoring. Instead, users are expected to provide Mnemo with a target workload descriptor, comprised of the workload features identified in Section III: key access distribution and request type sequence for a given dataset.

In its simplest form, these parameters are obtained directly by running the key-value store with a representative key and request type sequence. Then, Mnemo executes the actual workload on a physical hybrid memory system "as-is". The goal is to obtain real performance baselines corresponding to the two extreme configurations – when all memory capacity is allocated from FastMem vs. when only SlowMem capacity is used. The two baselines establish the bounds for the tradeoff estimation curve.

Next, Mnemo relies on the estimation models, described later in this section, to quickly calculate an estimate of the performance degradation for increasing capacity ratio of FastMem compared to SlowMem. Mnemo does not answer the question of what is the total capacity needed for a workload; that is an orthogonal question, and Mnemo uses a fixed total capacity to be the dataset size of the key-value store. The generated estimate enables users to choose the exact capacity sizing of the hybrid memory on the server side, at a key size granularity, that will enable cost efficiency and desirable performance.

Design Principles. Mnemo follows techniques that are widely-used in profiling tools, such as establishing performance baselines via workload execution and having analytical models for performance estimation. However, Mnemo focuses on delivering low overhead calculations, so as to provide users with quick and accurate cost-benefit trade-offs. Section V includes detailed comparison of Mnemo's design choices with respect to the profiling overhead.

Figure 5a hints that the application performance trendline follows the request distribution. Mnemo builds on that observation and given the request pattern information makes perfromance predictions following a very simple model. This model produces an estimate of the workload's performance based on the intuitive observation that the total runtime will be the product of the number of pending read and write requests with the average service time of read and write requests by the data store, for a given key tiering across FastMem and SlowMem. Thus, throughput (requests per second) will be the runtime divided by the total number of requests. For this model to work, Mnemo requires only two things: a-priori knowledge (or description) of the workload, and real performance baselines for the average read and write time. In Section V we see how such a simple design and light-weight implementation can provide almost perfect estimation accuracy of less than 0.1% median error.

Mnemo Architecture. Figure 6 shows a detailed representation of the software components of Mnemo, and the data flow through them:

1. The *Sensitivity Engine* is a customized YCSB client, which executes the actual workload itself, issuing a user-provided sequence of keys and request types. It determines the performance baselines for the best case, where all data is in FastMem, and worst case, where all data is in SlowMem, including average total runtime and average read and write request response times, as shown in the data flow of Figure 6.
2. The *Pattern Engine* analyzes the request access pattern of the workload, and establishes a relationship between the keys and requests $Req(keys)$.
3. The *Estimate Engine* takes as input the performance baselines from the Sensitivity Engine, the access pattern from the Pattern Engine, and the memory cost reduction factor p from the Mnemo user. Mnemo calculates the work-

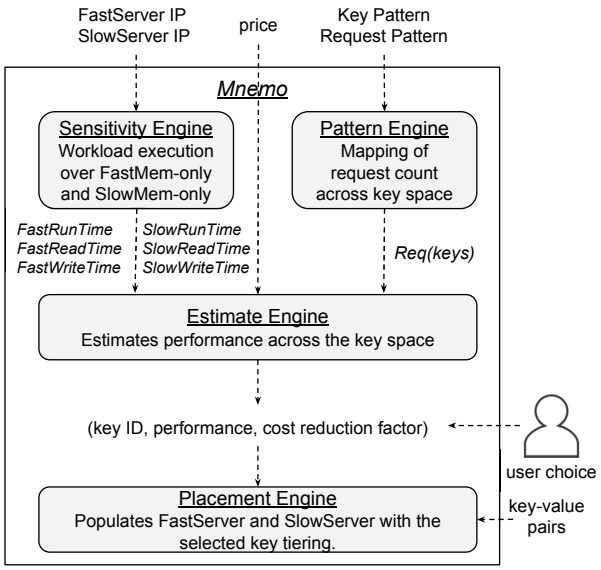


Fig. 6: Data flow and functionality description of the software components of Mnemo.

load's throughput for incremental tiering of the key space across FastMem and SlowMem, according to the following equation. It then correlates the throughput to the system cost, by calculating the cost reduction factor based on the FastMem to SlowMem capacity ratio that corresponds to each key tiering, following the cost model described in Section II.

$$Throughput = \frac{ReadTime + WriteTime}{Requests}$$

$$ReadTime = reads \times (SlowReadTime - FastReadTime)$$

$$WriteTime = writes \times (SlowWriteTime - FastWriteTime)$$

4. The *Placement Engine* takes the selected key tiering, that satisfies the user's performance to cost trade-offs, and statically places the key-value pairs to the corresponding FastServer and SlowServer, prior to the actual workload execution. At this step, the user needs to provide Mnemo with the actual dataset and not just the initial workload descriptor. However, this step is optional and can be performed by the user itself. Either way, Mnemo provides a static key allocation, with no support for dynamic data migration.

