Flexible and Effective Object Tiering for Heterogeneous Memory Systems[*]

BRANDON KAMMERDIENER, University of Tennessee, USA
J. ZACH MCMICHAEL, University of Tennessee, USA
MICHAEL R. JANTZ, University of Tennessee, USA
KSHITIJ A. DOSHI, Intel Corporation, USA
TERRY JONES, Oak Ridge National Laboratory, USA

Computing platforms that package multiple types of memory, each with their own performance characteristics, are quickly becoming mainstream. To operate efficiently, heterogeneous memory architectures require new data management solutions that are able to match the needs of each application with an appropriate type of memory. As the primary generators of memory usage, applications create a great deal of information that can be useful for guiding memory management, but the community still lacks tools to collect, organize, and leverage this information effectively. To address this gap, this work introduces a novel software framework that collects and analyzes *object-level* information to guide memory tiering. The framework includes tools to monitor the capacity and usage of individual data objects, routines that aggregate and convert this information into tier recommendations for the host platform, and mechanisms to enforce these recommendations according to user-selected policies. Moreover, the developed tools and techniques are fully automatic, work on standard Linux systems, and do not require modification or recompilation of existing software. Using this framework, this study evaluates and compares the impact of a variety of design choices for memory tiering, including different policies for prioritizes objects for fast memory tier as well as the frequency and timing of migration events. The results, collected on a modern Intel® platform with conventional DDR4 SDRAM as well as Intel Optane NVRAM, show that guiding data tiering with object-level information can enable significant performance and efficiency benefits compared to standard hardware- and software-directed data tiering strategies for a diverse set of memory-intensive workloads.

CCS Concepts: • **Software and its engineering** → *Runtime environments*; • **Computer systems organization** → *Heterogeneous (hybrid) systems.*

Additional Key Words and Phrases: heterogeneous memory systems, profiling, runtime systems, memory management, NVM

ACM Reference Format:
Brandon Kammerdiener, J. Zach McMichael, Michael R. Jantz, Kshitij A. Doshi, and Terry Jones. 2024. Flexible and Effective Object Tiering for Heterogeneous Memory Systems. 1, 1 (December 2024), 23 pages.

[*]**Extension of Conference Paper:** This work extends our earlier conference submission, titled *Flexible and Effective Object Tiering for Heterogeneous Memory Systems* and published in the ACM International Symposium on Memory Management [11]. New contributions include: (a) we investigate the potential of combining our original *reactive* object tiering policies with an approach that *proactively* assigns new allocations to a specific memory tier based on the call path context from which they are allocated, (b) we extend our original framework with new features for scenarios with multiple applications executing concurrently on the same platform, including an interface that allows applications to specify priority levels for high performance memories and a new policy that interprets and enforces application priorities, and (c) we evaluate the newly developed features using the same experimental setup as our original study and find that i) proactive data placement can significantly improve performance for certain workloads, and ii) leveraging application priorities can significantly increase the utilization of fast memory resources and enable higher throughput in multi-application scenarios.

Authors' addresses: Brandon Kammerdiener, bkammerd@vols.utk.edu, University of Tennessee, Knoxville, TN, USA; J. Zach McMichael, jmcmicha@vols.utk.edu, University of Tennessee, Knoxville, TN, USA; Michael R. Jantz, mrjantz@utk.edu, University of Tennessee, Knoxville, TN, USA; Kshitij A. Doshi, kshitij.a.doshi@intel.com, Intel Corporation, Chandler, AZ, USA; Terry Jones, trjones@ornl.gov, Oak Ridge National Laboratory, Oak Ridge, TN, USA.

1 INTRODUCTION

In recent years, multiple computing trends, including: the proliferation of AI and other data-driven analyses, multi-tenant backends, rising CPU core counts, and the relative stagnation of DRAM scaling, have combined to place enormous strain on memory systems. At the same time, several new media technologies (e.g., high-bandwidth memory [HBM] and non-volatile memory [NVM]), as well as new memory interconnect options (e.g., Compute Express Link), are bringing new capabilities that can potentially address the limitations of conventional memory hardware. As a result, many computing systems are now adopting a heterogeneous mix of memory devices and organizations, with the hope that the unique benefits and capabilities of the different technologies can be seamlessly combined into a single architecture.

Despite their potential benefits, heterogeneous memories present significant challenges for data management. Memory has traditionally been viewed as a homogeneous resource, sometimes divided into separate non-uniform memory access (NUMA) domains, but composed of devices with similar performance and capabilities. As a result, most modern operating systems (OSes) allocate and distribute physical memory with little or no knowledge of how applications intend to use these resources. However, to manage heterogeneous memories efficiently, the OS must be able to match allocation requests to the appropriate technology in consideration of both application requirements and hardware capabilities.

To address this problem, many recent projects have proposed software frameworks and tools to guide data management in hybrid memory environments [1, 5, 13, 15, 21–23, 25, 28–30]. Although these efforts have shown that guided data tiering can be effective, they each have constraints that limit their impact in certain scenarios. For example, some approaches profile memory usage and conduct tiering at the system level using architectural divisions, such as pages. While these approaches do not require coordination with application software, they lack insight into the logical structure of applications and their data and thus may make decisions that are at odds with application intentions.

Others have developed approaches that integrate profiling and tiering capabilities directly with application software, thereby providing tighter coordination between application behavior and system-level data management. These approaches often enable better understanding of memory usage because they associate memory profiling information with logical data features, such as object structures or allocation contexts. In many cases, profiling information associated with program features can be used to predict memory usage for classes or sets of data with similar features, even if these data have not been directly profiled. However, these approaches often require additional steps, such as a separate (profiled) program execution with representative input and/or source code modifications, that impede their usage for many applications. Moreover, these approaches are typically implemented by linking or compiling new code directly into a single process runtime, and thus, do not support multi-process workloads. Section 2.1 provides further details on the advantages and disadvantages of these prior efforts.

This work proposes a new software framework that aims to provide application-guided data tiering without the constraints and shortcomings of existing solutions. Our approach employs a custom memory allocator, known as *bkmalloc*, and a system-wide profiling and management tool called the *Memory Auto Tiering (MAT) Daemon*, which together enable automated tiering of program objects. bkmalloc allocates new objects into page-aligned regions of virtual memory and shares the address and size of each object with the running daemon. In turn, MAT Daemon profiles the memory usage of each known data object and then uses this information to create and enforce tier recommendations across the platform.

The proposed approach is fully automatic and does not require offline profiling, modifications, or recompilation of system or application software. Our Linux-based implementation employs standard system facilities, including perf [18] and Linux system calls, to profile and move program data from a user-level daemon. In this work, we demonstrate the flexibility and effectiveness of this approach by using it to direct data tiering for a variety of memory-intensive workloads on a modern Intel® platform with two tiers of memory: DDR4 SDRAM and non-volatile Optane DC RAM.

This work makes the following important contributions:

(1) We design and develop a novel software framework for guiding data object tiering on heterogeneous memory platforms. The proposed approach provides fast and flexible data tiering for single or multi process workloads without the need for offline profiling or recompilation of target applications. Our entire framework, complete with documentation, building, and testing scripts, is available open source in our public repository [10].
(2) We evaluate the effectiveness of a range of object tiering policies and parameters, including multiple strategies for prioritizing program data as well as novel approaches for deciding if and when to migrate program data between memory tiers. Overall, we find that automated object tiering can significantly improve performance compared to unguided approaches, but there is no single policy or set of parameter choices that produces the best performance for all workloads.
(3) Using a set hf memory intensive applications, we compare the best object tiering policies to popular alternative approaches, including hardware-directed caching and profile guided paging in the OS. While the hardware-based approach performs best on average, we find that object tiering can achieve similar or (up to 6%) better performance than caching in some cases, with much less data movement between tiers and without sacrificing the capacity of fast memory.
(4) We extend our framework with new features that: a) proactively assign program objects to a specific memory tier based on the context in which they were allocated, and b) interpret and enforce application priorities during object tiering. We show that these features increase fast-tier utilization significantly and deliver substantial speedups in both single- and multi-tenant scenarios without obliging applications to be hardware aware.

2 BACKGROUND AND RELATED WORK

Researchers and engineers across the computing community have proposed a variety of tools and techniques to manage data efficiently on heterogeneous memory platforms. One common strategy is to exercise the faster, smaller capacity tier(s) as a hardware-managed cache. For example, Intel®'s Cascade Lake processors include a "Memory Mode" option, which applies this approach with DDR4 as a direct-mapped cache to non-volatile Optane^{TM} DC memory [9, 16]. Although hardware-managed caching provides some immediate advantages, such as software-transparency and backwards compatibility, it is inflexible, often less efficient, and reduces the system's available capacity.

The alternative strategy of *software-directed* data tiering uses either the OS itself, or the OS in conjunction with applications, to assign data into different memory tiers with facilities to migrate data between tiers as needed. Implementations of this approach are often similar to data management on NUMA platforms [14], with each tier represented as its own NUMA domain. In many cases, the OS will also expose data tiering controls to user programs through the system call interface. For example, Linux applications can use the `mbind` or `move_pages` system calls to request or require that a specific range of virtual memory be backed with physical pages from a particular memory tier. These finer-grained controls allow applications to coordinate tier assignments with allocation and usage patterns, potentially enabling powerful efficiencies. However, because this approach requires expert knowledge and modifications to program source, its usage is still limited.

Table 1. Features of our approach (bkmalloc + MAT Daemon) compared with other software-directed data tiering approaches.

