ecoHMEM: Improving Object Placement Methodology for Hybrid Memory Systems in HPC

Marc Jordà, Siddharth Rai, Eduard Ayguadé, Jesús Labarta and Antonio J. Peña
Barcelona Supercomputing Center (BSC) Email: first.last@bsc.es

***Abstract*—Recent byte-addressable persistent memory (PMEM) technology offers capacities comparable to storage devices and access times much closer to DRAMs than other non-volatile memory technology. To palliate the large gap with DRAM performance, DRAM and PMEM are usually combined. Users have the choice to either manage the placement to different memory spaces by software or leverage the DRAM as a cache for the virtual address space of the PMEM. We present novel methodology for automatic object-level placement, including efficient runtime object matching and bandwidth-aware placement. Our experiments leveraging Intel® Optane™ Persistent Memory show from matching to greatly improved performance with respect to state–of–the–art software and hardware solutions, attaining over $2x$ runtime improvement in miniapplications and over 6% in OpenFOAM, a complex production application.**

***Index Terms*—data placement, hybrid memory systems, optane**

I. Introduction

Large RAM spaces help mitigate overheads inherent to distributed computing. One of the biggest challenges to build faster supercomputers is to improve energy efficiency; the main memory, traditionally built upon DRAM technology, is among the biggest energy consumers of high-performance computing (HPC) systems, posing over 25% of the system energy consumption [24]. DRAM technology evolution shows a flattening of the power consumption curve in the last years, being a factor of 1.5 per generation from 2000 to 2010, while being only a factor of 1.2 from 2010 to 2018 [37]. Since providing larger memory spaces is non-viable by means of top-performance memory technology only, due to energy consumption and dissipation constraints, HPC vendors are incorporating different kinds of memory devices within their compute systems. E.g., scratchpad or high-bandwidth memories bring high access speeds and reduced spaces; NVRAMs are energy-friendly devices providing slow data access rates; ECC-enabled memories deliver fault-tolerance at the cost of some performance and space overhead; and compute-capable memories offer processing–in–memory features.

In the past, deep memory hierarchies have been proposed to try to address the memory wall problem. Current trends, however, advocate for bringing the different memory subsystems as first-class citizens, building a set of explicitly-addressable subsystems that must be managed by software.

Recent byte-addressable PMEM technologies, such as the Intel® Optane™ Persistent Memory (PMem), offer dense bit packaging, attaining capacities of up to 6 TB per compute node and performance much closer to DRAM than other non-volatile memory technologies. To palliate the still severe gap with DRAM access latencies and bandwidth, which is exacerbated at non-sequential access patterns due to large access block sizes, DRAM and PMEM are usually combined. Users can either manage placement to different memory spaces manually (potentially assisted by research-class middleware), or leverage the DRAM as a hardware-managed inclusive cache for the virtual address space of the PMEM.

Deciding what data to host in each memory subsystem is nontrivial and poses notable performance implications. Whereas hardware-based mechanisms lack flexibility and are not always efficient, yielding inconsistent performance, software-based approaches pose management overheads and often require expert knowledge and intrusive code changes.

While kernel-based approaches at page granularity are reactive, we advocate for a proactive solution based on offline data-oriented profiling at memory object granularity. Recent research has proposed the use of secondary memory subsystems for fault-tolerance purposes [41] or to host selected data objects based on domain knowledge [31]; generic approaches, however, have not demonstrated viability beyond simple use cases. In this paper, we propose novel methodology for data placement in hybrid memory systems at object level granularity, based on offline data-oriented profiling and runtime memory allocation interception, that yields efficient executions and simplified workflows beyond state of the art solutions. Our implementations are released open source as ecoHMEM [1].

The contributions of this paper are summarized as follows: (1) we propose novel profiling–runtime object matching methodology to greatly improve runtime performance with respect to previous work based on call-stack comparison; (2) we provide a novel object placement algorithm to address the inefficiencies of previous work; and (3) we present, for the first time in the literature, runtime benefits in complex codes. Addressing the shortcomings of a previous solution with novel methodology, we attain performance in pair with the state of the art of data placement at object granularity for hybrid memory systems while notably simplifying the workflow.

The rest of the paper is organized as follows. Section II provides the necessary background to understand the contributions of this paper. Section III reviews related work. Section IV discusses the basic methodology leveraged in this paper, whereas sections V, VI and VII deepen into our contributions. Section VIII presents the analysis of our evaluations and Section IX concludes this manuscript.

© 2022 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes,creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works. DOI 10.1109/CLUSTER51413.2022.00040

II. Background: Intel Optane Persistent Memory

This technology, first released in 2019, offers high-density, byte-addressable persistent memory in the standard DIMM form factor. Currently, the largest capacity available is 6 TB on dual-socket systems. These support up to 2666 MT/s. Compared to DDR4 DRAM, PMEM latencies increase 2x–6x for reads and 6x–30x for writes depending on the access pattern, whereas bandwidth decreases around 75% for reads and 90% for writes in Intel's first-generation implementation [14]; the recently-released second generation provides around 40% additional performance. We note that in this work we use the PMem as a RAM subsystem, and hence the persistence capabilities are not leveraged.

