MemBrain: Automated Application Guidance for Hybrid Memory Systems

M. Ben Olson* Tong Zhou* Michael R. Jantz* Kshitij A. Doshi† M. Graham Lopez‡ Oscar Hernandez‡

*Electrical Engineering and Computer Science Department
University of Tennessee, Knoxville
{molson5,tzhou9,mrjantz}@utk.edu

†Intel® Corporation
Chandler, AZ
kshitij.a.doshi@intel.com

‡Computer Science and Mathematics Division
Oak Ridge National Laboratory
Oak Ridge, TN
{lopezmg,oscar}@ornl.gov

***Abstract*—Computer systems with multiple tiers of memory devices with different latency, bandwidth, and capacity characteristics are quickly becoming mainstream. Due to cost and physical limitations, device tiers that enable better performance typically include less capacity. Such heterogeneous memory systems require alternative data management strategies to utilize the capacity-constrained resources efficiently. However, current techniques are often limited because they rely on inflexible hardware caching or manual modifications to source code.**

This paper introduces MemBrain, a new memory management framework that automates the production and use of data-tiering guidance for applications on hybrid memory systems. MemBrain employs program profiling and source code analysis to enable transparent and efficient data placement across different types of memory. It automatically clusters data with similar expected usage patterns into page-aligned regions of virtual addresses (arenas), and uses offline profile feedback to direct low-level tier assignments for each region. We evaluate MemBrain on an Intel Knights Landing server machine with an upper tier of limited capacity (but higher bandwidth) MCDRAM and a lower tier of conventional DDR4 using a selection of high-bandwidth benchmarks from SPEC CPU 2017 as well as two proxy apps (Lulesh and AMG), and one full scale scientific application (QMCPACK). Our evaluation shows that MemBrain can achieve significant performance and efficiency improvements compared to current guided and unguided management strategies.

I. Introduction

Recent years have witnessed the emergence of several new memory technologies with distinct advantages over conventional double data rate (DDR) SDRAM. Storage-class memories (SCMs), –such as spin-torque transfer (STT) RAM, phase change memory (PCM), and resistive RAM (ReRAM), enable durable byte-addressable storage, with random access latencies that are multiple orders of magnitude shorter than state-of-the-art solid state and spinning disks. Several SCM technologies also allow finer resolution semiconductor integration and, since they do not require refresh power, are more energy efficient than DRAM. Another recent technology often referred to as "on-package" (or "die-stacked") memory places one or more 3D stacks of memory inside the same package as the processing unit to deliver orders of magnitude higher bandwidth, and thus has a strong potential to address the memory wall. Despite their promise, each of these new technologies also comes with its own set of drawbacks. At introduction, SCMs will lag behind modern volatile memory in access latencies and bandwidths [1], and are expected to present non-uniform performance under extreme load. Likewise, while on-package memory delivers high access bandwidth, it is generally only available in limited capacity: for example, in Intel®'s Xeon Phi processor, only eight MCDRAM modules (2GB each) are currently available [2].

To keep pace with rapidly growing data demands, many next-generation computing systems will provide multiple tiers of memory storage, including: 1) an on-package tier with high-performance but limited capacity, 2) a tier of conventional DRAM (e.g. DDR), and 3) a (physically) non-volatile memory tier with high capacity, but with less bandwidth and longer latencies for reads and writes. Depending on the configuration, the third tier may also provide durable in-memory storage or simply extend the capacity of volatile RAM.

Propelled by this shifting landscape, the architecture and systems communities are developing new data management strategies to take advantage of the different strengths of each tier. A popular software-based approach uses either the operating system (OS) by itself, or the OS in conjunction with the application to assign data into different memory tiers, with facilities to allow migrations of data between tiers as needed. Some hybrid memory systems already provide API's that allow applications to control the placement of their data objects through the use of source code annotations [3], [4]. These finer-grained controls permit developers to coordinate tier assignments with data allocation and usage patterns, and can potentially expose powerful new efficiencies not found by unguided strategies, such as hardware-managed caching or first touch paging.

Recent research has also proposed data profiling and allocation tools to generate and apply software-level guidance for hybrid memory management [5], [6], [7], [8], [9], [10], [11]. While these studies evince some of the benefits of application-guided data tiering, there are still significant limitations that can hinder their effectiveness for real programs. For instance, all of these prior works use simulation models or coarse-grained sampling to estimate bandwidth and/or latency re-

quirements for application data, and therefore are vulnerable to inefficiencies that arise from sparse or inaccurate profiles. Furthermore, most current toolsets require users to manually tag data structures and allocation sites in the application source code, and limit guidance to only a portion of heap objects.

Our approach, called MemBrain, aims to address the limitations of current software-based tiering strategies and enable more efficient data placement on hybrid memory systems. MemBrain facilitates the use of data-tier guidance by associating profiles of memory usage characteristics, such as bandwidth and capacity, with program *allocation sites*. Our framework includes a set of offline profiling tools that generate site-tier recommendations automatically, as well as a static compilation pass that annotates allocation instructions with unique identifiers, and optionally considers function call context to distinguish sites reached by multiple paths. During execution, applications interface with the MemBrain runtime, which intercepts their allocation instructions, and uses the static annotations to assign new data to the recommended tier.

This work describes the design and implementation of MemBrain, which is the first hybrid memory management framework to enable automated data-tier guidance with virtually no execution overhead. To test our approach, we deploy it on an Intel® Xeon Phi server machine with multiple memory tiers, including: an upper tier of high-bandwidth, but limited capacity, on-chip memory (MCDRAM), and a lower tier of conventional (DDR4) memory with more capacity. Our evaluation uses data-intensive workloads as well as a full-scale scientific computing application to compare the effectiveness of guided data tiering with standard unguided management strategies. Additionally, we use MemBrain to investigate and quantify the importance of various aspects of our approach, including: profile accuracy, the use of function call context to distinguish allocation sites, and alternative bin-packing algorithms for assigning data to each memory tier.

The main contributions of this work are as follows:

1) We design, implement, and evaluate MemBrain: a software-based data management framework that allows applications to automatically generate and apply fine-grained tiering guidance on real hybrid memory systems.
2) We develop novel tools to collect profiles of memory bandwidth and capacity and use them to investigate and quantify trade-offs between profile overhead, accuracy, and performance.
3) We demonstrate that considering function call context can enhance automated tiering guidance for some applications, and construct a static compilation pass to distinguish data with different allocation contexts with no execution time overhead, and
4) We propose a new approach for partitioning application data into device tiers under different capacity constraints, and present evaluation that shows that our strategy outperforms current guided and unguided software-based data management techniques.

II. Related Work

Spurred by such computing trends as Big Data and Exascale computing, research interest in emerging memory devices and how to utilize them efficiently has grown significantly in the last few years. Several recent projects have proposed software-driven strategies for managing data on multi-tier memory systems [5], [12], [6], [13], [14], [8], [7], [15], [9], [10], [16], [11]. Similar to MemBrain, many of these approaches rely on profiling or runtime feedback to steer data placement across the memory hierarchy. This work complements these prior efforts by introducing a novel tool that enables more accurate profiles of memory usage behavior, and is the first study to investigate trade-offs of profile accuracy and performance for feedback-directed hybrid memory management. Some of these prior works have also proposed automating the application of data-tier guidance by associating it with program allocation sites [9], [11]. However, prior works either relied on simulation, and did not account for the overhead of context detection, or used an expensive stack-walking strategy to detect function call context at each allocation request. In contrast, MemBrain uses a custom compilation pass to generate distinct code paths for each allocation instruction and its context, thereby avoiding the overhead of context detection during execution.