It is important to highlight the fact that server-side parameters, such as the server thread parallelism, hardware cache and prefetching efficiency, or the network speed, that define the processing speed of the key-value store on the given CPU, memory and network, are all incorporated into the average request response time ($ReadTime, WriteTime$) that the Sensitivity Engine extracts by actually executing the workload. Therefore, our model, even though it is simple enough, it is able to capture the needed information and generate a performance estimate with very strong accuracy.

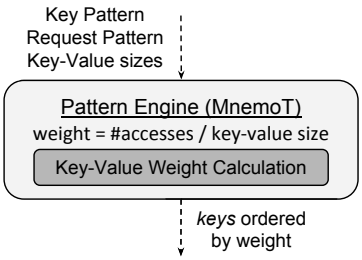


Fig. 7: Pattern Engine of MnemoT. Orders keys for prioritized FastMem allocations, similar to existing tiering solutions. The rest components of MnemoT are exactly the same with Mnemo.

Interfacing with Mnemo. Mnemo users need to provide as *inputs* the target workload, in a form of a key sequence and the corresponding request type, identifiers for the key-value store servers, and a cost reduction factor p of the target SlowMem compared to FastMem. As *output*, Mnemo will generate a text file in csv format with three columns, as depicted in Figure 6, as well as a graph representation of the estimate, as the solid blue line in Figure 5. Each row contains a key identifier, the estimated performance and cost reduction factor, when FastMem will service all previous keys in the file and have capacity equal to the sum of their corresponding values, whereas the rest of the keys, that follow in the output lines, will be attributed to SlowMem. The user of Mnemo should choose the line that satisfies its performance requirements and price allowance and then the Placement Engine will appropriately populate the FastMem and SlowMem.

Key-Value Store Optimized Tiering Extension. *MnemoT* is an extended version of Mnemo, with the exact same components and functionality depicted in Figure 6. MnemoT features a more robust *Pattern Engine* that analyzes the request access pattern and produces a priority ordering of the keys for FastMem allocations, using the tiering methodology that existing solutions use [24], [30], [32], [33]. In more detail, the Pattern Engine now takes as an input the key-value sizes and associates each key with a placement weight. The weight is the number of accesses the key receives, divided by the size of the key-value pair. In this way, keys that are heavily accessed (hot keys) are prioritized for DRAM allocations, as well as small keys also get an advantage, so that more key-value pairs can be satisfied by FastMem until capacity is full. We adopt his methodology, since it is predominantly used across most of the affore-mentioned solutions, because it provides optimal performance in hybrid memory systems. Some of the existing solutions map the tiering problem to the 0/1 knapsack, where the items are the key-value pairs, together with their calculated weights and sizes, and the size of the knapsacks are the fixed capacities. Figure 7 summarizes the input, output and internal functional of the extended Pattern Engine that MnemoT features. Section V contains detailed comparison of this tiering methodology compared to existing solutions, with respect to the profiling overhead.

V. Evaluation

We evaluate Mnemo on the same testbed described in Section II, using the workloads summarized in Table III across the three currently most rated in-memory key-value stores, Redis, DynamoDB and Memcached. In Subsection V-A our goal is to evaluate the utility of Mnemo, regarding the performance estimate accuracy, the scope of memory cost reduction and the practical use of our tool. Then, in Subsection V-B, we compare the design choices of Mnemo and MnemoT with existing profiling solutions, focusing on the fact that it provides low overhead and fast calculations.

A. Mnemo's Utility

Estimate Accuracy. Figure 5 includes the corresponding estimate curves for Redis. To justify the accuracy of Mnemo we keep track of the percentage error $\frac{r-e}{r} \times 100\%$ between the real performance points r and their corresponding estimate e, across all experiments. We repeated the experiments with the workloads defined in Table III for the three currently most highly rated in-memory key-value stores [5], Redis, DynamoDB and Memcached. The data distribution of the error values for each key-value store is presented in Figure 8a, in the form of boxplots. The strong accuracy of 0.07% median error highlights that the usage of real performance baselines and the knowledge of the actual request access pattern, put together with a simple model, can efficiently and quickly estimate performance.