PMem supports two different modes of operation. In *memory mode*, the DRAM acts as an inclusive cache for the PMem, managed directly by the memory controllers. The PMem is transparently exposed as the main system memory to the operating system. Although there is no official information on its internals, it is considered to be a direct-mapped, write-back cache [13], [18]. This mode enables users to exploit the high capacity of PMem DIMMs without the need for software changes; however, for many workloads, such as those featuring non-sequential access patterns or those pathological cases suffering from numerous conflict misses, the DRAM cache is not able to hide the increased latencies of PMem.

In the *app direct* mode, the DRAM is used as the system memory, while the PMem is exposed through a Linux char or block device. Applications may allocate buffers in PMem using libraries like `memkind` [16], and are responsible to decide which data is stored in each memory subsystem. This places a burden on application developers, which may be alleviated by a middleware layer to manage the placement.

III. Related Work

Heterogeneous memory systems are recently becoming a reality in the field of HPC and currently a hot topic in the community. The first paper proposing this scenario and methodology to address this situation, leveraged an emulated system based on a Valgrind tool [28] and cache simulation [29]. Early research focused on the discontinued Intel® Xeon Phi™ x200 systems (KNL); Laghari and Unat [22] proposed methodology based on object-level read and write information extracted from hardware counters and simulations, implementing a knapsack approach for distribution; Servat *et al.* [35] developed a similar knapsack-based approach, leveraging only read counters extracted from hardware sampling, but providing an allocation runtime interposer to prevent the need for the programmer's intervention. Dulloor *et al.* [7] also demonstrated that a careful placement of data-structures across memory tiers is necessary to incorporate NVM into the processor memory hierarchy. Plenty of methodology has been proposed for this purpose, including profiling based on sampling of hardware counters [21], [35], runtime-assisted profiling [3], or object-level placement granularity [8], [29] versus its page-level counterpart [39], [40]. Wen *et al.* [38] proposed ProfDP, a differential profiling mechanism (requiring three profiling runs) to estimate per-object latency and bandwidth sensitivity, and decide a priority to guide placement decisions. This proposal, requiring manual source code modification, neither considers memory capacity or instantaneous memory bandwidth usage nor provides any placement algorithm that can optimize system performance for a given system configuration. Since version 5.5, the Linux kernel includes support to expose PMem *devdax* devices as NUMA nodes. On top of this support, Intel [10] and other previous works (e.g., [25]) provide kernel-level solutions that enable automatic page migration between DRAM and PMem NUMA nodes. Kernel-level page migration approaches are orthogonal to our application-level design, and may be combined to leverage an initial proactive object placement provided by the latter along with reactive runtime page migration capabilities provided by the former. This is an approach that we intend to explore in future work, since it requires completely different object weighting heuristics, in order to optimize for initial placement instead of optimizing for the entire execution. There are also works leveraging specific domain knowledge to improve upon application-agnostic proposals [6], [31].

Extending the afore-mentioned framework proposed for the KNL [35], we propose the first framework that, operating on unmodified binaries, is able to cope with fully-featured production applications, providing efficient executions in modern hybrid memory systems. While other similar works to ours required either manual source code modification or, at best, source–to–source compilation phases to link against custom allocation libraries [26], [27], we provide a runtime-based solution to intercept unmodified allocation calls, developing a novel object matching approach that prevents high object identification overheads. We also provide a new object distribution algorithm that is able to adapt to the complexity of the access patterns incurred by production-level applications.

IV. Methodology

Our methodology (see Figure 1) is based in offline data-oriented profiling at object granularity along with a runtime allocation interposer, as described in [35] for the KNL platform. The workflow starts by profiling a *production-ready binary* to generate a trace-file and extract the performance metrics around the data-objects created during the application execution. Next, a profile analyzer reports an optimized object distribution across memory subsystems. During runtime, an allocation interposer substitutes heap memory allocations, honoring the previous report on the *same optimized binary* used during the profiling stage.

This section provides a brief overview of the methodology we build on, as originally devised for the KNL, whereas sections V, VI and VII discuss our key contributions to support the specifics of PMEM and remove performance bottlenecks.

A. Data-Oriented Profiling

Data-oriented profiling is supported by Extrae [4], an open-source instrumentation package that is injected into the application by means of the `LD_PRELOAD` mechanism, hence enabling the monitoring of unmodified optimized binaries.

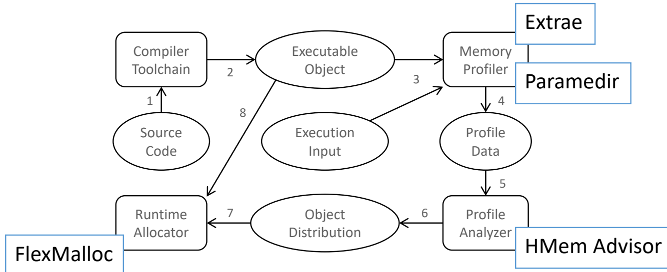

Fig. 1. Workflow. Note that this picture shows the name of our implementations of the different components instead of the original names in [35].

A configuration file specifies the performance metrics to be collected, preventing unnecessarily large trace-files.

We use Extrae to collect information regarding data objects and their related memory references, through instrumentation on allocation, reallocation and deallocation routines [33], [34]. This includes the size, the frames of the invoking call-stack and the returned address. Since most Linux systems implement Address Space Layout Randomization (ASLR) security techniques, the call-stack addresses are subject to change from run to run, and thus, Extrae translates the addresses (frames) of the call-stack to a proper code location identification.