Some hybrid memory systems include architectural features that allow data to move across the memory hierarchy without direction from application software. For instance, the "cache-mode" option on the Xeon Phi platform we use in this work exercises the MCDRAM tier as a large last-level cache to DDR4. Previous research has also explored the use of large capacity DRAM caches, and new designs have recently been proposed to improve their performance [17], [18], [19], [20], [21], [22]. While in-memory 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. It can also impose unpalatable architectural costs as the high-bandwidth tier must either be implemented as a tagless direct-mapped (non-associative) cache, or it requires storage for associative tags. By automating the application of software guidance, our approach aims to enable transparent data management for hybrid memory platforms, without the limitations of hardware-based approaches.

A number of other works have integrated application-level guidance with physical data management in the OS and hardware on conventional (i.e., homogeneous) memory platforms. Earlier projects developed frameworks or APIs to expose kernel resources to applications [23], [24] or to facilitate communication between upper- and lower-level data management routines [25], [26], [27], [28]. More recent efforts have combined these cross-layer approaches with automated collection of high-level guidance to address a variety of issues, including: DRAM energy efficiency [29], [30], cache pollution [31], traffic congestion for non-uniform memories [32], and data movement costs for non-uniform caches [33], [34]. These projects demonstrated that software profiling and analysis are powerful and effective tools for guiding low-level data

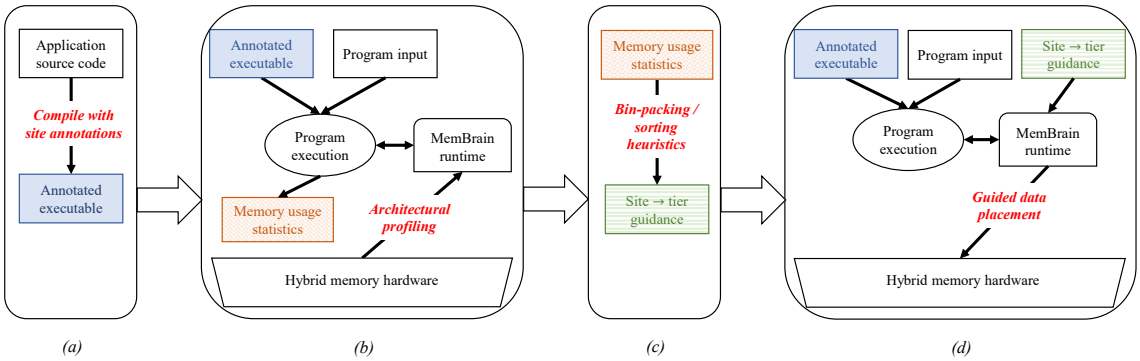

Fig. 1. Automated data-tiering guidance with MemBrain. (a) Compile executable with source code annotations at each allocation site, optionally with function call context, (b) Profile memory usage behavior of each site in a separate program run using architectural counters or sampling, (c) Employ bin-packing / sorting heuristics to assign data-tier recommendations to each site, (d) Apply data-tiering recommendations during subsequent program executions.

management, and have helped inspire our approach. While their purposes and goals are very different from ours, we will continue to draw upon their lessons as we adapt MemBrain for new use cases and different types of memory hardware.

III. Automated Application Guidance for Hybrid Memory Systems

The Intel® Xeon Phi-based platform we use for this study employs a two-layer memory system. The upper (MCDRAM) tier exhibits similar RD/WR latencies as the lower (DDR4) tier, but is able to sustain much higher bandwidths, and contains only a fraction of the capacity. Thus, our goal is to derive placement decisions that maximize the rate of access to the upper tier within its capacity constraint. Our approach collects profiles of memory usage behavior, and associates them with the static instructions that allocate program data (also called *allocation sites*). Each allocation site corresponds to a source code file name and line number and may optionally include part or all of the call path leading up to the site. A separate analysis pass (described in Section III-A) also converts the usage profiles into tier recommendations for each site prior to guided execution.

Figure 1 illustrates the process of generating and applying tiering guidance with MemBrain. There are two main motivations for this approach. First, associating data usage metrics with allocation sites, rather than address ranges or individual heap objects, makes it easier to compare memory behavior and apply tier recommendations across different executions of the same program. Second, intuitively, applications tend to use all or most of the data allocated from a particular site for the same purpose, and often allocate data for different purposes from different sites. Thus, this approach is also an effective means for grouping data into equivalence classes with similar expected usage behavior.

The line plots in Figure 2 help demonstrate the potential of this approach for multi-layer memory systems. To construct these plots, we first used MBI-based profiling tools (described in Section IV-C) to measure the bandwidth and capacity of the data associated with each allocation site in our workloads. We then computed a *hotness* score for each site as the average bandwidth (GB/sec) divided by the maximum resident set size (RSS) (GB) of its data. The markers in each graph show the cumulative capacity (on the x-axis) and bandwidth (on the y-axis) of a particular workload's allocation sites, plotted in the order of their hotness scores, and normalized by the average bandwidth and peak RSS of the entire workload. We provide two plots for each workload: the upper plot identifies allocation sites by their source code location alone, while the lower plot considers an additional four layers of function call context to distinguish sites reached by different call paths.

The plots reveal that most of the bandwidth of most workloads is contained within a relatively small portion of their capacity. For example, the upper (no context) plot for *qmcpack* shows that objects that account for only 23% of the workload's capacity also generate more than 63% of its bandwidth. Some workloads, such as *imagick* and *amg*, allocate almost all of their data from a single source code location. In these cases, there is not enough differentiation in the usage profiles for the upper plots to exhibit any sort of bandwidth clustering effect. However, the lower plots show that using call stack context to distinguish additional sites often enables more effective bandwidth clustering, especially when the number of sites without context is relatively small.

A. Assigning Application Data to Hybrid Memory Tiers

Several recent projects have proposed data management strategies that rely on program profiling to separate application data into different capacity-constrained memory regions [35], [36], [9], [11], [30]. For this work, we adopt two techniques from these recent studies for use with MemBrain. Our first adopted approach views the task of assigning application data into different device tiers as an instance of the classical 0/1 knapsack optimization problem. In this formulation, each allocation site is an item with a certain value (bandwidth) and weight (capacity). The goal is to fill a knapsack such that the total capacity of the items does not exceed some threshold (chosen as the size of the upper tier), while also maximizing the aggregate bandwidth of the selected items. Our second approach, called *hotset* and adopted from [11], aims to avoid a weakness of knapsack, namely, that it may