Key-value store Comparison. Figure 8b shows the application performance across DynamoDB, Redis and Memcached for the `Trending` workload. We choose to visualize these results for brevity and highlight the scope at which Mnemo can be useful. First, this is a very representative class of workloads across all social media platforms and generally cloud based data serving applications, where a certain subset of data is heavily requested for a certain period of time. More so, all workloads can be profiled in a way that orders keys with respect to request counts, thus transformed to a `Trending` version. Mnemo is able to capture this subset of data that is absolutely necessary to be allocated in FastMem, so as to satisfy any performance guarantees. This is what will determine the capacity ratio of FastMem to SlowMem and thus the overall memory cost. In such workloads, there is potential for huge cost reductions when the set of hot data is small compared to the whole dataset, as shown in Figure 9 and described later on.

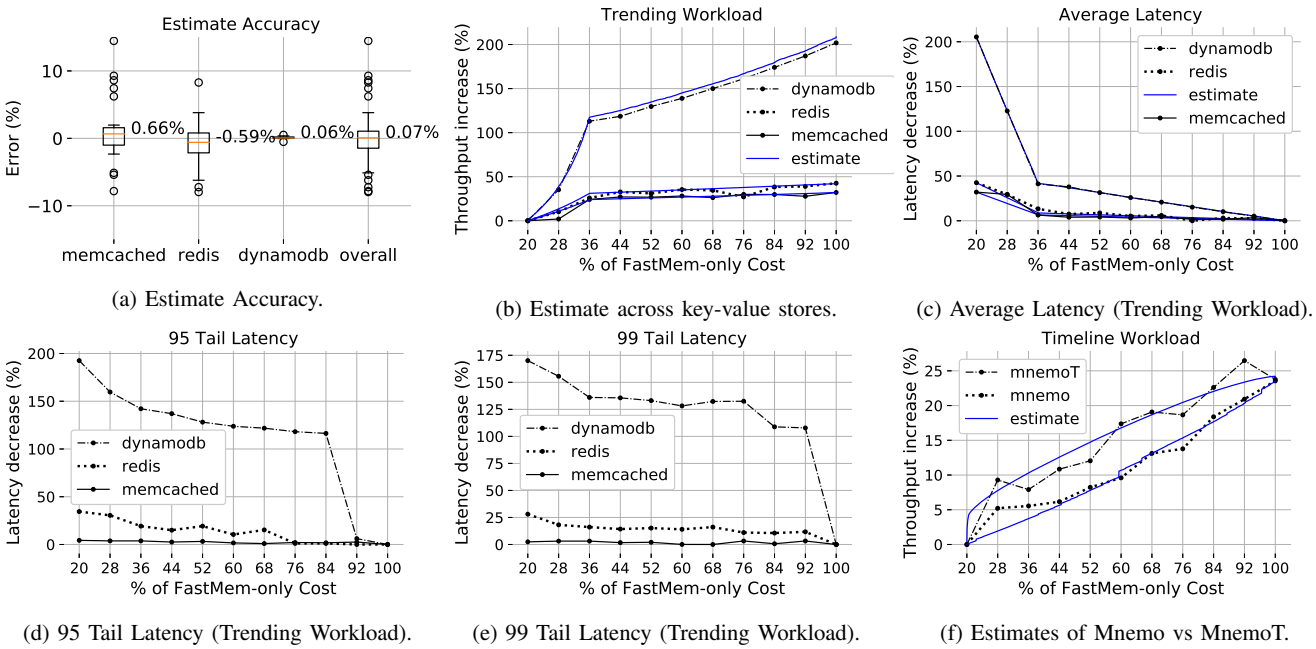
Second, we observe that the internals of a key-value store and the request processing rates it can achieve, set the degree of overall sensitivity to execution on a hybrid memory system. In our native experimental setup we observe that DynamoDB is severely impacted when allocating data in SlowMem, whereas Memcached barely gets influenced. Although it is out of the scope of this paper to understand the reasons of this sensitivity variation, what's important is that Mnemo can capture this behavior, through the performance baselines it acquires. In this way, Mnemo can shed light to whether the user even needs to be concerned about application performance over hybrid memory or not.