Memory references are captured using the Precise Event-Based Sampling (PEBS) mechanism [17]. On our target machine, Extrae uses the counter `MEM_LOAD_RETIRED.L3_MISS` to sample load instruction misses in the last-level cache (LLC). These have associated a data-linear address that is matched to instrumented data-objects.

Once the application execution has finished and generated a trace-file, we leverage Paramedir, the command-line tool to explore Extrae trace-files, to extract the largest allocation observed and the number of LLC misses for each allocation.

B. Placement Optimizer

The Heterogeneous Memory Advisor computes an optimized object distribution among the available memory subsystems. The idea is to reduce the CPU stall cycles due to memory accesses by placing the most accessed objects in the memory subsystem with highest bandwidth.

To compute the object distribution, the Advisor uses the object-differentiated memory access data extracted by Paramedir. Since the latency of memory accesses that hit in any level of the cache hierarchy is not affected by the backing memory subsystem, we are only interested in LLC misses.

The base algorithm of the Advisor is based on a greedy relaxation of the 0/1 multiple knapsack problem, where the memory objects have to be distributed among the available memory subsystems (the knapsacks) by solving a knapsack problem for each of them, in descending order of their provided performance. The memory objects' value is the ratio of cache misses divided by object size, to represent the density of misses per object. Each memory subsystem features its own coefficients representing read latencies, specified in a configuration file, which enables the use of the framework in systems with different heterogeneous memory configurations.

C. Runtime Allocation

FlexMalloc [32] is an interposition library that reads the indications obtained by the Advisor and drives the allocations of application data-objects into different memory subsystems at runtime. The library sits on top of a number of heap managers (each targeting a specific memory subsystem) and forwards the memory management calls invoked by the application to a specific heap manager honoring the report and a given memory-system configuration (such as, e.g., memkind, POSIX malloc or libnuma's numa_alloc). For the experimentation in this paper, memkind is used for PMem allocations, whereas POSIX malloc is leveraged to target the DRAM space.

FlexMalloc supports a fallback memory subsystem for data-objects not listed in the Advisor report. This subsystem (usually the largest) will also be used if other memory subsystems run out of available space.

We note that FlexMalloc may alter performance because of different heap management routine characteristics. E.g., NUMA affinity is determined on a first-touch policy in Linux (*i.e.*, memory pages are allocated on the NUMA domain from the CPU that first touches the pages). However, when using Intel Optane memory on top of memkind, NUMA affinity is specified for the whole data-object at allocation point.

V. Store Operations

Previous works [29], [35] derived object cost heuristics exclusively from load operations, assuming that buffered write-through cache implementations would absorb the impact of most store operations. However, non-volatile memories have brought RAM spaces in which store operations are particularly penalized. This led us to consider store operations in our heuristics. However, since PEBS store counters for LLC misses are non-existent (due to technical reasons beyond the scope of this paper), we explore the use of L1D store misses to capture the effect of stores in cost heuristics.

To implement this feature, we configured Extrae to sample the `MEM_INST_RETIRED.ALL_STORES` hardware counter, which targets all store instructions. To control the influence of loads and stores in the cost heuristics, we implemented the Advisor's algorithms to use two separate coefficients to weight the data from the profiling trace. Accordingly, the Advisor's configuration file requires now separate load and store coefficients per memory subsystem.

VI. Binary Object Matching (BOM)

Prior to this work, Extrae supported translating the frames into pairs consisting of file and line numbers with the help of the binutils package [9] and the debug information contained on the binary. While this approach was useful for a later source-code inspection, during the evaluation of this framework we observed (1) severe runtime overhead when parsing large binaries or long call-stacks, and (2) additional considerable memory used when loading debugging information. In order to circumvent these issues, we have modified Extrae to translate the call-stack frames into a binary form consisting of the binary object that contains the frame address and the offset from the base address of the binary object (see Table I).

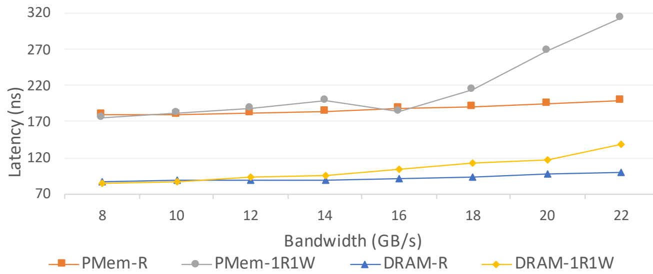

Fig. 2. Bandwidth vs. latency with read-only (R) and read-write (1R1W) memory traffic for DDR4 DRAM and Intel PMem using MLC [15].

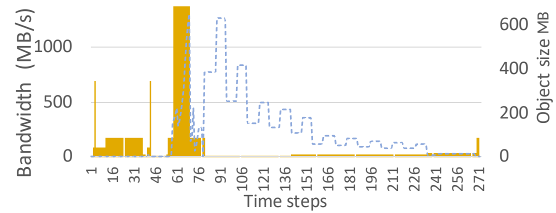

Fig. 3. Bandwidth consumption as objects are allocated for LULESH.