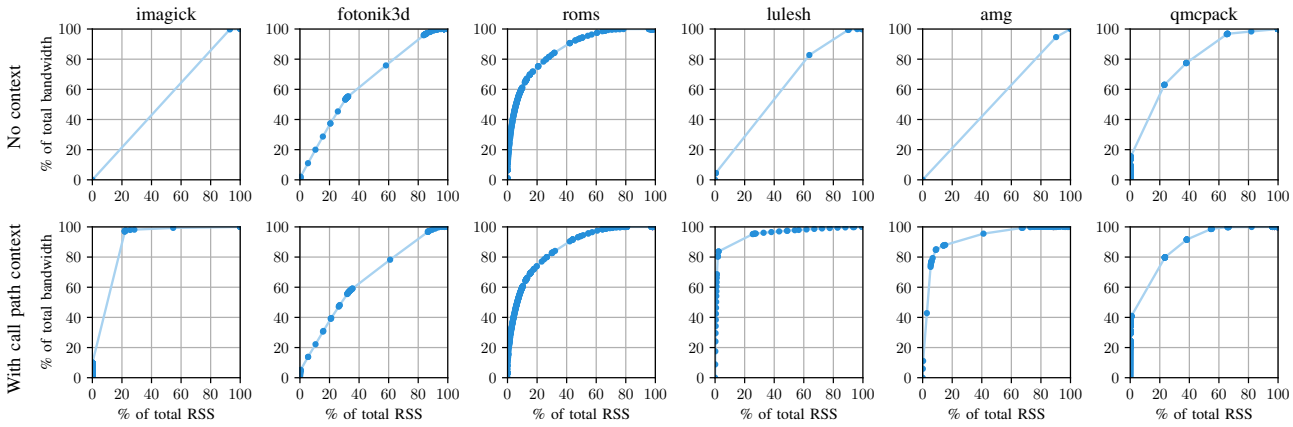

Fig. 2. Line plots of cumulative capacity and bandwidth associated with program allocation sites. The upper graphs do not use any function call context to distinguish sites reached by different call paths. The lower graphs use four additional layers of function call context to identify each site.

exclude a site on the basis of its capacity alone, even if it exhibits high bandwidth. Hotset simply sorts the sites by their bandwidth per unit capacity (similar to the hotness scores described above), and selects sites until their aggregate size exceeds a soft capacity limit. For example, if the capacity of the upper tier is C, then knapsack will select allocation sites so that their aggregate size is just below C, while hotset stops adding sites after the total size is just past C.

Since the hotset approach intentionally over-prescribes capacity in the upper tier, some cold or lukewarm data may end up crowding out the hottest objects. To address this drawback, we also propose an extension to hotset, called *thermos*, which aims to keep as much bandwidth as possible in the upper tier. Thermos is similar to hotset with one exception: it only assigns a new site to the upper tier if the bandwidth it contributes is greater than the aggregate bandwidth of the hottest data it could potentially displace. In this way, thermos avoids crowding out performance-critical data, while still allowing large-capacity, high-bandwidth sites to place a portion of their data in the upper-level memory.

B. Guided Runtime Data Management

During a guided run, MemBrain uses a custom runtime to intercept and apply data-tiering recommendations at each allocation instruction. The runtime divides new objects into independent groups of page-aligned address ranges known as arenas, and uses a system interface to inform the OS memory manager of its preferred arena-tier assignments. Depending on the hardware platform and application characteristics, there are a number of design choices and trade-offs related to arena management that can impact the performance of this approach. Since this study primarily focuses on the feasibility and effectiveness of static (offline) data-tiering guidance, we employ a *static arena allocation scheme* with distinct arenas for each memory hardware tier. This scheme assigns new objects to an arena corresponding to their recommended tier, and keeps object locations and arena-tier assignments fixed throughout the entire program run.

IV. MemBrain Implementation

MemBrain's profiling and data management tools rely on two common facilities to enable automated software guidance: 1) a static compilation pass that annotates each call site with a unique identifier and optionally distinguishes sites with different calling context, and 2) a custom memory allocator that interfaces with the annotated executables, and assigns application data into arenas according to a specified arena allocation scheme.

A. Allocation Site Identification and Annotation

Implemented as part of the LLVM infrastructure [37], our custom compilation pass traverses the application's static call graph, assigns unique integer identifiers to each allocation instruction, and replaces each such instruction with a corresponding invocation of the MemBrain allocator with the identifier passed as an argument. The current pass supports source code written in C, C++ and Fortran 90, and identifies the following allocation routines from their standard libraries: `malloc`, `calloc`, `realloc`, `posix_memalign`, `aligned_alloc`, `_Znam`, `_Znwm`, and `f90_alloc` as well as its variants.

Many applications use only one or a few allocation instructions throughout their entire source code, but invoke these instructions from a large number of different code paths during execution. To enhance guidance for these programs, our pass provides an option to clone functions in the source code so that the call path leading to each allocation instruction is unique up to some length n. Figure 3 shows an example of applying the function cloning pass with $n = 2$. The original call graph on the left contains a single allocation instruction that is reachable by three call paths of length 2: AD, BD, and CD. Starting from the inner-most node, the pass walks each path back (towards main) and creates a separate copy (including the subtree) of the first node it finds with multiple parents, resulting in the call graph shown on the right. Whenever the graph is modified, the pass recomputes unique identifiers and

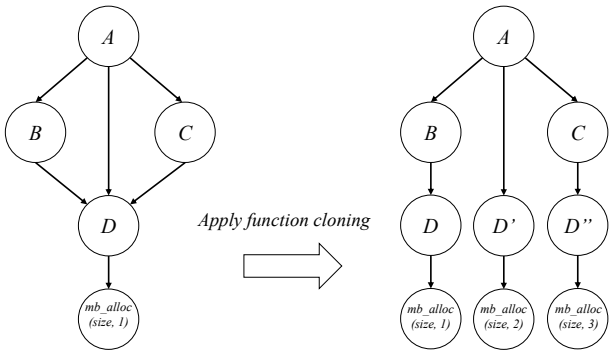

Fig. 3. Function cloning example. The original call graph contains one allocation instruction reachable by three call paths. After cloning, there are three distinct allocation instructions, each with their own unique call path.

call paths for each allocation instruction before continuing the cloning operation. Termination occurs when there are no call paths of length n (or less) that end at the same allocation site.

B. Arena Allocation and Management

The MemBrain runtime extends the popular jemalloc [38] allocator (v. 5.1) to distribute application data into arenas. Exactly how data maps to arenas depends on whether the runtime is used to profile or guide data management. For guided runs, MemBrain implements the static arena allocation scheme described in Section III-B with exclusive DDR4 and MCDRAM arenas for each program thread. At each allocation request, the runtime determines the allocating thread (using `pthread_self` [39]), queries the site-tier recommendation from the pre-loaded guidance, and uses it to assign the new data to the appropriate arena. MemBrain attaches tier recommendations to each arena whenever one is created or resized using the `mbind` system call with the `MPOL_PREFERRED` mode [40]. In contrast `MPOL_BIND` mode, `MPOL_PREFERRED` allows the application to over-prescribe the MCDRAM tier without causing an error.

The MemBrain profiling tools rely on an alternative arena management scheme that uses exactly one arena per allocation site shared between all application threads. This configuration maps allocation sites to new arenas as they are reached, but requires synchronization if multiple threads attempt to access the same arena. Assigning each site's data into its own arena allows our tools to profile application data usage over coarse-grained groups of address ranges, as described next.

C. Profiling Memory Bandwidth and Capacity

Most modern processors, including the Intel® Xeon Phi we use for this work, provide architectural support for monitoring memory usage at different granularities, rates, and overheads. To better understand the trade-offs between profile accuracy and performance, we created two tools that use different techniques for profiling the memory usage of application data.