Approach	No offline profiling	No modification, recompilation of user applications	No OS modifications	Supports tiering of logical program units (objects, data structures)	Allows user-level software to define tiering policy	Supports multiple processes
XMem [5], Servat [28], RTHMS [23], Unimem [30], MemBrain [22], Laghari [15], Akram [2]	×	×	✓	✓	✓	×
Thermostat [1], Kim et al. [13], Choi et al. [3], Linux tiering patches [29], MTM [26]	✓	✓	×	×	×	✓
HeMem [25]	✓	✓	×	×	✓	×
SICM + MemBrain [21]	✓	×	✓	✓	✓	×
bkmalloc + MAT Daemon	✓	✓	✓	✓	✓	✓

2.1 Automated Approaches for Guiding Data Tiering

Several recent efforts have proposed software tools and techniques that seek to address the limitations of these existing solutions by automating all or part of the data classification and migration processes. The present work has a similar goal, but it also has some distinct advantages over these prior efforts. Table 1 summarizes how our approach compares with other proposed efforts in terms of features that affect the transparency, flexibility, and efficacy of each approach.

Some previous studies developed tools to profile the usage of certain data structures offline, and then use heuristic models to assign data objects to the appropriate tier [2, 5, 15, 22, 23, 28, 30]. Although these efforts facilitate the production of high-quality tiering guidance, they still require manual updates to application code and/or recompilation of the target program to attach recommendations to program data. In contrast, our approach generates and enforces tier recommendations online in a concurrent process, without requiring any updates to existing applications.

Another set of efforts employed architectural profiling of physical memory regions (often pages) to assist data tiering in the OS [1, 3, 13, 29]. Although these approaches are completely transparent to user-level software, they also conduct data tiering with no knowledge of the logical structure of program data. Additionally, due to their dependence on system-level management, they are less flexible and cannot customize data management for individual applications. On the contrary, our approach monitors the usage of individual data objects and uses this information to build better profiles of memory behavior during execution. Moreover, it allows applications to implement their own feedback-directed data tiering policies in a modular user-level framework.

Some recent efforts have proposed tools and frameworks that address some of the drawbacks of these earlier approaches. Olson et al. extended MemBrain with functions to analyze memory usage and direct data tiering *online* and without the need for offline profiling. However, applications that use it still need to be recompiled [21]. The HeMem project provides facilities to define data tiering policies in user-level software, but it does not provide any mechanism for these policies to know or exploit the logical structure of application data [25]. Moreover, both SICM+MemBrain and HeMem enable guided data tiering by linking their capabilities into a single target process. Hence, these approaches

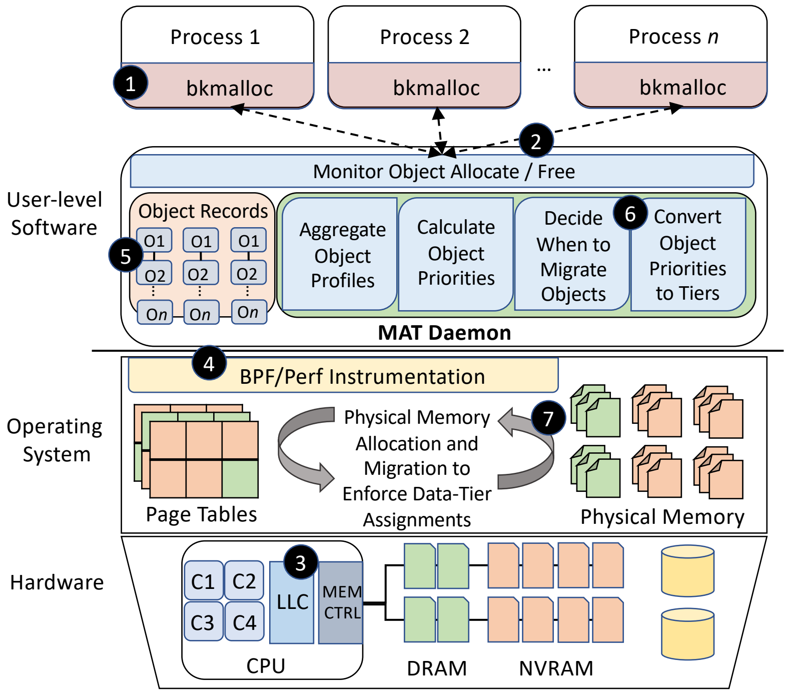
Process 1
bkmalloc
Process 2
bkmalloc
Process n
bkmalloc
1
2
User-level Software
Monitor Object Allocate / Free
Object Records
O1
O2
On
5
Aggregate Object Profiles
Calculate Object Priorities
Decide When to Migrate Objects
6
Convert Object Priorities to Tiers
MAT Daemon
Operating System
4
BPF/Perf Instrumentation
Page Tables
Physical Memory Allocation and Migration to Enforce Data-Tier Assignments
7
Physical Memory
Hardware
C1
C2
C3
C4
3
LLC
MEM CTRL
CPU
DRAM
NVRAM

Fig. 1. Design overview of our approach.

can only support one process at a time. By combining allocator instrumentation with a separate daemon process, our approach is the first to support transparent and flexible data object tiering for both single- and multi-process workloads.

3 FLEXIBLE AND EFFECTIVE OBJECT TIERING FOR HETEROGENEOUS MEMORY SYSTEMS

3.1 Design Overview

Figure 1 depicts the main components of our approach, which primarily consists of two new pieces of software:

(1) The bkmalloc allocator provides two essential capabilities: (1) page-aligned allocations for objects larger than a certain size and (2) instrumentation that records when an application allocates or frees an object. It is also capable of dictating the tier to which an object is initially allocated through standard operating system calls.
(2) The MAT Daemon runs alongside the applications and conducts object tiering through a series of complementary activities, including (1) monitoring and structuring profiles of object allocation and usage, (2) automated heuristics to prioritize objects for placement in fast memory, and (3) mechanisms to enforce tier recommendations when a particular event occurs.

3.2 Tracking Object Creation and Removal

On initialization, applications that will participate in guided object tiering dynamically link the bkmalloc allocator (marked with 1 in Figure 1).[1] bkmalloc is a general-purpose malloc implementation with capabilities similar to other

[1]In our Linux-based implementation, bkmalloc is built and distributed as a shared library (.so) file. Applications use the LD_PRELOAD environment variable to load bkmalloc in place of the standard system allocator.

modern allocators, such as the GNU allocator or jemalloc, but it includes additional features to support object tiering with MAT Daemon. When the application requests an allocation larger than a certain size, bkmalloc allocates the object in a page-aligned region of virtual memory and notifies MAT Daemon of the new object's address and size. Similarly, bkmalloc will send a corresponding notification whenever the application frees an object from its heap.

In our current implementation, the size threshold for individually tracked data objects is 4 KB, which matches the page size on our platform. Objects smaller than 4 KB are slot allocated into a set of buckets composed of page-aligned blocks, in which the size of each block depends on the size of the allocation. bkmalloc also notifies MAT Daemon of each active block, allowing the framework to manage data associated with smaller program objects in block-size units.

MAT Daemon receives object notifications through a first-in-first-out (FIFO) message queue ❷. To minimize the execution time of this communication, insertions into the queue are non-blocking except in the rare event that the queue is full. Additionally, to keep the latency of these operations as low as possible, the daemon polls the queue continuously from a separate thread, which is ideally pinned to its own, otherwise unused, computing core.

3.2.1 Handling Ephemeral Objects. For many applications, a significant portion (in some cases, the vast majority) of object allocations have very short lives and are quickly replaced in the address space by new allocations. Such ephemeral objects are not very consequential for data tiering because they typically consume a relatively small portion of capacity, which is often captured in processor caches. However, record keeping for ephemeral objects can be problematic for our approach because it can delay the collection of profiles for longer-lived objects with more impactful tiering consequences. Thus, rather than create and destroy object records immediately as new allocations are seen, our approach buffers the allocation events until the end of the current profiling interval. If MAT Daemon then receives a de-allocation event that corresponds to a buffered allocation event before the end of the interval, it simply removes that allocation from its buffer. In this way, our approach avoids record keeping for most ephemeral program objects.

3.2.2 Split Records for Large Objects. During our initial testing, we found that some applications allocate a small number of very large objects that comprise much or most of their overall memory capacity. To enable more precise monitoring and more fine-grained control over the placement of data within such large allocations, MAT Daemon provides the option to split objects that are larger than a certain size into multiple records. Specifically, this option splits large allocations into $\lceil n/s \rceil$ records, where n is the size of the original allocation, and s is the given split size. All split records, except possibly the last record by address order, cover a range of addresses equal to the split size.

3.3 Monitoring the Memory Usage of Active Objects

To understand how objects are using memory, our approach efficiently monitors memory usage events through architectural sampling and system-level instrumentation. Specifically, it integrates MAT Daemon with two facilities that are commonly available on modern Linux distributions: (1) the perf subsystem for architectural sampling [18] and (2) BPF [7], which is a compiler framework and tool set that allow user programs to insert custom instrumentation into a running kernel safely and dynamically without modifying kernel source code. Upon initialization, MAT Daemon configures perf to begin sampling memory reads that result in a miss in the processor's last level cache (LLC) ❸. It then invokes BPF to install system-level instrumentation that intercepts the LLC miss events generated by perf and then transmits them to MAT Daemon through a shared ring buffer ❹.

Next, as MAT Daemon receives allocation events from the attached applications, it creates records for the active heap objects ❺. Each record contains the starting virtual address and size of each object as well as a set of fields that can be used to aggregate and store profile information related to the usage of the object in memory. In our current implementation, these fields record the creation time of the object, the time corresponding to the most recent sampled LLC miss, as well as the number of LLC misses scaled and normalized over a configurable time interval.

3.3.1 Unified System and Application Guidance. One of the key advantages of our design is that it allows MAT Daemon to monitor and leverage allocation behavior as well as system-level events to direct memory tiering. Moreover, although our current implementation only uses BPF to record samples of LLC miss events, this approach could easily be extended to collect and incorporate other types of system-level guidance to further enhance data tiering. Indeed, earlier iterations of this work used BPF to monitor page fault and page release events to track the physical memory capacity of every data object. However, we disabled this feature for our evaluation because we found that the additional instrumentation can cause significant execution time overheads (up to 10% in our testing) without any discernible benefit compared with simply using the allocated size of each object as an estimate of its capacity utilization.