This novel approach reduces the runtime overhead because it no longer needs to translate the call-stack frames into human-readable source code locations. Furthermore, it removes the need for embedding debug information into the binary at compile time, which was mandatory in previous approaches.

When leveraging BOM, during the process initialization the library obtains the base address where each shared-library is loaded in memory, and calculates the absolute addresses for each frame of every call-stack. When the process invokes a heap memory call, it is intercepted by FlexMalloc, where the routine parameters and the call-stack are captured. If the call-stacks in the report are in human-readable format, the library has to translate the call-stack addresses into human-readable (with the help of binutils) and then compare the translated call-stack with those listed in the report. Conversely, if the call-stacks in the report are in binary format, the library only has to compare the captured call-stack addresses with the absolute call-stack addresses calculated during initialization. While using human-readable format suffers from the expense of translating the call-stack, which may be significant on large binaries, requiring multiple string comparisons, the new BOM approach requires only a number of address comparisons.

VII. Memory Bandwidth Aware Object Placement

Due to limited capacity, even with dense packing (in terms of LLC misses) of objects into DRAM, a large number of off-chip accesses still target PMem and thus suffer from its idiosyncratic behavior. Since memory access latency is a complex function of size, lifetime, and object's memory bandwidth demand, it varies widely across objects. Figure 2 shows the variation in latency as bandwidth is scaled from 8 GB/s to 22 GB/s for both DRAM and PMem. As we see, at low bandwidth, read and write latency variation is not noticeable for both memories; however, the gap widens quickly as bandwidth increases. At 22 GB/s, PMem costs 2.3x higher latency than DRAM. If this difference is not considered, a simple placement decision may well degrade performance.

Consider an application that consumes high memory bandwidth during 20% of its execution time and in the remaining 80%, its memory consumption remains low. For this application, we want to place two objects, **A** and **B**, of the same size **S** in the memory system composed of DRAM and PMem. The size of the objects is such that only one of these may fit in DRAM at a time. **A** spends 80% of its lifetime in low bandwidth region and the remaining 20% in high bandwidth region. **B** spends 100% of its lifetime in high-bandwidth region (thus, its lifetime is 20% of that of object **A**). Both incur **M** off-chip accesses. If bandwidth variation during the object's lifetime is not considered and uniform latency is assumed, a placement strategy relying on access density may well favor **A** over **B** in DRAM because both objects will be indistinguishable (same access/byte). However, this placement will be counterproductive, causing higher overall access latency. Using the data shown in Figure 2 (low/high latency for DRAM: 90/117 ns; for PMem: 185/239 ns):

*a) Strategy 1 — **A** is allocated in DRAM and **B** in PMem:* Since **A** spends 80% of its time in low bandwidth region and 20% of its time in high bandwidth region, on average, it will experience 90 ns and 117 ns during 80% and 20% of the execution time, respectively. **B**, on the other hand, will spend 239 ns throughout its lifetime. This results in *334 ns*.

*b) Strategy 2 — **A** is allocated in PMEM and **B** in DRAM:* In this case, **A** spends 185 ns during 80% of its lifetime and 239 ns during the remaining 20% of its lifetime. **B**, on the other hand, spends, on average, 116 ns throughout its execution. This results in *311 ns*.

Compared to Strategy 2, the traditional strategy leads to 7% increase in latency. Compared to the low-bandwidth region, the increase in latency during the high-bandwidth region is much higher for PMem and the object lifespan in different regions poses strong implications on its overall access latency.

A. Case Study: LULESH

We examine LULESH [20], an HPC miniapplication, to understand its memory bandwidth requirements. We further draw insights from this data to develop a bandwidth-aware object placement strategy. To this end, we execute LULESH on the setup presented in Section VIII. We configured PMem in App. Direct mode and used the access density based classification to place objects as described in Section IV-B. We compiled LULESH using icc 19.05 with O3 optimization and executed 25 iterations with 8 ranks and 3 OpenMP threads/rank.

The dotted graph in Figure 3 shows the change in PMem bandwidth consumption in LULESH during one of its recurring execution phases as objects are allocated and deallocated, with bandwidth consumption on the primary y-axis. In the beginning, PMem bandwidth consumption is considerably low; however, as execution progresses, more objects are allocated in PMem and its bandwidth consumption starts increasing, attaining its maximum at 1.3 GB/s, to gradually diminish by the end of the phase. To obtain a complete picture, we further

TABLE I
EXAMPLES OF THE SUPPORTED CALL-STACK FORMATS, IDENTIFYING ALLOCATION POINTS AND THE ASSIGNED MEMORY SUBSYSTEM

Human-Readable	`new_op.cc:50 > lulesh-init.cc:500 > lulesh-init.cc:130 > lulesh.cc:2716 > libc-start.c:342 @ posix` `new_op.cc:50 > lulesh-init.cc:82 > lulesh.cc:2716 > libc-start.c:342 @ memkind/pmem`
BOM	`lulesh2.0!004029df > lulesh2.0!00402979 > lulesh2.0!00402549 > libc-2.26.so!0002103a > lulesh2.0!004020ba @ posix` `lulesh2.0!00402979 > lulesh2.0!00402549 > libc-2.26.so!0002103a > lulesh2.0!004020ba @ memkind/pmem`