The first tool employs Precise Event Based Sampling (PEBS), which is an architectural sampling facility for x86 platforms that has previously been used to guide data placement on conventional and hybrid memory systems [32], [7], [16]. Our tool configures PEBS to sample and record the targets of memory read instructions that miss the last level cache (LLC) (using MEM_LOAD_UOPS_RETIRED.LLC_MISS) into a hardware buffer. When the buffer is full, the runtime counts the number of addresses in the buffer that fall within the bounds of an arena. In this way, the PEBS-based profiler constructs a heatmap of the relative access rates of data associated with each allocation site. Our tool also uses the Linux pagemap facility [41] to periodically count the number of pages in each arena that are resident in physical memory. At the end of the profile run, the tool outputs the access counts and peak resident set size associated with each allocation site.

Our second tool introduces a novel profiling technique, which we call Memory Bandwidth Isolation (MBI). The MBI technique uses precise architectural counters to provide more accurate estimates of bandwidth utilization than sampling, but requires multiple program runs to build a full profile of the application. For each run, the MBI-based tool places the data associated with a particular allocation site onto its own memory tier and all other data on the opposite tier. It then uses the `UNC_IMC_DRAM_DATA_READS` and `UNC_IMC_DRAM_DATA_WRITES` counters to monitor traffic on the the isolated tier. To build a complete MBI-based profile, we first use a separate program run to determine the sites that are reached during execution and their capacities, and then use the MBI-based tool to profile the bandwidth of each site.

V. Experimental Setup

A. Platform

All of our experiments were run on an Intel® Knight's Landing (KNL) machine with a Xeon Phi 7230 CPU (1.30GHz). The processor includes 64 compute cores with quadruple hyper-threading (256 hardware threads) and a unified 32MB L2 cache. The CPU interfaces with a two-level hybrid memory system with 16GB (8x2) of Intel® MCDRAM and 192GB (6x32) of Samsung DDR4-2400 ECC DRAM (M393A4K40BB1-CRC). We installed CentOS 7.3.1611, and use its default Linux kernel (v. 3.10.0-514).

B. Workload Description

Our evaluation uses a selection of applications from the SPEC CPU® 2017 benchmark suite [42] as well as three workloads from the CORAL set of throughput benchmarks [43]. For CPU 2017, we consider only those workloads that exhibit significant ($> 10\%$) speedups when placed entirely on the MCDRAM. Four workloads (*imagick*, *fotonik3d*, *lbm*, and *roms*) met this criteria, but we exclude *lbm* because it allocates the vast majority of its capacity within a single heap object, and is not likely to experience any benefit (or drawback) from automated allocation guidance. From CORAL, we select two proxy applications (*lulesh* and *amg*) and one full scale scientific application (*qmcpack*) based on their potential to stress the memory performance of our platform as well as our own prior expertise with these applications.

TABLE I

WORKLOAD STATISTICS. COLUMNS SHOW: WORKLOAD NAME, INPUT PARAMETERS, PERFORMANCE METRIC, PERFORMANCE WITH ALL DATA ON DDR4 / MCDRAM, PEAK RESIDENT SET SIZE (IN GB), AS WELL AS EXECUTABLE SIZE (IN MB) AND # OF ALLOCATION SITES (IN THE EXECUTABLE AND REACHED DURING EXECUTION) WITH AND WITHOUT APPLYING FUNCTION CLONING TO DISTINGUISH ALLOCATION CALL PATH CONTEXT.

Workload	Input Parameters	Performance Metric	DDR Perf.	MCDRAM Perf.	RSS (GB)	XZ (MB)	XZ (MB) (w/ cxt.)	Static Sites	Reached Sites	Static (w/ cxt.)	Reached (w/ cxt.)
imagick	Default SPECspeed `ref` input	Seconds	609.2	531.0	11.3	6.6	21	6	4	7,491	168
fotonik3d	Default SPECspeed `ref` input	Seconds	430.9	332.6	9.6	1.4	1.5	279	129	355	141
roms	`ref` w/ `NTIMES=20`, `Lm=1440`, `Mm=768`, `N=30`	Seconds	196.4	110.6	11	3.2	8.8	640	395	15,211	439
lulesh	`-s 220 -i 5 -r 11 -b 0 -c 64 -p`	$\frac{zones}{seconds}$	690.3	1,284.5	10.5	0.4	0.4	17	16	48	47
amg	`-problem 2 -n 120 120 120`	$\frac{nnz*(iters+steps)}{seconds}$	1.2e7	2.5e7	11.5	3.1	15	29	2	11,969	290
qmcpack	NiO with the VMC method and 256 walkers	$\frac{blocks*steps*N_w}{seconds}$	8.9	17.2	13.7	48	55	3,170	216	13,833	776

Table I lists usage information and statistics for our selected workloads. Descriptions of each workload are as follows:

- Imagick (ImageMagick) does in-memory manipulations of a 2068x1380 pixel image: resizing, sharpening, blurring, despeckling, rotating, etc., written in ANSI C.
- Fotonik3D uses the finite-difference time-domain (FDTD) method for the Maxwell equations to compute the transmission coefficient of a photonic waveguide, written in Fortran.
- ROMS, or Regional Ocean Modeling System, is a free-surface hydrostatic, primitive equation model discretized with a terrain-following vertical coordinate system, written in Fortran.
- Lulesh (v. 2.0) performs a hydrodynamics stencil calculation with very little communication between computational units. Makes heavy use of vectorization instructions, written in C.
- AMG (v. 1.0) is a parallel algebraic multigrid solver for linear systems arising from problems on unstructured grids. Highly synchronous and places very large demands on memory bandwidth, written in ISO-C.
- QMCPACK (v. 3.4) is a Quantum Monte Carlo simulation code with near-perfect weak scaling and extremely low communication. Mostly written in C++, it makes extensive use of high-level language features such as templating and `std::allocator`.

C. Common Experimental Configuration

All workloads were compiled using the LLVM compiler toolchain (v. 4.0.1) with `-O3` and `-march=knl`. C/C++ codes use the standard `clang` frontend, while Fortran codes are converted to LLVM IR using Flang [44]. The comparison configurations ('DDR4-only', 'MCDRAM-only', 'first touch', and 'cache-mode') all use the unmodified jemalloc [38] allocator (v. 5.0.1) with default parameters. Configurations that use MemBrain annotate the LLVM IR, and optionally apply function cloning, after all other optimizations have already been applied. For executables with call path cloning, we opted to clone up to four layers of context (i.e., $n = 4$) because we found that this configuration distinguishes a large enough number of allocation paths to aid memory usage guidance, but is still feasible for MBI-based profiling. A full evaluation of this parameter is left as future work.

All workloads use OpenMP with 256 software threads (one for each hardware thread) and (if applicable) one MPI rank for each evaluation run. We report performance as the mean average of five runs. To estimate the variability of our results, we also compute 95% confidence intervals for the difference between the means of the experimental and baseline configurations, and plot these intervals as error bars on the appropriate graphs [45]. Non-heap (i.e., global and stack) data is a relatively small portion of the total for all of our workloads.[1] All guided configurations use `numactl -p` to prefer assignment of non-heap data to the MCDRAM tier.