Similarly, BPF instrumentation could also be used to monitor the memory usage of kernel objects (e.g., file and network buffers) that are not mapped into any user process. Recent work has shown that kernel data usage is a significant and important factor in data tiering for some applications [12]. Although support for this feature is outside the scope of this study, we plan to investigate policies that can monitor and control the placement of kernel and application data simultaneously in future work.

3.4 Profile-Guided Object Tiering with MAT Daemon

To support profile-guided object tiering, MAT Daemon spawns a separate thread that operates on a timer-driven signal. At each timer tick, this thread invokes a set of routines to (1) aggregate the most recent profile information with information collected during prior intervals (aggregation routine) (2) calculate object priorities for placement in the smaller, faster memory (prioritization routine) (3) determine whether or not to enforce new tier assignments for the active objects at this time (trigger routine), and, if necessary, (4) convert the object priorities into object-tier assignments and migrate certain objects to enforce the new tier assignments (enforcement routine) ❻.

Because our design is modular, users can create custom tiering policies by selecting from a set of existing routines or by providing their own implementations of each of the above functionalities. For this work, we use this modular design to evaluate and compare the effectiveness of a variety of choices and policies for object tiering. Next, we describe the set of routines we implemented for this study.

3.4.1 Profile Aggregation. The goal of the aggregation routine is to combine profile information from the most recent interval with information collected during prior intervals. For some statistics, such as the address, size, creation time, and most recent access time, aggregation across intervals is unnecessary. Estimating the relative access rates of each object is more complicated because the frequency with which a program accesses a particular data object may shift substantially over time. Our approach computes a weighted average of the number of sampled accesses for each object during the most recent interval with the average from earlier intervals. Specifically, we use the following formula:

$$\tau_{n+1} = \alpha * t_n + (1 - \alpha) * \tau_n, \tag{1}$$

where τ_{n+1} is the predicted number of sampled LLC misses for the object for the next program interval, t_n is the actual number of sampled LLC misses for the object during interval n, and α is a parameter ranging from 0.0 to 1.0. For this work, we selected an α value of 0.1 after a brief tuning process, which is described in Section 4.5.1.

3.4.2 Calculating Object Priorities. The prioritization routine computes an ordering of all active objects that describes their priority for placement within the fast memory tier. Our study evaluates three prioritization routines:

(1) **First-in-first-out (FIFO)**: Priority is based on the age of the objects. Older objects have lower priority.
(2) **Least recently used (LRU)**: Priority is based on time since last access. The LRU objects have lower priority.
(3) **Accesses per byte (APB)**: Priority is based on the weighted average of accesses per byte of capacity. Objects with lower APB have lower priority.

3.4.3 Deciding When to Migrate Data Objects. Similar to other software-directed tiering approaches, our approach migrates program data from one tier to another by remapping virtual memory to a different set of physical pages. Unfortunately, this is often an expensive operation on modern computing systems. Before moving any program data, the OS must interrupt and suspend application threads to prevent inconsistencies caused by data races. There are additional costs for actually copying data from one tier to another as well as keeping page tables (and TLBs) synchronized with upper-level software. Thus, while object priorities may shift from interval to interval, frequently migrating program data to match these priorities can be counterproductive.

To control migration costs, MAT Daemon employs a trigger routine to decide when it should migrate program data to match object priorities. Our study evaluates three approaches for triggering the reassignment of objects to tiers:

(1) **Time Trigger**: Reassign every *n* timer ticks.
(2) **LLC Misses Per Instruction (LLCMPI) Trigger**: Reassign when the difference between the LLCMPI of the most recent interval and the average LLCMPI of the previous ten intervals, and normalized by the average LLCMPI, exceeds a certain threshold.
(3) **Allocation Trigger**: Reassign when *n* bytes have been allocated since the previous re-assignment.

Thus, the Time Trigger enables direct control over the rate of migration by limiting the number of migration events over time. In contrast, the LLCMPI and Allocation Triggers aim to detect events that indicate application memory usage is changing and only allow data migration to occur after shifting resource utilization has become evident.

3.4.4 Converting Priorities to Tier Assignments: When the trigger routine indicates that it is ready to migrate program data, MAT Daemon will invoke the enforcement routine to create and enforce new object-tier assignments. To implement this routine on our Intel® Linux platform, we configure the OS to view each tier of memory as a distinct (memory-only) NUMA node in a single physical address space.[2] This way, user software can use the standard NUMA API and related system calls to assign and remap virtual memory to a specific type of memory ❼.

The enforcement routine then partitions the active objects into different sets that correspond to each memory tier. Specifically, it traverses the live objects in the order of their priorities and adds objects to the fast memory set until the aggregate size of these objects is greater than the capacity limit of the fast tier. All the remaining objects are then assigned to the slow memory set. To move program data, the daemon invokes the `move_pages` system call to (1) demote objects that are currently in the fast tier but belong in the slow memory set and then (2) promote objects that are currently in the slow tier but belong in the fast memory set.

3.5 Proactive Object Tier Assignments with Allocation Sites

As with any profile-guided memory tiering approach, our approach has significant costs associated with: 1) executing in a suboptimal tiering configuration as the runtime collects memory usage information, and 2) migrating program

[2]Additional details and an example of this procedure are available at [27].

data if and when the runtime determines it should be reassigned to a new tier. To limit these costs, we have extended our reactive tiering framework with features that *proactively* assign certain data objects to a specific tier based on the context from which the data are allocated. Our approach relies on the observation, described and validated in prior works [6, 21, 22, 28], that data allocated from the same allocation site tend to have similar usage patterns. In this context, an *allocation site* refers to the address of the allocating instruction (e.g., `malloc` or `new`) with some amount of call path context. By aggregating the profiles of objects originating from the same site, the runtime may be able to predict whether an object should be assigned to the fast or slow memory *at the time it is allocated* by only knowing the site from which it originates.

To implement this approach, we extended bkmalloc to record the call path context of every allocation larger than the object tracking threshold of 4 KB. Specifically, we use the Linux `backtrace` facility [19] to record up to 16 addresses of call path context leading to the allocation instruction. Next, the allocator hashes these addresses together into a 64-bit site ID using a simple multiplicative hash and then sends the site ID along with the new object notification message to the MAT Daemon. While our chosen hash procedure theoretically permits collisions of distinct call path contexts, such collisions are very unlikely to occur, and we have confirmed that there are no collisions among the allocation contexts reached in our evaluation workloads.

The MAT Daemon proceeds as described before, profiling and periodically assigning active objects to a specific memory tier based on the current object tiering policy. However, it may now assign each new object directly to a certain tier based on the tier assignments of objects created at the same site. Specifically, if the proportion of a site's objects that are assigned to the same tier exceeds a certain threshold, then the daemon will send a message back to the allocator indicating that new objects from this site should be proactively assigned to that same tier. Given this information, the allocator can then use standard Linux facilities (e.g., `mmap` and `mbind`) to assign new allocations from that site into pages corresponding to the appropriate memory tier.

4 EXPERIMENTAL SETUP

4.1 Platform Details

Our evaluation platform contains a single Intel® Xeon® Gold 6246R CPU (code named Cascade Lake or CLX) with 16 physical compute cores hyperthreaded to 32 logical cores. The cores all run a 3.4 GHz clock and share a 35.75 MB L3 cache. The processor includes a memory controller that services requests to both DDR4 SDRAM as well as Optane™ DC persistent memory through a common memory bus. The bus is divided into six identical channels, each of which is connected to one 32 GB, 2933 MT/s, DDR4 DIMM and one 128 GB, 2666 MT/s, Optane™ DC module. Thus, the system contains a total of 192 GB of DDR4 SDRAM and 768 GB of Optane™ DC persistent memory. Data reads from the non-volatile memory require 2× to 3× longer latencies, and sustain only 30%–40% of the bandwidth of the DDR4 memory. Although latency for writes is similar for both types of memory, the DDR4 also supports 5×–10× more write bandwidth than the Optane™ memory [9]. All experiments use Debian 11 with Linux as the base operating system.

4.2 Workloads

Our evaluation employs a variety of applications that we selected based on their potential to stress cache and memory performance on our platform. Table 2 describes our selected benchmarks with the mechanism that is used to parallelize each workload. Several of these applications, namely LULESH, AMG, SNAP, and QMCPACK, come from the CORAL [17] suite, which is a set of high-performance computing applications developed and maintained by the US Department

Table 2. Benchmark descriptions with parallelization method.

App	Description	Parallel
LULESH	Hydrodynamics stencil calculation, very little communication between computational units.	OpenMP
AMG	Parallel algebraic multigrid solver for linear systems on unstructured grids.	OpenMP
SNAP	Mimics the computational needs of PARTISN, a Boltzmann transport equation solver.	OpenMP
QMCPACK	Quantum Monte Carlo simulation of the electronic structure of atoms, molecules.	OpenMP
WarpX	Highly optimized and parallelized advanced electromagnetic Particle-In-Cell computation.	OpenMP
IdenProf	DenseNet-121 (dense neural network) training for a large set of images of professionals.	Thread Pool
Graph500	Search and find the shortest path to a set of random keys in a large undirected graph.	MPI

Table 3. Benchmark inputs with default performance. For each input the columns show the number of software threads, the benchmark arguments, and the execution time, peak resident set size (GB), and average memory bandwidth (GB/s) of the default configuration (system allocator with DRAM-preferred first-touch policy on an otherwise idle machine with 32 logical cores).