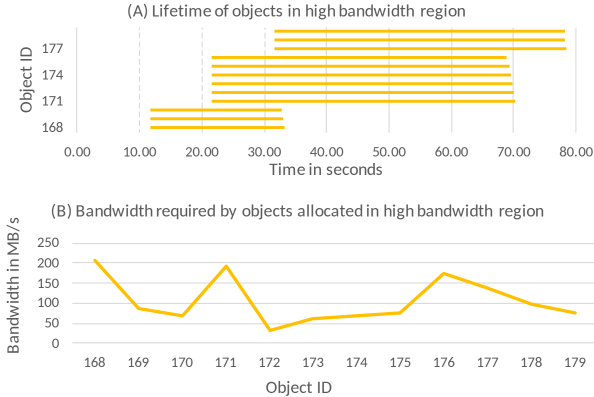

Fig. 4. Lifetime and bandwidth use of objects in high bandwidth region.

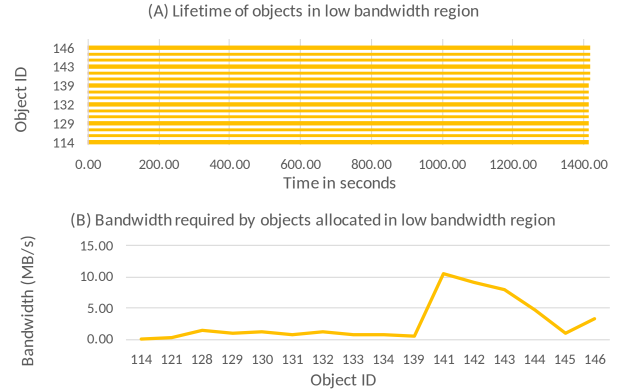

Fig. 5. Lifetime and bandwidth use of objects in low bandwidth region.

complement the bandwidth consumption plot with the memory objects using solid bars, with object size shown on the secondary y-axis. As the figure shows, the allocation size varies widely across objects, spanning from a few KB to hundreds of MB. Nonetheless, most of the large allocations occur at the start of the phase while, in the remaining part, relatively smaller objects are allocated. In the part of the phase where PMem bandwidth consumption is low, most of the allocations fall into DRAM, whereas, in the part where the bandwidth consumption is high (while allocation is happening), most of the allocations are directed to PMem. However, in the rest of the phase (where small allocations happen that can be cached in the upper part of the memory hierarchy), bandwidth consumption primarily depends on the lifetime and bandwidth requirement of the larger objects allocated earlier.

To gain understanding of the lifetime and bandwidth consumption of the objects allocated in different memory subsystems, we profiled simultaneously-living objects. The top panel (A) of Figure 4 shows the lifetime of PMem objects as a bar chart. In this chart, the left end of the bar denotes the allocation and the right end denotes the deallocation time. On average, an object resides for about 18 seconds, which is about 25% of the execution phase. The shortest living object experiences a lifetime of 8 seconds and the longest, 27 seconds. The bottom panel (B) of the figure shows bandwidth consumed by each of these objects. On average, these consume 93 MB/s of memory bandwidth, where highest (206 MB/s) and lowest (33 MB/s) bandwidth is consumed by objects 168 and 172, respectively. This data clearly shows that a small number of objects contribute significant amount of PMem bandwidth and these remain alive for a fraction of an execution phase. If those objects can be moved to DRAM, a large amount of PMem bandwidth can be released.

To understand the state of DRAM objects that may free the space for the PMem objects, Figure 5 shows similar data for DRAM objects. On average, these objects remain alive for 23 minutes, close to the overall execution time. Panel B further shows the bandwidth consumed by each of these objects. Despite all objects remaining alive for similar amount of time, their bandwidth consumption experiences wide variation. On average, an object consumes slightly above 1 MB/s of bandwidth, from a minimum of 50 KB/s to a maximum of 10.5 MB/s. Nonetheless, compared to PMem, objects in DRAM exhibit very small amount of bandwidth consumption (the peak consumption is less than the minimum consumed per object in PMem). *In summary, this data shows that an access density based object placement algorithm in practice may place objects with low bandwidth requirement into DRAM, whereas some of the most bandwidth demanding objects are placed into PMem. Moreover, it is possible to allocate high bandwidth objects into DRAM if some of the less demanding objects are moved to PMem and in this way substantial improvement in PMem bandwidth consumption may be attained, leading to higher overall performance.*

B. Memory Bandwidth Aware Object Placement Algorithm

This algorithm receives as input a set of objects already classified for placement in DRAM or PMem using our access density based algorithm and further divides these into groups using additional criteria. In particular, we consider bandwidth consumption, memory allocation pattern (single allocation/ multiple allocation), and access type (read-only, read-write).

1) Step 1 — Categorization: The objective is to find objects currently in DRAM that experience low bandwidth demand. These may be either moved to PMem directly or act as replacement candidates for other objects currently assigned to

TABLE II
BANDWIDTH OF OBJECTS SHOWN IN FIGURES 4 AND 5

Object ID	Allocation BW			Execution BW		
	B_{low}	B_{mid}	B_{high}	B_{low}	B_{mid}	B_{high}
114–134	T	F	F	T	T	T
139–146	F	T	F	T	T	T
168–179	F	F	T	F	F	T

TABLE III
AVERAGE NUMBER OF ALLOCATIONS FOR OBJECTS IN FIGURES 4 AND 5

Object ID	Allocations/Object	Lifetime (s)
114–146	1	1411
168–179	200	17

PMem. It also identifies objects in PMem that experience high bandwidth demand and would benefit if placed into DRAM.