All of our profiling runs use the same program input and number of threads as the evaluation run. The PEBS-based profiling employs a fixed buffer size of 4KB per core. It samples resident set size (with pagemap) in a separate thread every 5 seconds and at the end of the run. MBI-based profiling isolates the data of the target site on the DDR4 tier and places the remaining data on MCDRAM.[2] It samples DDR4 bandwidth every second using the Intel ® Performance Counter Monitor software [46], and uses the average over the entire run as the bandwidth for the target site.

To bound the scope of our evaluation, the workloads execute on an otherwise idle machine and assume an environment with limited upper tier capacity. Input and problem sizes are intentionally selected so that the entire application fits within the MCDRAM tier.To limit the upper tier capacity available to the application, our experiments use a separate process to reserve a portion of the MCDRAM so that the remaining free space is only a fraction of the total capacity that is needed. This approach has two main advantages: 1) it allows for easier comparison with an ideal 'MCDRAM-only' configuration, and 2) it avoids the potential issue with the MBI-based profiling of running out of space on the lower capacity tier.

VI. EVALUATION

A. Overhead of Allocation Call Path Detection

We first consider the execution time overhead of distinguishing data allocations from different call paths. Previous efforts to automate memory usage guidance were either conducted in

[1]Non-heap data makes up about 12% of the memory footprint of *imagick*, and $< 2\%$ of all other workloads.

[2]In contrast to the alternative, this approach ensures that the MBI profiling will not slow down if the non-isolated data requires more bandwidth than is available on the DDR4 tier.

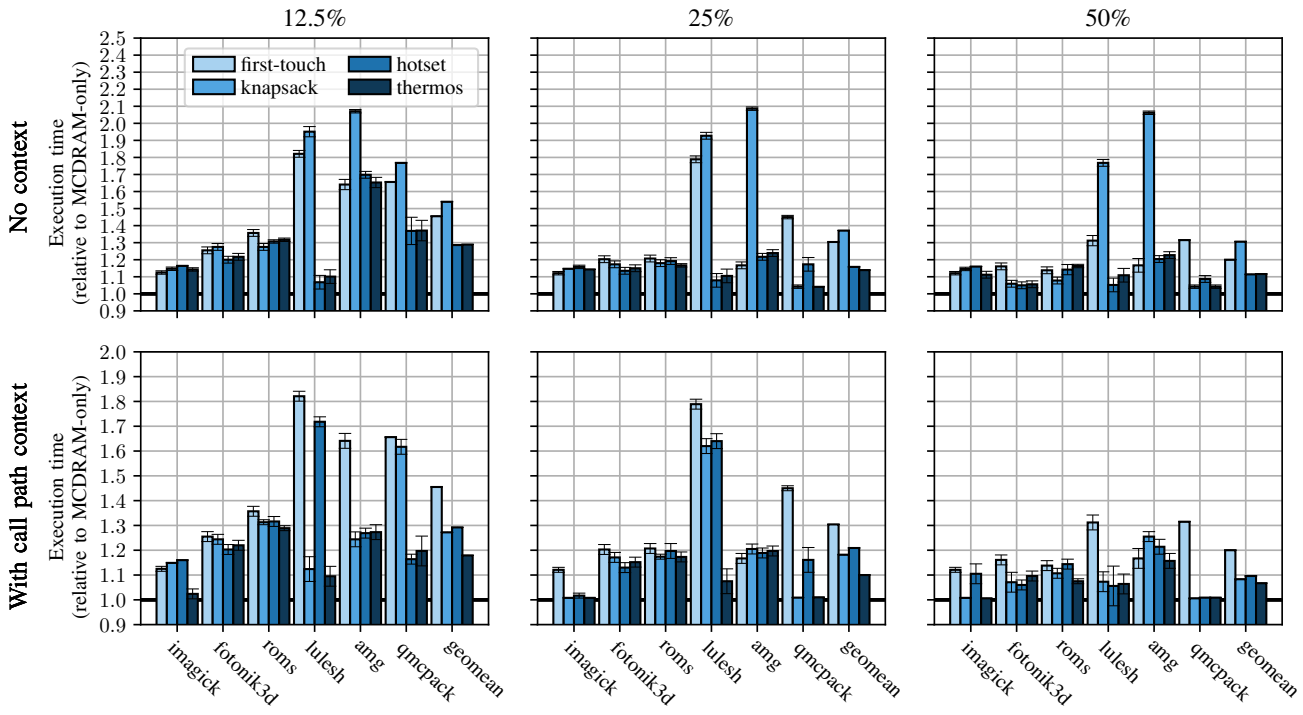

Fig. 4. Performance (execution time) of using MBI-based profiling for automated application guidance relative to MCDRAM-only (lower is better).

simulation or used workloads that did not allocate data very frequently [9], [11]. These studies were less concerned with the overhead of call path detection, and used a naïve stack-walking approach based on the `backtrace` facility from the C standard library to detect call path context at each allocation request. To demonstrate the potential of call path cloning to reduce these overheads, we implemented dynamic call path detection with `backtrace` in the MemBrain runtime. At each allocation request, the runtime invokes `backtrace` to collect up to four layers of call path context and stores unique contexts as keys in an efficient map structure. We tested both locking and non-locking versions of this approach and found no significant difference between the results.

Table II shows the execution time overhead of context detection with `backtrace` alongside the overhead of executables that use the call path cloning technique to distinguish allocation contexts. A separate column (KAllocs) shows the allocation rate (in thousands of allocations per second) for each workload. As with all other performance results in this section, the overhead is computed as the execution time of the experimental configuration relative to the default execution time. For consistent presentation of results, we also convert the performance metric of the throughput workloads (*lulesh*, *amg*, and *qmcpack*) to execution time by computing the reciprocal of their result. For these experiments, both the experimental and baseline configurations use only the MCDRAM tier.

Thus, while dynamic call path detection with `backtrace` can incur prohibitive overheads, static call path cloning does not cause any degradation in most cases, and is, on average, only 1% slower than default. Indeed, the cost of static call path cloning is mostly paid in larger executable sizes (as shown in Table I) and longer compilation times. However, even our prototype implementation of this technique adds only a few seconds (with *lulesh*) to a few minutes (with *qmcpack*) of compilation time for our workloads. It is also important to note that the overheads in Table II do not always capture the full expense of the `backtrace` technique as some of our workloads only begin timing after initializing most of their data structures. For instance, the initialization period of *qmcpack* is over seven times longer with dynamic call path detection ($\sim$ 870 seconds) than with static cloning ($\sim$ 120 seconds), but this difference is not reflected in Table II.

TABLE II
EXECUTION TIME WITH CONTEXT DETECTION TECHNIQUES RELATIVE TO MCDRAM-ONLY WITH NO CONTEXT DETECTION (LOWER IS BETTER).