Third, as far as the performance metrics are concerned, Mnemo makes perfromance estimates in terms of throughput (operations / seconds). Regarding latency, Mnemo estimates the Average latency to service a request from the client perspective, based on the performance baseline it acquires and the simple estimation model described in Section IV. The estimate is extremely accurate, as depicted in Figure 8c. However, regarding the tail latency of the requests, Mnemo does not produce any estimate, since the simple analytical model it uses is not sufficient to capture the variabilities of the tail latencies. We report those in Figures 8d, 8e.

Estimate of MnemoT. The Pattern Engine of MnemoT analyzes the request frequency of the key-value pairs and priotizes them for FastMem allocations, similarly to existing solutions, as described in Section IV. In this way hot keys will be the first ones to be considered for FastMem allocations. Looking back at Figure 3 this is similar to identifying the hot keys of `scrambled zipfian` distribution, that are spread across the key space, and prioritizing them in the beginning of the key space, converting the distribution to look like `zipfian` Similarly, the Pattern Engine of MnemoT identifies the hot keys and transforms the input distribution into a `zipfian` like one. Figure 8f shows the performance estimate of Mnemo vs MnemoT, proving that the estimate model is also accurate for the new ordering of keys.

It is important to note that existing tiering solutions, work over *fixed* memory capacities, thus if used they would provide only 1 point of the curve that Mnemo produces. For example, let's assume we have a 70:30 FastMem:SlowMem capacity ratio (76% of FastMem-only cost). Then, data tiering (MnemoT) will provide almost 6% more throughput. However, the tiered throughput is only 7% less than the optimal one, when only FastMem is used. The application performance guarantees may allow for 10% less throughput compared to FastMem-only, which can be achieved with 50:50 FastMem:SlowMem and only 52% of FastMem-only cost. MnemoT facilitates this observation for significant cost reduction, that existing tiering solutions cannot provide, unless the user manually runs them across variable capacity ratios. MnemoT does that automatically and quickly and allows users to observe not only the performance benefits through tiering, but also to find the sweet spot between cost and performance.

Memory Cost Efficiency. Figure 9 shows the scope of the cost reduction Mnemo is able to provide, while allowing application performance degradation to be 10% from the ideal case where all data could reside in FastMem. We choose to represent the 10% performance degration SLO, as it is commonly used in other research on optimizing performance and resource efficiency [27], [31], [34]. The lower the cost reduction is, the more cost savings a user can obtain. The minimum threshold in this experiment corresponds to 20%



(a) Estimate Accuracy.

(b) Estimate across key-value stores.

(c) Average Latency (Trending Workload).

(d) 95 Tail Latency (Trending Workload).

(e) 99 Tail Latency (Trending Workload).

(f) Estimates of Mnemo vs MnemoT.

Fig. 8: Evaluation of Mnemo's estimate with respect to the overall accuracy across key-value stores.

which is determined by the assumption that SlowMem will be $p = 0.2$ times the cost of FastMem.

First, we observe that Memcached is overall non-sensitive to execution over SlowMem, allowing for maximum cost savings, where it runs solely on SlowMem, without affecting the application more that 10%.

Redis shows more interesting results, and highlights the contribution of the workload access pattern to the scope of possible cost reduction. More specifically, workloads in the `Trending` category, which contain a small subset of hot keys, can perform with an overall slowdown of 10%, while utilizing only the absolutely necessary amount of FastMem to host the hot keys. This can reduce the cost close to the baseline of 20% of the (expensive) FastMem-only cost. On the other side, `News Feed` workloads, really depend on the latest accessed data to reside in FastMem, thus can only allow very small portions of SlowMem to be used, and barely present any cost reduction opportunities. Next, comparing the `Timeline` and `Edit Thumbnail` workloads that follow a more regular access pattern, the latter benefits from the heavy amount of writes, that are not affected by SlowMem, and allow for more cost savings compared to read heavy workloads.

Finally, we observe that DynamoDB is the most impacted when executing over SlowMem, tolerating only small amounts of SlowMem capacity to be used in order to respect the performance guarantees. However, even for DynamoDB, for certain access patterns, we observe an opportunity to reduce the memory cost by 20-30%. Given the trend toward growing in-memory data stores, and memory capcities in the 100s of GBs and beyond, these cost reductions still represent a meaningful fraction of the infrastructure cost.