App	Input	Threads	Input Arguments	Time (s)	GB	GB/s
LULESH	small	32	`-s 300 -i 12 -r 11 -b 0 -c 64 -p`	307.7	23.17	87.68
	full	32	`-s 850 -i 3 -r 11 -b 0 -c 64 -p`	16,205	573.21	8.47
	shared	16	`-s 500 -i 3 -r 11 -b 0 -c 64 -p`	461.9	119.30	75.1
AMG	small	32	`-problem 2 -n 300 300 300`	297.86	46.41	84.60
	full	32	`-problem 2 -n 700 700 700`	9,326	587.39	33.6
	shared	16	`-problem 2 -n 400 400 400`	787.11	104.39	77.62
SNAP	small	32	`nx=320, ny=60, nz=46`	297.20	27.18	42.05
	full	32	`nx=5120, ny=60, nz=46`	14,757	424.98	15.96
	shared	16	`nx=512, ny=60, nz=46`	957.26	43.28	25.3
QMCPACK	small	32	NiO S64, VMC method, 40 walkers	256.63	19.40	20.99
	full	32	NiO S256, VMC method, 40 walkers	12,948	353.22	21.24
	shared	16	NiO S128, VMC method, 40 walkers	539.63	51.29	24.53
WarpX	small	32	`max_step=20, n_cell=256 256 4096`	289.56	50.23	29.59
	full	32	`max_step=20, n_cell=1024 1024 4096`	15,267	425.38	5.44
	shared	16	`max_step=20, n_cell=512 512 4096`	1,347.31	111.02	24.01
IdenProf	small	32	`num_objects=10 batch_size=256`	740.84	37.30	36.85
	full	32	`num_objects=10 batch_size=2575`	1,448	306.68	9.62
	shared	16	`num_objects=10 batch_size=512`	905.1	66.09	29.97
Graph500[3]	small	32	`graph500_reference_bss, scale=24`	258.88	9.10	44.26
	shared	16	`graph500_reference_bss, scale=27`	2,880.02	52.84	21.26

of Energy. The other applications represent important and widely used computations from machine learning, graph search, and scientific computing domains.

Table 3 presents the inputs we use to run each benchmark and some basic performance characteristics with our default configuration. We evaluated our approach with three input sizes called *small*, *full*, and *shared*. The full inputs are designed to take advantage of the large capacity NVM on our platform and thus generate data outlays beyond

[3]We omit the full input size for Graph500 because Graph500 inputs that are large enough to generate larger than DRAM memory capacities always failed with a segmentation fault on our platform, even in the default configuration.

the available DRAM. While the full size workloads generate high rates of memory access, their observed bandwidths are relatively low because a substantial portion of their hot data are on the slower memory devices in the default configuration. Moreover, they often require several hours of execution time for each experimental run. Hence, for faster evaluation and for comparisons with an ideal everything-in-DRAM configuration, we also constructed a smaller inputs that are able to complete their execution within a few minutes and fit entirely within the DRAM tier. However, these "small" inputs still require several to dozens of GBs of memory capacity and generate memory bandwidth well beyond what is sustainable by the NVM tier. To model scenarios where multiple applications share memory resources, we also constructed the shared inputs. These inputs use only half the number of software threads as there are logical cores on our platform and have memory outlays that fit entirely within the DRAM tier. In this way, the shared inputs can be configured to execute concurrently with another application without sharing computing resources, but may still contend for fast memory resources if the combined capacity of the concurrent tasks exceeds that of the upper tier.

4.3 Operating System Details

Experiments with the small input sizes were run with Linux kernel version 5.14. To control and generate contention for capacity in the fast memory tier with relatively small workloads, we extended this kernel with facilities to constrain the amount of DRAM available to each application. Specifically, we added an option to the memory control group (cgroups) interface to limit the amount of physical memory that processes in the group can allocate and keep resident in fast memory at any given point in time. Thus, if a process requests a page from the faster tier when the specified limit has been reached, the kernel will supply a page from the slower memory tier or fallback to page reclaim if no other memory is available. In addition to allowing relatively fast evaluation of many tiering configurations, this approach also enables easy and direct control over the amount of fast memory available as well as comparison with an ideal configuration in which the size of the faster tier is unconstrained.

To test our approach against alternative tiering strategies at full system scale, we deployed and ran the full size inputs on Linux kernel version 5.15 with Intel®'s tiering patches [29] patches applied. While our approach does not require any functionalities included with these patches, the results with the full inputs are affected by some page migration optimizations included in this patch. Specifically, the patch significantly improves page migration throughput by batching certain operations (e.g., TLB shootdowns) when migrating application pages from one tier to another.[4] Since we expect these optimizations will likely be included in future kernels that support software-based tiering, we left them enabled for runs with our own approach.

4.4 Compilation and Parallelization

Aside from IdenProf, which relies on several Python-based libraries, including TensorFlow and numpy, all of the selected benchmarks are written entirely in popular, statically compiled languages, such as C, C++, and Fortran. We compiled these static language applications and the static components of IdenProf by using the GNU Compiler Collection v.10.2.1 with default optimization settings. We execute IdenProf using the standard CPython implementation of Python v.3.7.6.

Each benchmark provides a method to parallelize its execution over multiple concurrent threads or processes. LULESH, AMG, SNAP, QMCPACK, and WarpX employ OpenMP to distribute their workloads across multiple concurrent threads. Graph500 achieves a similar effect with multiple separate processes by using MPI. To enable these features, we linked

[4]In experiments with the LULESH workload, the page migration throughput improved by about 2.8× with these optimizations enabled.

each application with the standard OpenMP (v.4.5) or MPI runtime (MPICH v.3.4.1-4) during compilation. IdenProf executes as a single process but uses a pool of threads to parallelize the DenseNet-121 training procedure.

Our tiering experiments use `numactl` to restrict the application to 30 logical cores and reserve the remaining two cores (which correspond to the same physical core) for the MAT Daemon. In comparison to a configuration that creates 32 threads for the application alone, we found that using two fewer threads to avoid scheduling conflicts with MAT Daemon enabled better overall performance for our benchmarks. For all benchmarks except IdenProf, which automatically adjusts the number of threads in its thread pool based on the available computing resources, we also employ the relevant OpenMP or MPI options to set the number of threads to match the number of available cores.

4.5 Common Experimental Configuration

For each experimental configuration, we report overhead and performance results as the mean execution time of five experimental runs relative to the default configuration. To estimate and report variability in these results, we also compute the 95% confidence intervals for the difference between the means of the experimental and default configurations, as described in Georges et al. [8]. These intervals are plotted as error bars around the sample means in the relevant figures. While we have computed and plotted these error bars for all of our results, variability is too low for the bars to be visible in some cases. To reduce sources of variability between runs, all experiments execute in isolation on an otherwise idle machine. An automated script also clears out the page cache and disables transparent huge pages for the application process prior to each experimental run.

4.5.1 Common Profiling and Tiering Parameters. To monitor memory access behavior, MAT Daemon configures perf to sample the `MEM_LOAD_L3_MISS_RETIRED.LOCAL_DRAM` and `MEM_LOAD_UOPS_RETIRED.LOCAL_PMM` events with a sampling period of 4,096. We selected this sampling period because we found that it is sufficient to generate tens of thousands of samples per second with only negligible ($< 1\%$) execution time costs for our workloads. The LLCMPI trigger policy also uses perf to compute the number of LLC misses per instruction as the ratio of the `MEM_LOAD_RETIRED.L3_MISS` and `INST_RETIRED.ALL` hardware counters.

We experimented with several timer interval lengths for operating the profile collection and aggregation capabilities in MAT Daemon, including 0.1, 0.2, 1.0, 2.0, and 10 seconds. We found no difference in the execution time overhead of the concurrent workload with each interval. However, values less than 2 seconds produced more irregular profiles because the aggregation and analysis procedures often required longer than the profile period. Hence, all of our profiling and profile-guided configurations operate the timer-driven thread in MAT Daemon with a 2-second interval.

Using this 2-second interval, we also tried a number of α values for aggregating the profiles of object access counts (see Equation 1 in Section 3.4.1), including 0.01, 0.1, 0.5, 0.9, and 1.0. We found that 0.1 provided relatively high accuracy when predicting usage in the next program interval while still being responsive to shifts in program behavior. Thus, experiments that use this capability always set α to 0.1.

We configured MAT Daemon to always split objects larger than 64 MB into multiple records (Section 3.2.2). In early testing, we found that 64 MB is small enough to enable sufficient control over the placement of data within large objects without increasing the execution time overhead of profiling. Lastly, to leave a portion of fast memory for future allocations, each tier reassignment only fills the fast memory up to 90% of its available capacity. In preliminary testing, we found this approach produced better overall performance than configurations that use 50%, 80%, or 100% of the available DRAM at each migration event.

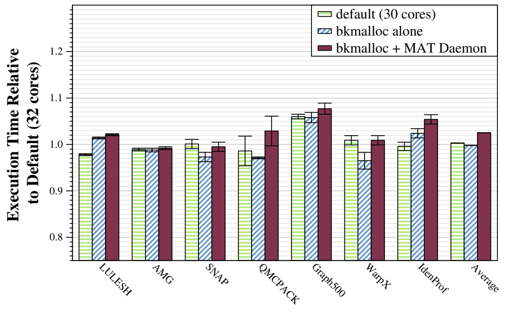

Fig. 2. Execution time overhead of object profiling relative to default (32-cores, system allocator) (lower is better).

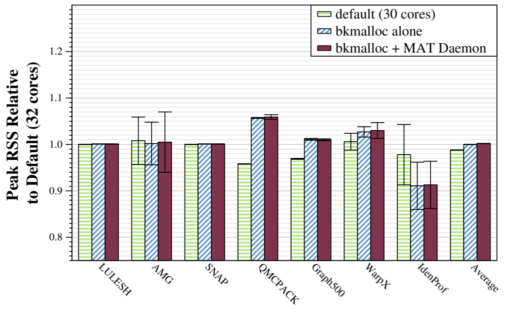

Fig. 3. Memory capacity overhead of object profiling relative to default (32-cores, system allocator) (lower is better).