Workload	KAllocs / Second	Backtrace Time	Static Time
imagick	35.2	2.92	1.01
fotonik3d	0.15	0.99	0.98
roms	39.8	4.80	1.05
lulesh	0.06	1.00	0.99
amg	40.2	1.17	1.03
qmcpack	34.1	1.05	1.00
geomean	24.9	1.61	1.01

B. Automated Application Guidance with MBI-based Profiling

Figure 4 shows the performance (execution time) of program runs that use the MBI-based profiling to guide data-

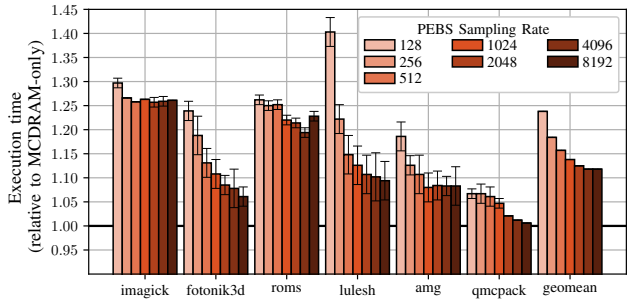

Fig. 5. Performance (execution time) of PEBS-based profiling with different sampling rates relative to default MCDRAM-only (lower is better).

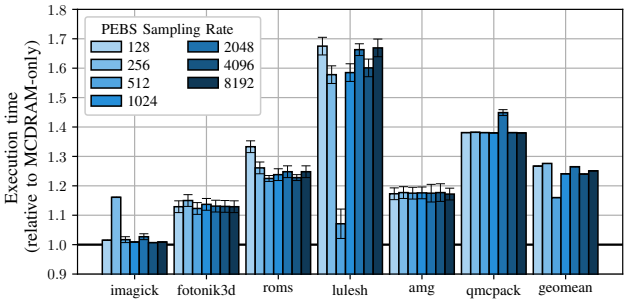

Fig. 6. Performance (execution time) of PEBS-based profiling with different sampling rates. All runs use the thermos strategy, 25% upper tier capacity limit, and are relative to MCDRAM-only (lower is better).

tier assignments relative to the performance with the entire application on the MCDRAM tier. Across the three columns, we present different sets of results to show the performance when the capacity of the upper tier is constrained to be a different percentage of the peak RSS of the workload (either 12.5%, 25%, or 50%). The upper row shows the results when no additional allocation context is used, while the lower row shows the performance of executables that use function cloning to distinguish allocation sites with different call path context. Within each subfigure, we plot four bars for each workload to show performance with an unguided (static) first touch policy (on the left), as well as with guidance constructed using the knapsack, hotset, and thermos strategies. The group of bars on the right show the geometric mean performance for each configuration across all of the workloads.

We can make a number of observations from these results: 1) MBI-based profile guidance enhances static data placement, which can perform surprisingly well in some cases. For instance, some configurations of *imagick* and *qmcpack* obtain the full benefit of the MCDRAM with only a fraction of their data statically assigned to the upper tier. 2) High-quality tiering guidance is more important when the capacity of the upper tier is more constrained. On average, guidance improves performance by more than 20% over unguided first-touch when the upper tier capacity is limited to 12.5% of peak RSS, but by less than 10% when 50% of the application's data is able to fit in the upper tier. 3) In almost all cases, the thermos strategy obtains equal or better performance than the knapsack and hotset approaches. On average, thermos significantly outperforms the other approaches when the profile guidance uses additional call path context, regardless of the capacity of the upper tier. 4) Using call path context to distinguish allocation sites enhances data tiering guidance (note the different y-axis scales), and is particularly important for workloads such as *imagick* and *amg* that reach only a small number of allocation sites if context is not considered. For the remaining evaluation, we only consider executables that use call path cloning to distinguish allocation site context.

It is also important to note that the guided approach exhibits a small increase in total capacity compared to the default jemalloc allocator. For the thermos strategy, capacity increases by $< 5\%$, on average, with a maximum increase of 15% for *roms* with 50% upper tier capacity limit.

C. Automated Application Guidance with PEBS Profiling

While the above results show that automated application guidance can enhance hybrid data placement, the MBI-based profiling on which they rely requires a separate program run for each allocation site, and is not suitable for online applications. Next, we consider the use of low overhead PEBS-based profiling to guide static data placement, and compare its performance to the MBI-based tools.

The PEBS subsystem is able to control the amount of detail it collects by changing the rate by which it samples architectural events. To understand how this parameter affects our approach, we varied the sampling rate of our our PEBS-based profiling tool from once every 128 LLC misses up to once every 8,192 LLC misses, with steps increasing in powers of two.[3] Figure 5 displays the execution time of the PEBS-based profiling tool with different sampling rates relative to the default MCDRAM-only performance. Overall, we find that the performance overhead of the PEBS-based profiling tool is relatively low – between 22% and 32%, on average. While the sampling rate does have an impact, most of the overhead is due to contention caused by application threads attempting to allocate data from the same arenas. If we account for this cost, the overhead of the PEBS-based profiling alone is only 1.6% to 10.7%, on average.

Figure 6 shows the performance (execution time) impact of using PEBS-based profiling with different sampling rates to guide the thermos data placement strategy on our MCDRAM-DDR4 platform with an upper tier capacity limit of 25% of what is needed by the application. We find that, with only a few exceptions, the sampling rate does not have much impact on the performance of profile-guided data placement. This outcome suggests that, for online applications, the potential benefits of higher frequency sampling are not likely to outweigh its additional cost.

Next, we consider how the accuracy of the PEBS-based profiling affects its performance, and if there is any potential

[3]Further increasing the sampling rate past once every 128 LLC misses caused the system to become unstable and crash frequently.

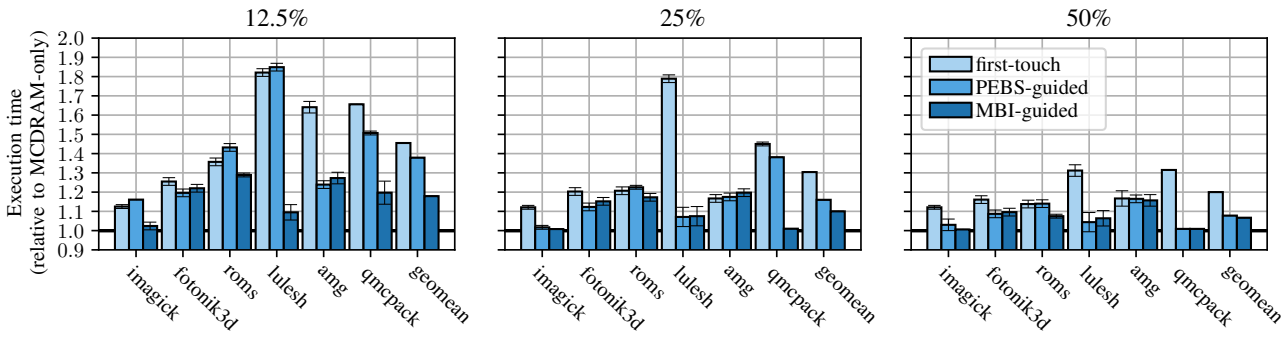

Fig. 7. Performance (execution time) of unguided, PEBS-guided, and MBI-guided strategies relative to MCDRAM-only (lower is better).