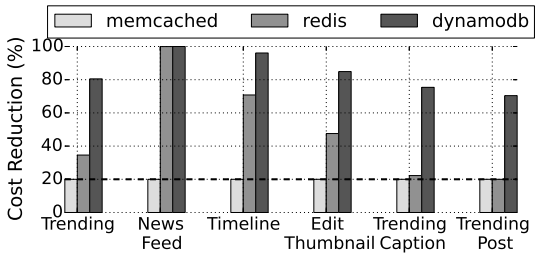


Fig. 9: Cost reduction across all workloads and key-value stores for performance that adheres to 10% permissible application slowdown. The lower the cost the better, with a threshold of 20%, which is the assumed relative cost of using only SlowMem compared to FastMem. Cost reduction is extrapolated using the formula in Section II.

Workload downsampling. Mnemo requires a-priori knowledge about the workload access pattern, and its estimation model depends on this to achieve its accuracy. In real use case scenarios, this knowledge may be restricted, either due to lack of access to the actual workload or due to their significant size, consisting of millions of requests. Thus, the user may either create a synthetic workload with similar request distribution or downsize a real workload via sampling the number of requests.

For this reason, we downsize our workloads via random sampling, where we choose to evict from the workload random key requests at fixed intervals. This reduces the number of requests issued, but ensures that the characteristics of the original key distribution are preserved. Our experiments show that Mnemo still produces an accurate performance estimate and that the downsized workload's performance is affected

Profiling Step	MnemoT	Other Solutions	Lowest Overhead?
Input Preparation	Collect the workload (key access pattern + key-value pair sizes)	Collect the workload (key access pattern) & instrument the server code with custom API [24], [30], [32]	*MnemoT*
Performance Baselines	Actual Execution of Workload with FastMem-only and SlowMem-only data allocations.	Obtaining the FastMem and SlowMem access latency through prior microbenchmark execution [24]. Actual workload execution of one performance baseline and inference of the other one, through a pre-trained Machine Learning Model [33].	*MnemoT*
Tiering	Weight Calculation on a key-value pair granularity using just an input description of the key-value pair sizes and request distribution.	Weight Calculation on a per server's internal data structure level (e.g. slab) using low-level memory access instrumentation [24], [30] or sampling low-level architecture counters (e.g. hardware cache misses) [32], [33]	*MnemoT*

TABLE IV: Comparison of the profiling overheads between MnemoT and existing tiering solutions.

in the same degree as the original workload. In this way, Mnemo can establish the same performance baselines as with the full-sized workload. Then, together with the accurate performance estimate that follows the request distribution, Mnemo ensures to deliver cost-to-performance trade-offs that will be applicable in the full-sized workload.

Target applications. Mnemo can run against any key-value store, as a black box, as its Sensitivity Engine includes a customized version of YCSB that can execute against popular data stores. Mnemo follows a very simple performance estimation model, which proves to be extremely accurate specifically for in-memory key-value data stores. We do not argue that the estimation model will work for any data store, especially those engaging storage components. Rather, data accesses that go through the storage subsystem, need to be appropriately studied and modeled, in order to capture the relevant impact to the application runtime.

B. Profiling Overhead Comparison

MnemoT's profiling overhead breaks down into the time taken for the user to prepare the input, for the Sensitivity Engine to extract the performance baselines and the Pattern Engine to analyze the key access pattern and calculate the tiering ordering. The Estimate Engine runs a simple analytical model, so its execution is instantaneous. Table IV summarizes the overhead comparison with existing profiling solutions, that we next discuss in detail.

First, as far as the *input preparation* is concerned, as with any application profiling tool, the user needs to provide the workload itself. Next, most profiling tools need to instrument the application source code, in order to facilitate the monitoring of the metrics for their analysis. Tiering profiling tools need to keep track of the memory accesses that correspond to the application's data structures, thus usually expose a custom memory allocation API to the user [24], [30], [32]. In this way, the user needs to spend significant amount of time understanding the internal application functionality, so as to properly utilize the custom API. In contrast, MnemoT treats the key-value store as a black box and requires no modification to its source code or understanding of its internal data structures. Thus, MnemoT requires only the workload description as any profiling tool does.