Our experiments show that although long living objects experience varied bandwidth demands throughout their lifetime, short living objects primarily live in the bandwidth region of their allocation. Moreover, object's allocation count strongly correlates with its movement across bandwidth regions. Table II presents the bandwidth experienced by the objects shown in figures 4 and 5 during their lifetime. Objects with similar characteristics are grouped in rows; columns B_{low}, B_{mid}, and B_{high} correspond to regions with demand <20%, 20–40%, and >40% of the peak, respectively. We have also separated bandwidth demand at (nearby) allocation and (rest of) execution times. The cell is set to *true* (T) or *false* (F), reflecting whether an object belongs to a particular region.

We see two clear trends: while objects 114–134 and 139–146 experience different bandwidth demand during their lifetime than the demand at the time of their allocation, other objects stay in the region of their allocation throughout their lifetime. We further observe that object's bandwidth region, in fact, bears a strong correlation with its allocation frequency. Table III shows the allocation count and the lifetime of the objects shown in Table II. The objects that live in more than one bandwidth region are allocated and deallocated fewer times and live longer, whereas objects that stay in one bandwidth region for the entire execution are allocated and deallocated frequently. This implies that, although exactly associating a bandwidth region to objects with **few allocations** is difficult, objects with **high allocation count** usually live for short duration and most probably stay within the region of their allocation. Thus, these objects may be categorized as low or high bandwidth objects based on the region of their allocation, which forms our second classification criterion.

Our algorithm classifies objects into three different groups: Fitting, Streaming-D, and Thrashing. Fitting are those DRAM objects that are allocated less than T_{ALLOC} times; thus, these potentially feature long lifetime and experience bandwidth demand below $T_{PMEMLOW}$ at allocation. Hence, these, despite being allocated in the low bandwidth region, may experience different bandwidth demand during the rest of their lifetime. The DRAM objects with more than T_{ALLOC} allocations and bandwidth demand below $T_{PMEMLOW}$ are classified as Streaming-D. These objects feature a high number of allocations and only a few of these live simultaneously; thus, their lifetime is short and tend to stay within the allocation bandwidth region. Similarly, PMem objects allocated more than T_{ALLOC} times and featuring bandwidth demand above $T_{PMEMHIGH}$ are classified as Thrashing. We summarize our classification criterion in Table IV. Based on empirical observations, we set $T_{ALLOC} = 2$; $T_{PMEMHIGH} = 40\%$ and $T_{PMEMLOW} = 20\%$ of peak PMEM bandwidth.

TABLE IV
CRITERIA FOR OBJECT CLASSIFICATION

Initial Memory	Category	Description
DRAM	Fitting	DRAM object with less than T_{ALLOC} allocations and PMem bandwidth below $T_{PMEMLOW}$
	Streaming-D	DRAM object having no writes with more than T_{ALLOC} allocations and incurring bandwidth demand below $T_{PMEMLOW}$
PMEM	Thrashing	PMem object with more than T_{ALLOC} allocations and PMem bandwidth above $T_{PMEMHIGH}$

2) Step 2 — Placement: This step assigns objects categorized during Step 1 to DRAM and PMEM. Algorithm 1 presents the pseudocode for our placement logic, considering inputs pre-placed as shown in Table IV. All objects in the Streaming-D group are directly moved to PMEM, since that releases DRAM capacity that may be used to move objects from PMEM to DRAM. Next, objects in the Thrashing category are sorted first by their bandwidth consumption and then by their allocation and deallocation time and a replacement object is searched from the Fitting category.

Algorithm 1: Placement in DRAM and PMEM

```
Result: Set of objects classified into DRAM and PMEM;
Input: Objects in Fitting, Thrashing, and Streaming-D;
for All objects in Streaming-D do
    Set object allocation as PMEM;
end
for Each object in Thrashing do
    Set object2 as smallest number in Fitting that can
      accommodate object for its entire lifetime;
    if object2 found then
        Set object allocation as DRAM;
        Set object2 allocation as PMEM;
    end
end
```

To implement this feature within the Advisor, the output of the post-processing stage adds information on allocation and deallocation timestamps for all dynamic memory allocations in the trace, which was not required in simpler heuristics. Bandwidth consumption is derived from load and store hardware counters (as described in sections IV-A and V), divided by object's lifetime.