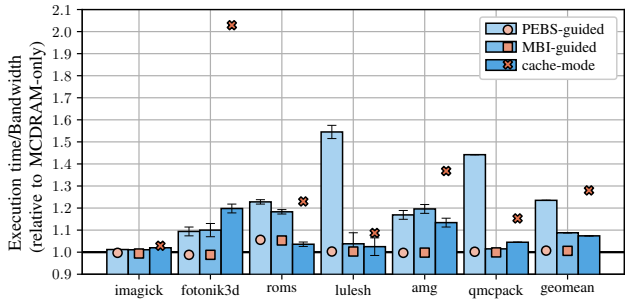

Fig. 8. Performance (execution time) (as bars) and total memory bandwidth (as markers) of PEBS- and MBI-guided runs, alongside cache-mode, relative to MCDRAM-only (lower is better).

for further improvement. Figure 7 compares the performance (execution time) of the best PEBS profile-guided configuration (with sampling rate=512) to MBI-guided placement and an unguided static first touch strategy. Both guided configurations use the thermos strategy and all results are shown relative to MCDRAM-only. The results show that the PEBS-guided strategy typically outperforms an unguided approach, but is still significantly slower than MBI-guided data placement. Interestingly, there is less difference between the performance of the PEBS- and MBI-guided runs when there is more space available in the upper tier. Further analysis with individual workloads indicates that the PEBS profiling does correctly distinguish some large allocation sites as cold, but it struggles to differentiate between hot and lukewarm data. Thus, there is still a need for more accurate bandwidth profiling with low enough overhead to be suitable for online applications.

D. Comparison with Hardware-Managed Caching

Lastly, we compare the performance of static data placement guided by application profiling to an adaptive, unguided approach that relies on hardware-managed caching. For these experiments, we prepare a "cache-mode" configuration that uses the KNL Hybrid memory mode [47] to exercise 25% (4GB) of the MCDRAM as a direct-mapped cache to application data on the lower tier. To provide an 'apples-to-apples' comparison with our approach, we ran the static guidance-based strategies alongside an otherwise idle process that reserves all but 4GB of the capacity in the MCDRAM tier.

Figure 8 displays the performance (as bars) and total memory bandwidth (as markers) of the PEBS-guided, MBI-guided, and cache-mode approaches, relative to the results with the MCDRAM-only configuration. Not surprisingly, cache-mode outperforms the static guidance-based approaches in some cases (*roms* and *amg*). The static approaches have no mechanism for changing data-tier assignments during the run, and can struggle with workloads that generate highly variable usage patterns. Despite this property, static placement with MBI-guidance either meets or exceeds the performance of cache-mode for 4 of our 6 workloads. Moreover, MemBrain achieves this performance without the need to migrate application data between tiers, and thus, presents an opportunity to increase memory efficiency. Overall, the PEBS- and MBI-guided strategies exhibit performance that is 16% and 1% worse than cache-mode, respectively, but generate 28% less memory bandwidth for our workloads.

VII. Future Work

Our results show that low-overhead architectural sampling is often not accurate enough to produce optimal data-tiering guidance. In the immediate future, we plan to conduct deeper analysis of the PEBS-based profiling approach, and will use the more accurate MBI-based profiles to characterize and, if possible, resolve issues that hinder its effectiveness. Later, we plan to build and integrate with MemBrain an online profiling tool that automates the production and application of memory usage guidance without prior profiling. We have also found that, even with highly accurate data-tiering guidance, some applications perform better with hardware-managed caching than with software-directed data placement. To facilitate the use of software guidance with existing hardware features, we plan to design and implement new data characterization tools that automatically identify objects and usage patterns that work well with hardware caching. Finally, while this study targeted two-level memory on the Intel® Xeon Phi, the source code analysis, profiling, and data management tools we have developed can be adapted for applications on any platform

with multiple memory tiers. In the future, we plan to modify our framework for use with other architectures and emerging memory technologies, including the Intel®-Micron® Hybrid Memory Cube [48] and Intel® 3D XPoint [49], and will explore the potential challenges and opportunities that arise from managing their data with MemBrain.

VIII. Conclusions

Emerging memory technologies are forcing systems to alter their data management strategies to take advantage of the different capabilities and performance provided by the new types of hardware. Most current strategies rely on source code modifications and/or hardware-based caching to adapt memory traffic and access patterns to heterogeneous memory devices. In this paper, we present MemBrain: a software-based data management framework that aims to address the limitations of current approaches and enable more efficient data placement on hybrid memories. MemBrain introduces new compilation, profiling, and runtime tools to automate the production and use of effective data tiering guidance for application software. Our evaluation, conducted on an Intel® Knight's Landing machine with MCDRAM and DDR4, quantifies the importance of profile accuracy, and demonstrates that our approach yields significant performance and efficiency improvements compared to current techniques.

Acknowledgements

This research is supported in part by the National Science Foundation under CCF-1619140, CCF-1617954, and CNS-1464288, as well as a grant from the Software and Services Group (SSG) at Intel® Corporation.

References

[1] I. C. Kristian Vatto and R. Smith, "Analyzing intel-micron 3d xpoint: The next generation non-volatile memory," http://www.anandtech.com/show/9470/intel-and-micron-announce-3d-xpoint-nonvolatile-memory-technology-1000x-higher-performance-endurance-than-nand, July 2015.

[2] A. Sodani, "Knights landing (knl): 2nd generation intel® xeon phi processor," in *Hot Chips 27 Symposium (HCS), 2015 IEEE*. IEEE, 2015, pp. 1–24.

[3] C. Cantalupo, V. Venkatesan, and J. R. Hammond, "User extensible heap manager for heterogeneous memory platforms and mixed memory policies," 2015.

[4] NVIDIA, "Gp100 pascal whitepaper," https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf, 2016.

[5] N. Agarwal, D. Nellans, M. Stephenson, M. O'Connor, and S. W. Keckler, "Page placement strategies for gpus within heterogeneous memory systems," *SIGPLAN Not.*, vol. 50, no. 4, pp. 607–618, Mar. 2015.

[6] 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*. ACM, 2016, p. 15.

[7] N. Agarwal and T. F. Wenisch, "Thermostat: Application-transparent page management for two-tiered main memory," in *Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '17. New York, NY, USA: ACM, 2017, pp. 631–644.

[8] I. B. Peng, R. Gioiosa, G. Kestor, P. Cicotti, E. Laure, and S. Markidis, "Rthms: A tool for data placement on hybrid memory system," in *Proceedings of the 2017 ACM SIGPLAN International Symposium on Memory Management*, ser. ISMM 2017. New York, NY, USA: ACM, 2017, pp. 82–91.

[9] H. Servat, A. J. Pea, G. Llort, E. Mercadal, H. Hoppe, and J. Labarta, "Automating the application data placement in hybrid memory systems," in *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, Sept 2017.

[10] M. Laghari and D. Unat, "Object placement for high bandwidth memory augmented with high capacity memory," in *29th Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, Oct 2017, pp. 129–136.

[11] T. C. Effler, A. P. Howard, T. Zhou, M. R. Jantz, K. A. Doshi, and P. A. Kulkarni., "On automated feedback-driven data placement in hybrid memories." in *LNCS International Conference on Architecture of Computing Systems (ARCS18)*, 2018.

[12] M. Meswani, S. Blagodurov, D. Roberts, J. Slice, M. Ignatowski, and G. Loh, "Heterogeneous memory architectures: A hw/sw approach for mixing die-stacked and off-package memories," in *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*, Feb 2015, pp. 126–136.