5 EVALUATION

5.1 Online Profiling Overhead

Let us first consider the execution time and memory overhead of object profiling with bkmalloc and MAT Daemon. For this evaluation, we compare execution with the standard malloc implementation from glibc (v.2.3.1) against execution with our custom allocator and profiling tools. To avoid performance effects related to data tiering, the experiments in this subsection only use the small input sizes and assign all program data to the DRAM tier. They do not include any of the object tiering or migration capabilities described in Section 3.4.

5.1.1 Execution Time Overhead. Figure 2 shows the execution time of each benchmark when run with bkmalloc alone and when run with bkmalloc and MAT Daemon collecting object usage statistics throughout the run. Since both *bkmalloc alone* and *bkmalloc + MAT Daemon* restrict the application to use only 30 logical cores (as discussed in Section 4.4), we also plot the execution time of the default system allocator with the application restricted to only 30 logical cores to isolate this effect. All results are relative to the execution time of a default (32 core) configuration with the standard system allocator (lower is better).

Thus, limiting the workload to only 30 logical cores has very little impact on the performance of these benchmarks. The worst case of Graph500 slows down by about 6%, but in some cases, performance slightly improves when computing resources are restricted, perhaps due to less communication overhead than the default configuration. Similarly, bkmalloc has negligible average impact but may have some impact on the performance of individual benchmarks. Since bkmalloc changes the layout of data in the heap, it may also affect the efficiency of processor caches. In some cases (e.g., LULESH and IdenProf), these effects do cause some slowdown, but in others (e.g., SNAP and WarpX), they actually improve performance compared to the default allocator. Across all seven benchmarks, these effects mostly cancel each other out and result in a slight ($< 1\%$) average improvement over the default allocator. Additionally, we find that adding object profiling with MAT Daemon has only a small additional cost compared to *bkmalloc alone.* In the worst case of QMCPACK, MAT Daemon adds an extra 5.8% of execution time overhead, whereas the average execution time cost of object profiling is only 2.8%.

5.1.2 Memory Capacity Overhead. Next, let us examine the memory capacity overhead of our approach. Figure 3 shows the peak memory usage of each workload with the *default (30 cores), bkmalloc alone*, and *bkmalloc + MAT Daemon* configurations relative to the default (32 core) configuration (lower is better). Similar to execution time, bkmalloc has a

Table 4. Object usage statistics: For each benchmark, the columns show the peak number of active object records, the number of object records created per second, the number of MBs allocated per second, the average lifetime of each tracked object (in seconds), and the portion of LLC miss events that are associated with some object during profiling.

Small inputs (30 cores)					
Application	**Peak OR**	**OR/sec**	**MB/sec**	**Life (s)**	**LLC**
LULESH	346	5.26	235.90	35.96	0.951
AMG	997	8.93	428.21	67.26	0.956
SNAP	628	2.12	93.17	290.11	0.997
QMCPACK	191,706	747.01	79.57	243.77	0.99
WarpX	37,130	475.87	577.00	25.30	0.678
Graph500	1,125	7.65	47.76	115.74	0.999
IdenProf	3,981	44.44	1,508.35	77.03	0.367

mixed impact on the memory capacity of our workloads. On average, these effects cancel each other out, and there is essentially no difference between the peak capacity utilization of the default and bkmalloc allocators. Additionally, the MAT Daemon uses a small amount of memory storage to create and maintain profiles of each active data object. Specifically, our current implementation requires only 136 bytes of memory for each object record and 152 bytes for each process record. As expected, we find that this additional storage has little to no impact on the peak capacity utilization of our benchmark set.

5.2 Object Usage Characteristics

Next, let us examine how our selected workloads allocate and use objects in memory. Since these experiments depend on execution rates and timing, we omit results for the shared and full inputs, which may be impacted by thread restrictions and tiering effects, and only present results for the small inputs with all objects assigned to DRAM.

Table 4 presents object statistics for each application with its small input. Note that, since these statistics only include *object records*, they do not include ephemeral objects, and allocations larger than 64 MB are split into multiple records (Sections 3.2.1 and 3.2.2). Thus, for these workloads, the peak number of object records ranges from only a few hundred to over 190,000. The peak number of records is relatively low because the vast majority of allocation events are for small and/or ephemeral objects, which are never converted to object records. For the same reason, the average lifetime of each record is relatively high, at least 21 seconds for all applications. This property is important because longer lifetimes enable our tools to collect more information about the usage of each object.

The column on the right shows the portion of memory read events (i.e., LLC misses) that are associated with an active object record during execution. Thus, for most benchmarks, the vast majority (> 95%) of sampled LLC misses correspond to an active object record. However, for WarpX and IdenProf, many of the sampled accesses correspond to addresses that are not part of any object record. This result indicates that a significant portion of their memory bandwidth corresponds to non-heap data (e.g., stack or kernel objects) or possibly ephemeral heap objects. Despite this result, we found that our approach can still be effective for these workloads because it identifies and moves a large proportion of *cold* application data to slower memory, thereby freeing up fast memory for untracked application data.

5.3 Comparison of Object Tiering Policies

Our next set of experiments aims to identify the best performing tiering policies for each workload under different capacity constraints. In order to compare a larger number of policies and configurations, these experiments only test the small inputs and use the custom control group interface (described in Section 4.2) to control the capacity of the fast memory tier. Specifically, these experiments limit the amount of DRAM for each workload to be 12.5%, 25%, and 50% of the same workload's peak capacity utilization in its default (32 core) configuration. Additionally, these experiments, along with the first set of full scale experiments in Section 5.4, only evaluate reactive object tiering, and do not employ the proactive approach described in Section 3.5.

5.3.1 Selection of Trigger Thresholds. Each of the strategies that our framework uses to decide when to migrate program data includes a threshold value that can affect performance in different scenarios. To limit the set of configurations in our comparisons, we conducted some preliminary experiments to identify reasonable threshold values for each approach. For these initial tests, we limited the capacity available in the DRAM tier to 25% of the application's peak utilization and configured MAT Daemon to use the APB object prioritization policy. We then executed each workload once with a range of options for each trigger policy and selected the value that provided the best performance for our full set of experiments. The range of options we evaluated with each trigger policy are shown below with the value we selected for our experiments emphasized and in bold.

- Time trigger (number of profiling intervals, each interval is 2 seconds): 1, 2, **5**, 10, and 100.
- LLCMPI trigger (normalized difference between LLCMPI of the current interval and average LLCMPI of prior intervals): 0.5, 1, **2**, 5, 10, 20, 30, 40, and 50.
- Allocation trigger (MBs allocated): 512; **1,024**; 2,048; and 4096.

5.3.2 Object Tiering Performance. Figure 4 presents the performance of each object prioritization policy with each migration trigger policy when the fast memory capacity is limited to 12.5%, 25%, and 50% of the application's peak capacity. For each workload and capacity limit, we also plot the performance of an unguided, *first touch* (FT) configuration. Similar to the standard Linux policy for non-uniform memories, FT simply faults data into the DRAM tier until it is full and then assigns new data to the NVM tier. Note also that, aside from the DRAM capacity limit, FT is identical to default (32 core) execution. All results are shown in execution time relative to the default configuration with all program data assigned to the DRAM tier (lower is better).

The results show that, for most workloads and capacity limits, there is at least one object tiering policy that significantly outperforms the unguided FT configuration. However, in a few cases (e.g., LULESH-50%, AMG-12.5%, and Graph500-50%), profile-guided object tiering provides no benefit or can even degrade performance vs. an unguided approach. On further analysis, we found that although our approach does produce slightly more efficient data-tier assignments for these cases, the additional profiling and migration costs imposed by our tools negate their benefits. Despite these cases, our approach still produces substantial performance gains on average compared to unguided FT. The best configurations overall use the allocation trigger with either the LRU or APB object prioritization policies and achieve speedups of 1.4×, 1.6×, and 1.35× over FT with the 12.5%, 25%, and 50% capacity limits, respectively.

Comparing the effectiveness of the tiering strategies allows us to make several additional observations:

(1) In general, the LRU and APB object prioritization strategies are more effective than FIFO. Intuitively, this result makes sense because past and recent usage of a particular data object are more likely to predict future usage than the age of the data. However, for workloads such as IdenProf, which creates new data more quickly than the other benchmarks, FIFO can be effective because it does not need to wait to build a usage profile to identify hot program data.