VIII. EVALUATION

For our evaluation we used a combination of codes, ranging from proxy applications demonstrating widespread scientific

TABLE V
CHARACTERISTICS OF THE APPLICATIONS USED IN THE EXPERIMENTAL EVALUATION

	MiniFE	MiniMD	LULESH	HPCG	CloverLeaf3D	LAMMPS	OpenFOAM
Version	2.2.0	2.0	2.0.3	3.1	1.2 beta	Stable_Oct20	v1906
MPI Ranks/Threads	12/2	12/2	8/3	6/4	24/1	12/2	16/1
Input Size	(400,400,400)	t=2 s=224	-p i=10 s=224	(192,192,192) rt=0	(512,512,512)	var=(8,8,8) rhodo.scaled 25 it.	depth charge 3D (240,480,240)
Memory High-Water Mark (MB/Rank)	1989	2196	10658	6414	1467	4240	3360

computation problems, to production-ready scientific simulation applications (see Table V):

- **MiniFE [11]:** A proxy application for unstructured implicit finite element codes.
- **MiniMD [11]:** A proxy application for parallel molecular dynamics simulation of a Lennard-Jones or EAM system.
- **Livermore Unstructured Lagrange Explicit Shock Hydrodynamics (LULESH):** Proxy application for simplified 3D Lagrangian hydrodynamics on unstructured mesh.
- **High Performance Conjugate Gradient (HPCG) [12]:** Benchmark based on an additive Schwarz, symmetric Gauss-Seidel preconditioned conjugate gradient.
- **CloverLeaf3D [36]:** Lagrangian-Eulerian hydrodynamics
- **Large Atomic/Molecular Massively Parallel Simulator (LAMMPS) [30]:** Molecular dynamics for materials.
- **OpenFOAM [19]:** Computational fluid dynamics (CFD) simulator widely used in engineering and science. Our use case is a 3D compressible fluid simulation.

The experiments were performed in a machine equipped with 2 Intel® Xeon® Platinum 8260L processors running at a nominal frequency of 2.3GHz, 4×8 GB DDR4 DIMMs and 12×512 GB Intel® Optane™ PMem 100 series DIMMs running at 2666 MT/s. The ratio of DRAM to PMem is intentionally lower than the recommended by the vendor, advocating for more energy friendly and cost-efficient system configurations [23]; results with only 4 PMem DIMMs are also presented to assess the impact of this ratio. The server runs Linux Fedora 27 (kernel 4.18.8) and we used the Intel compiler and Intel MPI version 2019.5.281. The object placement strategy is released as ecoHMEM v1.0 [1].

We compare our results with the server configured in memory mode (*baseline*) and with two other state–of–the–art approaches: (1) a kernel-level online page migration implementation, version tiering-0.71 of the experimental line of Linux kernels developed at Intel [10]; and (2) a state–of–the–art user-level data distribution approach, ProfDP [38], which uses a profile–analyze–run approach similar to our framework, built on top of HPCToolkit [2]. While reproducing the ProfDP workflow for our benchmark applications, we faced a few technical difficulties; since ProfDP is currently not integrated in the main HPCToolkit branch, we used the *datacentric-master* branch (the ProfDP-related branch with most recent activity). However, we could not manage to obtain the ProfDP specific metrics from the tools and we computed those following the formulas presented in [38], using profiling data obtained with the HPCToolkit profiler configured with the flags required to collect the data needed by ProfDP. Besides this, we faced another question not addressed in the ProfDP paper: how to aggregate profiling data in multi-process applications. We have used two alternatives: sum and average. Combined with the latency-based or bandwidth-based metrics, we obtained four different ProfDP memory object relevance rankings. For each ProfDP experiment, we used all four and present that providing the highest performance. To implement the data distribution we use FlexMalloc, which avoided the need for modifying the application's source code manually. This way, we provide an apples–to–apples comparison—note that the output from the Advisor may also be used to modify the source code manually—but we also demonstrate the generality of FlexMalloc's methodology outside our framework. Last, since profiling with HPCToolkit, unlike Extrae, modifies some call-stack frames, we had to manually fix some of these when generating the FlexMalloc input file in order to ensure matching call-stacks at the production run (without HPCToolkit) with those obtained during the profiling run.

To control the amount of DRAM used by the hardware-managed cache in memory mode, and to limit the variability introduced by NUMA effects, we perform our experiments pinning the threads and memory allocation to a single NUMA node. This limits the maximum DRAM available to 16 GB, and disk swap is disabled to avoid inadvertently overpassing this limit. We use a sampling rate of 100 Hz for both loads and stores in PEBS data collection. The profiling stage covers full runs with the same inputs as those in the performance evaluation stage. Applications showing input-dependent behaviors would require specific profiling runs to fine-tune runtime performance, whereas a study of the sensitivity of our heuristics to different data inputs is left for future work.

A. *Profiling Metrics and DRAM Size*

We first evaluate the performance of the object placement generated by the main algorithm of HMem Advisor based on density of LLC misses, as described in Section IV-B. Figure 6 shows the performance of a subset of the evaluated applications, exploring two configurations for the profiling metrics and three different maximum DRAM limits for dynamic memory allocations used by the HMem Advisor. PMem-6 results leverage our target DRAM-PMem ratio, while PMem-2 features a reduced PMem capacity and bandwidth of 1/3 (by physically removing DIMMs). For the *Loads* configuration, only LLC load misses are used to compute the memory objects' cost heuristics, while the *Loads+stores* configuration leverages the new adaptation of LLC load misses plus L1D store misses to derive the cost heuristics. Although there are