[13] M. Oskin and G. H. Loh, "A software-managed approach to die-stacked dram," in *2015 International Conference on Parallel Architecture and Compilation (PACT)*. IEEE, 2015, pp. 188–200.

[14] M. Giardino, K. Doshi, and B. H. Ferri, "Soft2lm: Application guided heterogeneous memory management," in *IEEE International Conference on Networking, Architecture and Storage (NAS), Long Beach, CA, USA, August 8-10, 2016*, 2016.

[15] Y. Li, S. Ghose, J. Choi, J. Sun, H. Wang, and O. Mutlu, "Utility-based hybrid memory management," in *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, Sept 2017.

[16] 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*, ser. SC '17. New York, NY, USA: ACM, 2017, pp. 58:1–58:14.

[17] X. Dong, Y. Xie, N. Muralimanohar, and N. P. Jouppi, "Simple but effective heterogeneous main memory with on-chip memory controller support," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society, 2010, pp. 1–11.

[18] G. H. Loh and M. D. Hill, "Efficiently enabling conventional block sizes for very large die-stacked dram caches," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2011, pp. 454–464.

[19] J. Sim, A. R. Alameldeen, Z. Chishti, C. Wilkerson, and H. Kim, "Transparent hardware management of stacked dram as part of memory," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-47. Washington, DC, USA: IEEE Computer Society, 2014, pp. 13–24. [Online]. Available: \url{http://dx.doi.org/10.1109/MICRO.2014.56}

[20] C. Chou, A. Jaleel, and M. K. Qureshi, "Cameo: A two-level memory organization with capacity of main memory and flexibility of hardware-managed cache," in *Proceedings of the IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-47. Washington, DC, USA: IEEE Computer Society, 2014, pp. 1–12.

[21] X. Yu, C. J. Hughes, N. Satish, O. Mutlu, and S. Devadas, "Banshee: Bandwidth-efficient dram caching via software/hardware cooperation," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-50 '17. New York, NY, USA: ACM, 2017, pp. 1–14.

[22] V. Young, P. J. Nair, and M. K. Qureshi, "Dice: Compressing dram caches for bandwidth and capacity," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ser. ISCA '17. New York, NY, USA: ACM, 2017, pp. 627–638.

[23] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr., "Exokernel: an operating system architecture for application-level resource management," *SIGOPS Oper. Syst. Rev.*, vol. 29, no. 5, pp. 251–266, Dec. 1995.

[24] A. Belay, A. Bittau, A. Mashtizadeh, D. Terei, D. Mazières, and C. Kozyrakis, "Dune: safe user-level access to privileged cpu features," in *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, ser. OSDI'12. USENIX Association, 2012, pp. 335–348.

[25] G. Banga, P. Druschel, and J. C. Mogul, "Resource containers: a new facility for resource management in server systems," in *Proceedings of the third symposium on Operating systems design and implementation*, ser. OSDI '99. USENIX Association, 1999, pp. 45–58.

[26] A. Kleen, "A numa api for linux," *SUSE Labs white paper*, August 2004.

[27] M. R. Jantz, C. Strickland, K. Kumar, M. Dimitrov, and K. A. Doshi, "A framework for application guidance in virtual memory systems," in *Proceedings of the 9th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE '13, 2013, pp. 155–166.

[28] N. Beckmann and D. Sanchez, "Jigsaw: Scalable software-defined caches," in *Parallel Architectures and Compilation Techniques*, ser. PACT '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 213–224.

[29] M. R. Jantz, F. J. Robinson, P. A. Kulkarni, and K. A. Doshi, "Cross-layer memory management for managed language applications," in *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA 2015. New York, NY, USA: ACM, 2015, pp. 488–504.

[30] M. B. Olson, J. T. Teague, D. Rao, M. R. JANTZ, K. A. Doshi, and P. A. Kulkarni, "Cross-layer memory management to improve dram energy efficiency," *ACM Trans. Archit. Code Optim.*, vol. 15, no. 2, pp. 20:1–20:27, May 2018.

[31] R. Guo, X. Liao, H. Jin, J. Yue, and G. Tan, "Nightwatch: integrating lightweight and transparent cache pollution control into dynamic memory allocation systems," in *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, 2015.

[32] M. Dashti, A. Fedorova, J. Funston, F. Gaud, R. Lachaize, B. Lepers, V. Quema, and M. Roth, "Traffic management: a holistic approach to memory placement on numa systems," in *ACM SIGPLAN Notices*, vol. 48, no. 4. ACM, 2013.

[33] A. Mukkara, N. Beckmann, and D. Sanchez, "Whirlpool: Improving dynamic cache management with static data classification," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '16. New York, NY, USA: ACM, 2016, pp. 113–127.

[34] P.-A. Tsai, N. Beckmann, and D. Sanchez, "Jenga: Software-defined cache hierarchies," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ser. ISCA '17. New York, NY, USA: ACM, 2017, pp. 652–665. [Online]. Available: \url{http://doi.acm.org/10.1145/3079856.3080214}

[35] A. J. Pea and P. Balaji, "Toward the efficient use of multiple explicitly managed memory subsystems," in *2014 IEEE International Conference on Cluster Computing (CLUSTER)*, Sept 2014, pp. 123–131.

[36] M. R. Jantz, F. J. Robinson, and P. A. Kulkarni, "Impact of intrinsic profiling limitations on effectiveness of adaptive optimizations," *ACM TACO*, 2016.

[37] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, ser. CGO '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 75–. [Online]. Available: http://dl.acm.org/citation.cfm?id=977395.977673

[38] J. Evans, "A scalable concurrent malloc (3) implementation for freebsd," 2006.

[39] "pthreads - posix threads, linux programmer's manual." [Online]. Available: http://man7.org/linux/man-pages/man7/pthreads.7.html

[40] "mbind - set memory policy for a memory range." [Online]. Available: http://man7.org/linux/man-pages/man2/mbind.2.html

[41] "pagemap, from the userspace perspective." [Online]. Available: https://www.kernel.org/doc/Documentation/vm/pagemap.txt

[42] SPEC, "Spec cpu 2017," 2017. [Online]. Available: https://www.spec.org/cpu2017/

[43] LLNL, "Coral benchmark codes," https://asc.llnl.gov/CORAL-benchmarks, 2014.

[44] "Flang," https://github.com/flang-compiler/flang.

[45] A. Georges, D. Buytaert, and L. Eeckhout, "Statistically rigorous java performance evaluation," in *Object-oriented programming systems, languages, and applications*, ser. OOPSLA '07, 2007, pp. 57–76.

[46] T. Willhalm, R. Dementiev, and P. Fay, "Intel performance counter monitor - a better way to measure cpu utilization," https://software.intel.com/en-us/articles/intel-performance-counter-monitor, 2012.

[47] Intel, "Intel xeon phi x200 processor - memory modes and cluster modes: Configuration and use cases," https://software.intel.com/en-us/articles/intel-xeon-phi-x200-processor-memory-modes-and-cluster-modes-configuration-and-use-cases, December 2015.

[48] H. M. C. Consortium, "Hmc specification 2.1," 2014.

[49] Intel, "3d xpoint," http://www.intel.com/content/www/us/en/architecture-and-technology/3d-xpoint-unveiled-video.html, 2016.