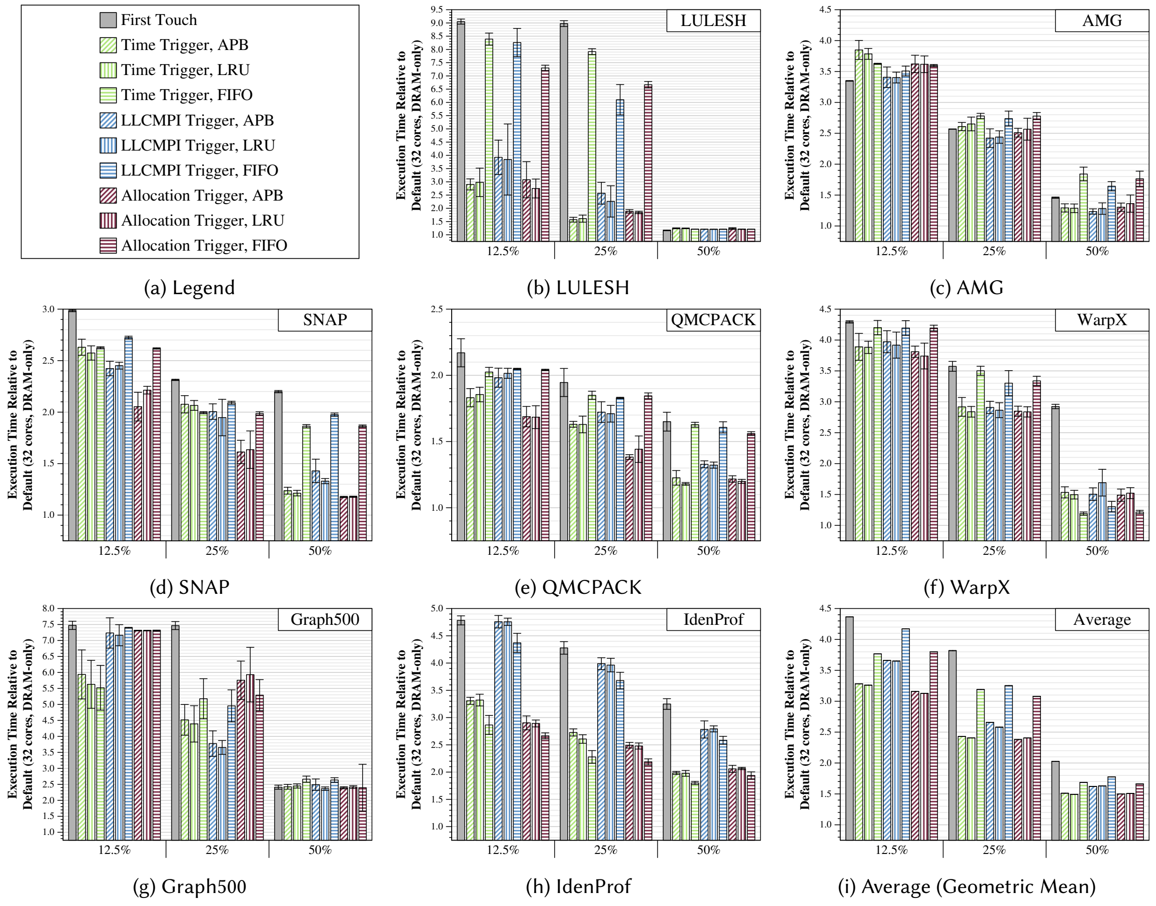

Fig. 4. Performance (execution time) of guided object tiering with varying amounts of capacity available in the faster DDR4 memory tier. All results are shown relative to the default (32 core) configuration with all program data allocated to the DDR4 tier (lower is better). The DDR4 capacities shown along the x axis are calculated as a percentage of the peak resident set size during execution with the default configuration.

(2) On average, the time and allocation trigger policies are more effective than the LLCMPI trigger. Although the LLCMPI policy can be effective when tuned for individual workloads, we found that each individual threshold value we tested was significantly worse than the best threshold for at least some benchmarks.
(3) Although the results show that the allocation trigger with either the LRU or APB object prioritization policies work best on average, there is no single policy that works best for all workloads and capacities. For instance, although the allocation trigger effectively identifies the beginning of new program phases for SNAP and QMCPACK, this policy is too conservative for Graph500 and results in worse performance due to cold and stale data remaining in the fast memory tier throughout much of the execution.

Overall, the results suggest that a single approach is unlikely to work best for all workloads and architectures. However, by providing tools and controls that allow user-level software to customize object tiering policies quickly and easily, this work can enable applications to adapt to various hardware capabilities and constraints more efficiently.

Table 5. Configurations used in the full scale evaluation.

Name	Description
First Touch (FT)	DRAM and NVM available to software in a flat address space. The OS assigns new allocations to DRAM until it is full and then assigns new data to NVM.
Memory Mode	Hardware exercises DRAM as a large last level cache to program data, which is stored in NVM. DRAM capacity is not visible to upper-level software.
Linux Tiering Patches (LTP) [29]	Similar to FT, but uses page-based profiling to promote hot pages to DRAM periodically. Algorithm and interface extend the standard NUMA balancing facility, with all parameters set to default values (i.e., `rate_limit_mbps` is set to 65536 and `wake_up_kswapd_early` and `scan_demoted` are disabled).
bkmalloc+MD avg-best	Employs the best performing bkmalloc+MD configuration on average (across all six workloads) for the small input sizes with the 50% capacity limit (i.e., Allocation Trigger, APB object priority).
bkmalloc+MD indv-best	Employs the best performing bkmalloc+MD configuration for each individual workload with the small input and 50% capacity limit.

5.4 Object Tiering with a Single Application at Full System Scale

Next, let us consider the performance of guided object tiering with the full size workloads. In addition to evaluating our approach at real system scale, these experiments allow for direct comparison with other system level tiering strategies, including hardware-directed caching and profile guided paging in the OS. The full set of configurations we compare for these experiments is described in Table 5. There are a few other important notes: 1) For FT, Memory Mode, and LTP, the application is configured to use the default system allocator and all 32 logical computing cores, 2) *bkmalloc+MD avg-best* and *bkmalloc+MD indv-best* use the results for the small workloads with the 50% capacity limit because this ratio most closely matched the ratio of each application's peak RSS to fast memory capacity on our server platform, 3) we ran *indv-best* for only two applications, IdenProf and WarpX, because these were the only workloads with a configuration that performed significantly better (i.e., outside the 95% confidence interval) than *avg-best* with the small input, and 4) to account for the larger capacities of the full scale workloads, we tested each application with 1 GB and 10 GB, and 100 GB allocation trigger thresholds and report performance of the best threshold. Hence, the *avg-best* results use a 1 GB threshold for QMCPACK and 10 GB thresholds for every other workload.

Figure 5 shows the execution times of the full size workloads with the *Memory Mode*, *LTP*, *bkmalloc+MD avg-best*, and *bkmalloc+MD indv-best* configurations, relative to the execution times of the *FT* configuration. We find that the automated tiering strategies significantly improve performance over the static FT approach in almost every case. While Memory Mode performs best on average, there are cases where our approach performs similarly (e.g., LULESH and IdenProf) or outperforms (by up to 6% for AMG) hardware-directed caching. In contrast to Memory Mode, our approach achieves high performance in these cases without requiring hard-wired architectural features and without sacrificing the capacity of the fast memory devices. It is also important to note that we selected full size inputs that do not exceed the capacity of the NVM devices so that we could directly compare our approach to Memory Mode. Larger inputs that take advantage of the additional capacity available in the software-based tiering configurations would almost certainly perform much worse (or crash) in Memory Mode.

Moreover, our approach can potentially operate more efficiently than Memory Mode, in some cases. Consider Figure 6, which shows the total memory read on our platform (including data transfers between memory tiers) during execution of each workload with each tiering configuration. All results are shown relative to the total memory read

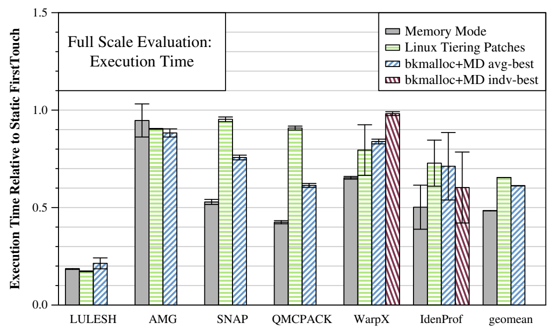

Fig. 5. Performance (execution time) of different tiering configurations with full inputs (lower is better).

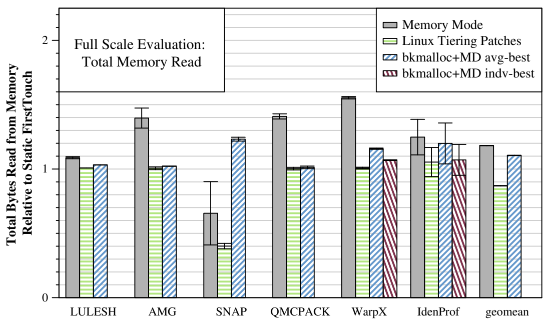

Fig. 6. Memory read bandwidth over the entire run for different tiering configurations with full inputs (lower is better).

during execution with the default FT configuration. While SNAP is a significant outlier,[5] most applications generate much more memory bandwidth with Memory Mode than with the software-based tiering configurations. On average, workloads aside from SNAP generate 23% more memory traffic in Memory Mode than with *bkmalloc+MD avg-best*. Further, since operational and data movement costs are the dominant factor in memory power consumption [20, 24], these results show there is significant potential for guided object tiering to reduce energy costs in memory, while still achieving the best possible performance, for most workloads.

Additionally, we find that *avg-best* achieves similar or better performance than *LTP* for every workload. Specifically, *avg-best* reduces execution time compared to *LTP* by 6.4%, on average, with a best case reduction of 32% for QMCPACK. The LTP approach extends the NUMA balancing infrastructure to scan and find the most frequently used pages in slower memory and then promotes them to faster memory. In addition to being entirely page based, LTP relies on timing-based heuristics that are very workload dependent and relatively slow to respond to changing access patterns [4]. As a result, LTP is much more conservative when moving data between tiers than our approach, as evidenced by the bandwidth results in Figure 6. However, the increased efficiency of profiling objects rather than pages and a migration policy that responds more quickly to changing program behavior can overcome the additional migration overhead of our approach, and leads to better overall performance for these workloads.

Lastly, in the two cases where the *indv-best* configuration differs from *avg-best*, *indv-best* exhibits similar or worse performance than *avg-best* in both cases. Thus, while the *avg-best* policy performs relatively well for all full size applications, some individual object tiering policies may not scale with different inputs of the same application.

5.4.1 Performance Impact of Proactive Object Tiering. Our next set of experiments examines the performance impact of proactively assigning data objects to a specific tier based on the tier recommendations of objects allocated at the same allocation site. For this evaluation, we use our framework with the bkmalloc+MD avg-best configuration and extend it to support proactive object tiering as described in Section 3.5.

There are a number of design choices and parameters that can impact the operation and effectiveness of this technique. For example, one option is to enable proactive object tiering for a given allocation site even if only a bare majority of its active objects have been assigned to the same tier, while another may require that all of a site's objects remain on the same tier for some time before altering the tier preference for that site. We considered a range of options for deciding if

[5]The full SNAP input seems to be very sensitive to microarchitectural effects on our platform. We suspect that changes in the data layout are causing outsize differences in memory traffic due to effects on processor caching and prefetching. Indeed, we found that simply disabling the hardware prefetchers causes *Memory Mode* and *LTP* to generate *more* memory bandwidth than the default configuration (though still less than *bkmalloc+MD avg-best*).

and when to apply proactive object tiering and chose to conduct our evaluation and present detailed results for the following conservative scheme. By default, each allocation site attempts to assign new objects to the faster memory tier, if space is available. At each migration event, the MAT Daemon examines all of the object tier assignments, and only if *all* of the active objects associated with a particular site have been assigned to the slower memory, it instructs the allocator to assign new data from that site to the slower memory tier, and otherwise, the allocator continues to prefer faster memory for that site.