Next, regarding the time to collect *performance baselines*, MnemoT chooses to actually execute the workload 'as-is' in the two extreme cases, where all data is allocated in FastMem and similarly in SlowMem. This methodology, of establishing workload characteristics requirements during a brief pre-deployment stage, is very common for tools that are built to satisfy goals and performance concerns, such as effects of sharing memory (DRAM), CPUs or other resources for collocated workloads [22], [28], [35], [36]. In the hybrid memory system domain, X-Mem [24] runs microbenchmarks in order to get the latency of the different memory components across different access patterns. The authors of Tahoe [33] execute the workload itself and obtain the all-in-SlowMem performance baseline and choose to train Machine Learning models in order to infer the all-in-FastMem baseline. Although they claim that the training and inference time is trivial, the time to collect the training data, via workload execution and monitoring of hardware level counters, is significant. In contrast, by using both performance baselines, Mnemo results in a much faster and a far less complicated procedure.

Finally, concerning the *tiering* calculations, existing solutions involve rigorous application profiling [24], [30], [32], [33], in order to determine the access frequency of the application's data structures. They utilize binary instrumentation tools, like Intel's Pin [12], or low-level performance counters using Precise Event- Based Sampling from Intel or Instruction-based Sampling from AMD in order to identify the various data structures and keep track of every individual memory access in order to then calculate the individual weights and order the data objects for FastMem allocations. The utilization of such tools as part of the profiling solution, can add up to 40x overhead, as per the authors of X-Mem [24]. However, in the case of key-value stores, we can quickly calculate the allocation weights on a key-value pair granularity, as the Pattern Engine of MnemoT does. For this purpose, we only need a description of the key and value sizes, not the actual data, compared to existing solutions. In this way, the calculation of the object weights can be instantaneous, with no need for low-level memory access monitoring. Figure 8f validates that this methodology can still produce good tiering propositions and Mnemo's estimate is still extremely accurate. Therefore, MnemoT still follow's the predominant methodologies in the calculation of the allocation ordering, but can do so at zero overhead compared to existing profiling solutions.

Overall, MnemoT with its design choices is able to deliver fast performance estimations with minimal user effort.

VI. Summary

This paper presents the design and implementation of Mnemo, an open-source key-value store application profiling tool that aims to provide users with a cost-benefit presentation of the different sizing configurations of a hybrid memory subsystem. Motivated by the potential to drastically reduce the system's memory cost with minimal performance slowdown for workloads with frequently accessed data, Mnemo is able to automate the process of finding the sweet spot between cost efficiency and ensured performance guarantees. It does so by accurately estimating the application runtime degradation across incremental sizing of FastMem compared to SlowMem, for fixed overall capacity. Summarizing its impact, Mnemo is able to estimate performance with only 0.07% median error across Redis, Memcached and DynamoDB and shows the potential to reduce the memory cost down to only 20% of the cost of a DRAM-only system, while respecting performance guarantees of 10% application slowdown.

References

[1] Intel confirms Optane DIMMs for late 2018 - The Tech Report. https://techreport.com/news/32841/intel-confirms-optane-dimms-for-late-2018, Nov 2017.

[2] Amazon DynamoDB - NoSQL Cloud Database Service. https://aws.amazon.com/dynamodb/, May 2018.

[3] Amazon Elasticache Pricing. https://aws.amazon.com/elasticache/pricing/, November 2018.

[4] Available first on Google Cloud: Intel Optane DC Persistent Memory. https://cloud.google.com/blog/topics/partners/available-first-on-google-cloud-intel-optane-dc-persistent-memory, November 2018.

[5] DB-Engines. Ranking - popularity ranking of key-value store. https://db-engines.com/en/ranking/key-value+store, May 2018.

[6] Google Compute Engine Pricing. https://cloud.google.com/compute/pricing, November 2018.

[7] Intel Optane DC Persistent Memory. https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html, November 2018.

[8] Know Your Limit: The Ideal Length of Every Social Media Post. https://sproutsocial.com/insights/social-media-character-counter/, May 2018.

[9] Legion Overview - Legion Programming System. http://legion.stanford.edu/overview/, Feb 2018.

[10] Memcached - a distributed memory object caching system. https://memcached.org/, May 2018.

[11] Microsoft Azure Linux Virtual Machines Pricing. https://azure.microsoft.com/en-us/pricing/details/virtual-machines/linux/, November 2018.

[12] Pin - A Dynamic Binary Instrumentation Tool. https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool, November 2018.

[13] Redis. https://redis.io/, May 2018.

[14] Setting Up DynamoDB Local (Downloadable Version). https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/DynamoDBLocal.html, November 2018.