Figure 7 shows the execution time of this approach for our full size workloads relative to the default bkmalloc+MD avg-best configuration without proactive object tiering. Thus, this technique significantly reduces the execution time for two of our six workloads, with speedups of 19% and 17% for QMCPACK and IdenProf, respectively. Unsurprisingly, we found these workloads have a relatively high number of unique allocation sites (ranging from 2 to 4 thousand) compared to the other benchmarks (with only a few hundred), and thus provide more opportunities for this technique to be applied effectively. Additionally, the execution times of the other benchmarks are mostly unaffected, with a worst case slowdown of 2% for LULESH. In experimenting with different parameters, we found that strategies that apply this approach more aggressively often achieve similar performance gains for cases such as QMCPACK, but may incur more significant slowdowns for benchmarks that are unaffected by this conservative scheme.

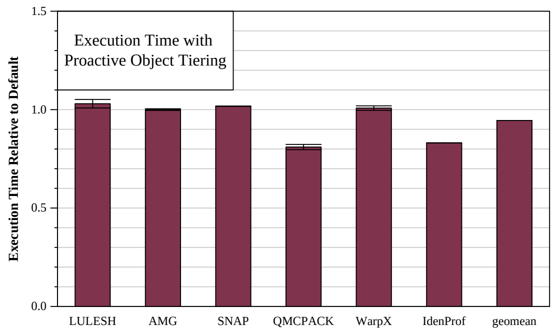

Fig. 7. Performance (execution time) of proactive object tiering with allocation site guidance. All results are shown relative to the bkmalloc+MD avg-best configuration (lower is better).

5.5 Flexible and Effective Object Tiering for Multi-Application Scenarios

Our final set of experiments aims to evaluate the effectiveness of our framework for managing the memory of multiple applications executing concurrently on a shared memory platform. For this evaluation, we consider a scenario where a low priority task with high capacity requirements executes alongside a high priority task that needs faster memory for high performance. Most heterogeneous memory platforms provide software tools, such as `numactl`, that enable applications to require or prefer that all of their allocations are assigned to a specific memory tier. Hence, one approach to address this scenario is to use `numactl`, or a similar tool, to bind allocations from the high priority task to the faster memory and allocations from the low priority task to the slower memory. However, this approach is inflexible and may under-utilize fast memory resources if the high priority application is not always running or does not use all of the capacity in the faster tier.

Another option is to use an existing system-wide management approach that manages data for both applications at a finer grain. For instance, one could employ the Linux Tiering Patches to conduct profile guided data tiering at the page level or one of the previously described policies available in MAT Daemon to conduct tiering of objects for both applications in a unified manner. While these approaches are less wasteful of fast memory resources, they do not consider application priority, and are thus vulnerable to slowdowns due to suboptimal placement of high priority data.

To address these limitations and support task sets with different priorities, we extended our framework with features to interpret and enforce application priorities during object tiering. Specifically, when an application initially connects to the MAT Daemon or sets its own priority, the daemon will compare the new priority to that of the other connected applications. If there are multiple applications with different priorities, the daemon will begin migrating all of the data corresponding to lower priority applications to the slower memory. Then, as long as a priority difference exists among the active task set, the runtime will continue to ensure that new and existing objects for the low priority tasks are always assigned to the slower memory tier. In this scenario, our extended framework may still employ guided object tiering to manage access to the faster memory, but it will only consider objects corresponding to the highest priority task(s) for assignment in the faster memory tier.

Hence, this policy assumes that the high priority application(s) will use most or all of the fast memory resources effectively. For cases where it is known that the high priority processes will use only a portion of fast memory resources, the policy could be further extended to move only a certain amount of the low priority data to slower memory and to limit new allocations so that the fast memory capacity that is necessary for high priority allocations is always available.

To evaluate this approach, we conducted experiments with two tasks executing concurrently on our experimental platform: 1) a high priority task that executes one of our benchmarks with the shared input, and 2) a low priority task that always executes QMCPACK with the full input, but restricted to use only 16 software threads. We chose QMCPACK's full input as the low priority task due to its capacity requirements, which exceed the capacity of DDR4, as well as its ability to stress both allocator and memory bandwidth on our platform. For each experiment, we start the low priority task first and allow it to initialize its main data structures and reach its main execution loop (about 20 minutes of program startup) before starting the high priority task. We also use `numactl` to assign application threads to computing cores such that threads from different applications do not share private (L1 and L2) caches. The last level (L3) cache and both memory tiers are shared among the threads in both the high and low priority tasks.

We tested each application with four memory tiering configurations:

(1) the baseline configuration uses `numactl` to assign data from the high priority task to fast (DDR4) memory and data from the low priority task to slow (Optane) memory,
(2) Linux Tiering Patches is the approach from [29] described in Table 5 with no application priorities,
(3) bkmalloc+MD Default is bkmalloc+MD avg-best from Section 5.4 with no application priorities, and
(4) bkmalloc+MD Priority employs the bkmalloc+MD avg-best approach with the priority scheme described above.

Figure 8 presents the execution time of the high and low priority tasks with the LTP, bkmalloc+MD Default, and bkmalloc+MD Priority configurations relative to the baseline configuration with each task bound to a specific memory tier. For each configuration, we plot separate bars for the performance of the high (left) and low (right) priority tasks. The names of the high priority tasks are presented along the x-axis.

We find that the LTP and bkmalloc+MD Default configurations exhibit very similar performance for both tasks. This result occurs because any potential benefit of one approach over the other is negated by their inability to consider application divisions and priorities. Specifically, we observed that both strategies capture a portion of each application's most frequently used data in the faster memory tier, but also leave a significant amount of hot data from both tasks on slower memory. Moreover, migrations are more frequent because phase changes in one application are more likely to evict hot data belonging to the other application. As a result, the high priority task slows down and the low priority task exhibits little to no improvement, in most cases.

Conversely, the high priority task of the bkmalloc+MD Priority configuration achieves very close to the ideal "everything-in-DRAM" performance of the baseline configuration. This result makes sense because our priority scheme reserves the entire upper tier for the high priority task as soon as its execution begins. The small slowdown is primarily due to the extra cost to move low priority data out of faster memory when the high priority task starts up. At the same

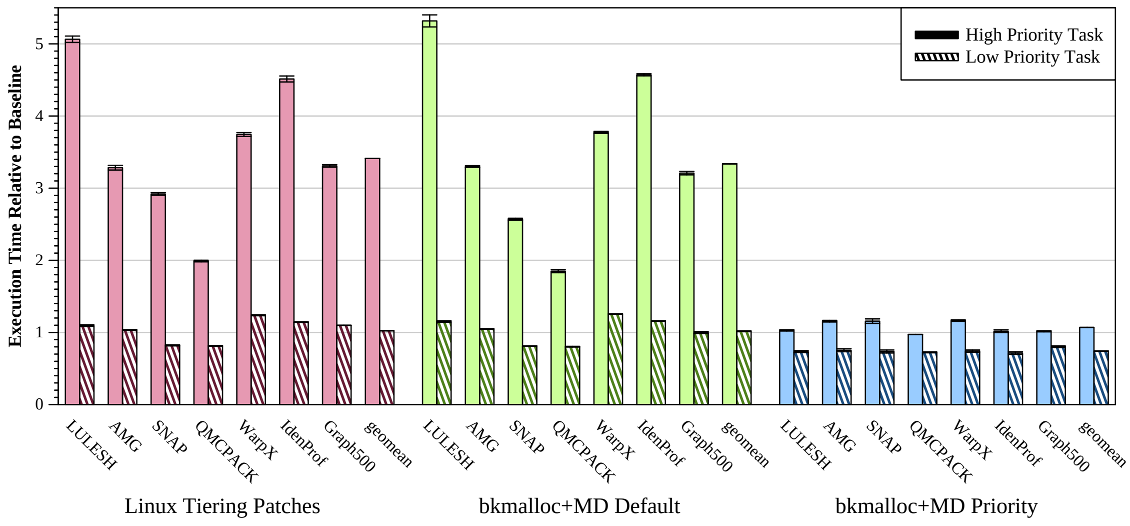

Fig. 8. Performance (execution time) of high and low priority applications executing concurrently with different tiering approaches (lower is better).

time, this approach enables significant speedups for the low priority task. In contrast to the baseline configuration, which binds all of the low priority task's data to the slower memory tier throughout its entire execution, the bkmalloc+MD Priority configuration is able to employ guided object tiering and utilize the faster memory more efficiently when the high priority task is not running. As a result, the low priority task enjoys a 26% speedup, on average, compared to the inflexible baseline configuration. Thus, our approach has potential to enable significantly better utilization of heterogeneous memory resources in mixed priority scenarios.

6 FUTURE WORK

There are many avenues for future research. First, we plan to develop techniques that leverage deeper integration of MAT Daemon with system-level migration routines to reduce data migration costs. In particular, we are working to build new tools that identify and replicate RD-only and RD-mostly data across memory tiers and coordinate moving pages into and out of faster memory with application behavior. Our approach will copy application data to larger and slower memory tiers during periods when system bandwidth is under-utilized. Later, during periods of higher demand, our runtime will be able to free up fast memory capacity quickly by simply remapping the page table entries of replicated data to point to copies in slower memory storage.

Next, we plan to enhance our proactive object tiering approach by extending it to interpret static, compiler-based analyses that identify and track other program data features, such as the set of routines or instructions that access groups of data. In this way, our runtime will be able to distinguish related sets of objects before dynamic profile information is available. At the same time, we will also extend our BPF tools to enhance system-level profiling and enable control over the placement of kernel memory objects, as described in Section 3.3.1.

Additionally, we will use our framework to evaluate object tiering with a broader set of architectural configurations and workloads. In the next few years, memory systems will become even more complex, with more diverse memory technologies and capabilities, including: high bandwidth and disaggregated memories, mixed HW/SW data management modes, processing-in-memory, and new types of GPUs and accelerators. By building upon standard Linux facilities and the longstanding NUMA API, our framework has been intentionally designed for portability to systems with new and diverse memory technologies. Moreover, its modular policy description framework enables researchers to rapidly devise and develop new tiering strategies for memory hardware with varying characteristics and constraints. Thus, as richer memory architectures and interconnects emerge, we plan to extend this work to investigate and exploit the diverse mix of hardware capabilities, platform capacities, and system organizations.

7 CONCLUSION

This work presents a novel software framework for enabling profile-guided object tiering on heterogeneous memory platforms. The approach employs a custom allocator and system-wide monitoring daemon to provide fast and flexible object tiering for single process or mixed workloads without offline profiling or recompilation of target applications. The framework is also used to evaluate the effectiveness of various choices made during object tiering, including how to prioritize objects for placement in the fast memory tier and when to migrate program data. Overall, the findings show that profile-guided object tiering produces substantial speedups compared to other software-based approaches, including a recent OS-based approach that uses page profiling to direct memory tiering. Moreover, it extends this framework with new features that are able to assign program data objects to the appropriate tier before profile information is available and new policies for utilizing fast, but capacity-limited, memories efficiently in scenarios where high and low priority applications share the same memory resources. The extended evaluation shows that these capabilities can be seamlessly integrated into the original framework and enable substantial performance and efficiency benefits for a variety of memory-intensive workloads in multiple execution scenarios.

REFERENCES

[1] Neha Agarwal and Thomas F. Wenisch. 2017. Thermostat: Application-transparent Page Management for Two-tiered Main Memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (Xi'an, China) *(ASPLOS '17)*. ACM, New York, NY, USA, 631–644. https://doi.org/10.1145/3037697.3037706

[2] Shoaib Akram. 2021. Performance Evaluation of Intel Optane Memory for Managed Workloads. *ACM Trans. Archit. Code Optim.* 18, 3, Article 29 (April 2021), 26 pages. https://doi.org/10.1145/3451342

[3] Jinyoung Choi, Sergey Blagodurov, and Hung-Wei Tseng. 2021. Dancing in the Dark: Profiling for Tiered Memory. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, Portland, OR, USA, 13–22. https://doi.org/10.1109/IPDPS49936.2021.00011

[4] Jonathan Corbett. 2023. Two memory-tiering patch sets. https://lwn.net/Articles/898766/

[5] Subramanya R. Dulloor, Amitabha Roy, Zheguang Zhao, Narayanan Sundaram, Nadathur Satish, Rajesh Sankaran, Jeff Jackson, and Karsten Schwan. 2016. Data Tiering in Heterogeneous Memory Systems. In *Proceedings of the Eleventh European Conference on Computer Systems* (London, United Kingdom) *(EuroSys '16)*. Association for Computing Machinery, New York, NY, USA, Article 15, 16 pages. https://doi.org/10.1145/2901318.2901344

[6] T. Chad Effler, Brandon Kammerdiener, Michael R. Jantz, Saikat Sengupta, Prasad A. Kulkarni, Kshitij A. Doshi, and Terry Jones. 2019. Evaluating the effectiveness of program data features for guiding memory management. In *Proceedings of the International Symposium on Memory Systems* (Washington, District of Columbia, USA) *(MEMSYS '19)*. Association for Computing Machinery, New York, NY, USA, 383–395. https://doi.org/10.1145/3357526.3357537

[7] Matt Fleming. 2021. *A thorough introduction to eBPF*. LWN.net. https://lwn.net/Articles/740157/

[8] Andy Georges, Dries Buytaert, and Lieven Eeckhout. 2007. Statistically Rigorous Java Performance Evaluation. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications* (Montreal, Quebec, Canada) *(OOPSLA '07)*. Assoc. for Computing Machinery, New York, NY, USA, 57–76. https://doi.org/10.1145/1297027.1297033

[9] 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. 2019. Basic Performance Measurements of the Intel Optane DC Persistent Memory Module. arXiv:1903.05714 [cs.DC]

[10] Brandon Kammerdiener. 2024. The Memory Auto Tiering Daemon. https://www.gitlab.com/bkammerd/mat-daemon

[11] Brandon Kammerdiener, J. Zach McMichael, Michael R. Jantz, Kshitij A. Doshi, and Terry Jones. 2023. Flexible and Effective Object Tiering for Heterogeneous Memory Systems. In *Proceedings of the 2023 ACM SIGPLAN International Symposium on Memory Management* (Orlando, FL, USA) *(ISMM 2023)*. Association for Computing Machinery, New York, NY, USA, 163–175. https://doi.org/10.1145/3591195.3595277

[12] Sudarsun Kannan, Yujie Ren, and Abhishek Bhattacharjee. 2021. KLOCs: Kernel-Level Object Contexts for Heterogeneous Memory Systems. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 65–78. https://doi.org/10.1145/3445814.3446745

[13] Jonghyeon Kim, Wonkyo Choe, and Jeongseob Ahn. 2021. Exploring the Design Space of Page Management for Multi-Tiered Memory Systems. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, virtual, 715–728. https://www.usenix.org/conference/atc21/presentation/kim-jonghyeon

[14] A. Kleen. 2004. A NUMA API for Linux.

[15] Mohammad Laghari, Najeeb Ahmad, and Didem Unat. 2018. Phase-Based Data Placement Scheme for Heterogeneous Memory Systems. In *2018 30th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. IEEE, Lyon, France, 189–196. https://doi.org/10.1109/CAHPC.2018.8645903

[16] Baptiste Lepers and Willy Zwaenepoel. 2023. Johnny Cache: the End of DRAM Cache Conflicts (in Tiered Main Memory Systems). In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. USENIX Association, Boston, MA, 519–534. https://www.usenix.org/conference/osdi23/presentation/lepers

[17] LLNL. 2014. CORAL Benchmark Codes. https://asc.llnl.gov/CORAL-benchmarks.

[18] Linux Manual. 2022. perf – performance analysis tools in Linux. https://man7.org/linux/man-pages/man1/perf.1.html

[19] Linux Manual. 2024. backtrace(3) – Linux Manual Page. https://man7.org/linux/man-pages/man3/backtrace.3.html

[20] Micron. 2023. TN-40-07: Calculating Memory Power for DDR4 SDRAM. https://www.micron.com/-/media/client/global/documents/products/technical-note/dram/tn4007_ddr4_power_calculation.pdf

[21] M. Ben Olson, Brandon Kammerdiener, Michael R. Jantz, Kshitij A. Doshi, and Terry Jones. 2022. Online Application Guidance for Heterogeneous Memory Systems. *ACM Trans. Archit. Code Optim.* 19, 3, Article 45 (jul 2022), 27 pages. https://doi.org/10.1145/3533855

[22] M. Ben Olson, Tong Zhou, Michael R. Jantz, Kshitij A. Doshi, M. Graham Lopez, and Oscar R. Hernandez. 2018. MemBrain: Automated Application Guidance for Hybrid Memory Systems. In *2018 International Conference on Networking, Architecture and Storage, NAS 2018, Chongqing, China, October 11-14, 2018*. IEEE, Chongqing, China, 1–10. https://doi.org/10.1109/NAS.2018.8515694

[23] Ivy Bo Peng, Roberto Gioiosa, Gokcen Kestor, Pietro Cicotti, Erwin Laure, and Stefano Markidis. 2017. RTHMS: A Tool for Data Placement on Hybrid Memory System. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on Memory Management* (Barcelona, Spain) *(ISMM 2017)*. ACM, New York, NY, USA, 82–91. https://doi.org/10.1145/3092255.3092273

[24] Ivy B. Peng, Maya B. Gokhale, and Eric W. Green. 2019. System Evaluation of the Intel Optane Byte-Addressable NVM. In *Proceedings of the International Symposium on Memory Systems* (Washington, District of Columbia, USA) *(MEMSYS '19)*. Association for Computing Machinery, New York, NY, USA, 304–315. https://doi.org/10.1145/3357526.3357568

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

[26] Jie Ren, Dong Xu, Junhee Ryu, Kwangsik Shin, Daewoo Kim, and Dong Li. 2024. MTM: Rethinking Memory Profiling and Migration for Multi-Tiered Large Memory. In *Proceedings of the Nineteenth European Conference on Computer Systems* (<conf-loc>, <city>Athens</city>, <country>Greece</country>, </conf-loc>) *(EuroSys '24)*. Association for Computing Machinery, New York, NY, USA, 803–817. https://doi.org/10.1145/3627703.3650075

[27] Steve Scargall. 2023. How To Extend Volatile System Memory (RAM) using Persistent Memory on Linux. https://stevescargall.com/2019/07/09/how-to-extend-volatile-system-memory-ram-using-persistent-memory-on-linux/

[28] H. Servat, A. J. Peña, G. Llort, E. Mercadal, H. Hoppe, and J. Labarta. 2017. Automating the Application Data Placement in Hybrid Memory Systems. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, Hawaii, USA, 126–136. https://doi.org/10.1109/CLUSTER.2017.50

[29] Vishal Verma. 2022. Intel Tiering Patches for Linux. https://git.kernel.org/pub/scm/linux/kernel/git/vishal/tiering.git/

[30] Kai Wu, Yingchao Huang, and Dong Li. 2017. 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* (Denver, Colorado) *(SC '17)*. ACM, New York, NY, USA, Article 58, 14 pages. https://doi.org/10.1145/3126908.3126923