[15] Social Media Cheat Sheet 2018: Must-Have Image Sizes! https://louisem.com/2852/social-media-cheat-sheet-sizes, May 2018.

[16] System Memory at a Fraction of the DRAM Cost. https://www.intel.com/content/dam/www/public/us/en/documents/brief/intel-ssd-software-defined-memory-with-vm.pdf, 2018.

[17] What Happens in an Internet Minute in 2018? http://www.visualcapitalist.com/internet-minute-2018/, November 2018.

[18] H. Amur, W. Richter, D. G. Andersen, M. Kaminsky, K. Schwan, A. Balachandran, and E. Zawadzki. Memory-efficient groupby-aggregate using compressed buffer trees. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 18:1–18:16, New York, NY, USA, 2013. ACM.

[19] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. *SIGMETRICS Perform. Eval. Rev.*, 40(1):53–64, June 2012.

[20] C.-C. Chou, A. Jaleel, and M. Qureshi. BATMAN: Maximizing Bandwidth Utilization for Hybrid Memory Systems. In *Technical Report, TR-CARET-2015-01 (March 9, 2015)*, 2015.

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

[22] C. Delimitrou and C. Kozyrakis. Paragon: Qos-aware scheduling for heterogeneous datacenters. *SIGPLAN Not.*, 48(4):77–88, Mar. 2013.

[23] T. D. Doudali and A. Gavrilovska. Comerge: Toward efficient data placement in shared heterogeneous memory systems. In *Proceedings of the International Symposium on Memory Systems*, MEMSYS '17, pages 251–261, New York, NY, USA, 2017. ACM.

[24] S. R. Dulloor, A. Roy, Z. Zhao, N. Sundaram, N. Satish, R. Sankaran, J. Jackson, and K. Schwan. Data tiering in heterogeneous memory systems. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, pages 15:1–15:16, New York, NY, USA, 2016. ACM.

[25] S. Kannan, A. Gavrilovska, V. Gupta, and K. Schwan. HeteroOS - OS Design for Heterogeneous Memory Management in Datacenter. In *44th International Symposium on Computer Architecture (ISCA'17)*, Toronto, ON, 2017.

[26] S. Kannan, A. Gavrilovska, and K. Schwan. pvm: Persistent virtual memory for efficient capacity scaling and object storage. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, pages 13:1–13:16, New York, NY, USA, 2016. ACM.

[27] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis. Heracles: Improving resource efficiency at scale. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, pages 450–462, New York, NY, USA, 2015. ACM.

[28] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa. Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, pages 248–259, New York, NY, USA, 2011. ACM.

[29] O. Mutlu. Main memory scaling: Challenges and solution directions. In *More than Moore Technologies for Next Generation Computer Design*, chapter 6, pages 127–153. Springer, 2015. Invited Book Chapter.

[30] D. Shen, X. Liu, and F. X. Lin. Characterizing emerging heterogeneous memory. In *Proceedings of the 2016 ACM SIGPLAN International Symposium on Memory Management*, ISMM 2016, pages 13–23, New York, NY, USA, 2016. ACM.

[31] P. Tembey, A. Gavrilovska, and K. Schwan. Merlin: Application- and platform-aware resource allocation in consolidated server systems. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, pages 14:1–14:14, New York, NY, USA, 2014. ACM.

[32] K. Wu, Y. Huang, and D. Li. Unimem: Runtime data managementon non-volatile memory-based heterogeneous main memory. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '17, pages 58:1–58:14, New York, NY, USA, 2017. ACM.

[33] K. Wu, J. Ren, and D. Li. Runtime data management on non-volatile memory-based heterogeneous memory for task-parallel programs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, SC '18, pages 31:1–31:13, Piscataway, NJ, USA, 2018. IEEE Press.

[34] H. Yang, A. Breslow, J. Mars, and L. Tang. Bubble-flux: Precise online qos management for increased utilization in warehouse scale computers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 607–618, New York, NY, USA, 2013. ACM.

[35] C. Zhang. *DeepDive: A Data Management System for Automatic Knowledge Base Construction.* PhD thesis, University of Wisconsin-Madison, 2015.

[36] Y. Zhang, D. Meisner, J. Mars, and L. Tang. Treadmill: Attributing the source of tail latency through precise load testing and statistical inference. In *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16, pages 456–468, Piscataway, NJ, USA, 2016. IEEE Press.