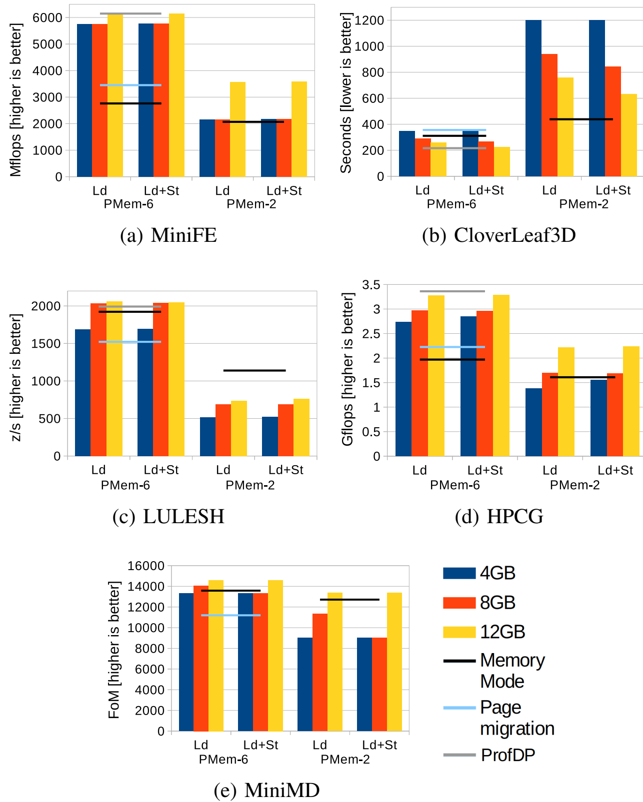

Fig. 6. Performance using different profiling metrics and limits on DRAM usage in HMem Advisor, for two PMem–DRAM memory ratios.

TABLE VI
MEMORY-RELATED PROFILING OF THE MEMORY MODE EXECUTIONS

	MiniFE	MiniMD	LULESH	HPCG	CloverLeaf3D
Memory Bound Pipeline Slots	90.2%	41.5%	65.5%	80.5%	93.5%
DRAM Cache Hit Ratio	39.9%	61.5%	61.7%	54.4%	59.2%

16 GB of DRAM available in a NUMA node, we have to reduce the DRAM limit in the HMem Advisor configuration file to 12 GB, to account for the other memory usage within the application (e.g. stacks, statically allocated data, etc.) and the memory used by the operating system. We also performed experiments with 4 and 8 GB limits to evaluate the performance impact of further reducing the amount of DRAM. Note that the memory mode baseline features all 16 GB of DRAM available to be used as cache. Each data point is the arithmetic mean of five executions, and the maximum relative standard deviation (RSD) is under 3%.

For the PMem-6 configuration and a 12 GB DRAM limit, executions using our framework experience higher performance than leveraging memory mode for all five miniapplications. More specifically, we observe three distinct behaviors: for MiniFE and HPCG our results reveal a significant performance improvement even when reducing our DRAM limit to 4 GB, attaining speedups of up to 2.22x for the former and 1.67x for the latter. CloverLeaf3D shows a 10% slowdown in the most restricted DRAM case, but for the other two DRAM limits it shows a reasonable improvement, attaining 39% over the baseline in the 12 GB case, which is the fairest comparison with memory mode in terms of available DRAM. In the other two applications, our performance improvement is less notable, reaching 8% for MiniMD and 7% for LULESH for the 12 GB DRAM experiments. Due to this lower advantage over the memory mode baseline, limiting the DRAM introduces a performance degradation compared to the baseline, reaching a maximum of 12% for LULESH.

These performance results correlate well with the memory subsystem behavior of the memory mode executions as presented in Table VI (obtained with Intel® VTune™). We observe that the memory mode for MiniFE and HPCG experiences a lower hit ratio than for the other three cases, which indicates that the active workload size is sufficiently large to cause more capacity misses, or these suffer from further conflict misses. Moreover, the performance of these two applications is highly dependent on the memory subsystem latencies, as shown by the memory bound pipeline slots statistic, which represents an approximation of the processor pipeline slots that are stalled due to memory loads and stores. Both characteristics combined leave more room for improvement for our framework to exploit, and explain the observed performance in MiniFE and HPCG. CloverLeaf3D is also markedly memory bound, but it experiences a higher DRAM cache hit ratio, limiting the possible improvement. MiniMD and LULESH show a more extreme case of this limitation: their performance is less memory bound and these feature a high hit rate, which reduces the room for improvement.

In these experiments, the effect of including the L1D store miss data in the HMem Advisor algorithm is negligible in all cases except for CloverLeaf3D. For HPCG, there is slight improvement of 5% in the 4 GB DRAM limit cases, but for the 8 and 12 GB cases the difference is negligible. For MiniMD with 8 GB of DRAM, considering the store data removes the 4% improvement over the baseline to a 2% slowdown. However, in CloverLeaf3D, including store data enables the HMem Advisor cost heuristics to capture relevant memory objects that were missed by the loads-only case. The 8 GB DRAM case improves the speedup over the memory mode baseline by an additional 9% and by an additional 19% for the 12 GB DRAM case. As discussed in Section IV, the lack of precise LLC store miss profiling events prevents our cost heuristics for stores from being more precise, which may result in lower-quality memory object placements.

All the results with the PMem-2 configuration show lower performance due to the reduction of the available bandwidth. This poses a larger impact for our proposal, because the memory mode can mitigate the reduction with the DRAM cache. Still, we obtain better performance for MiniFE, MiniMD and HPCG, attaining a maximum speedup of 1.74× in the former. We consider the PMem-6 configuration more relevant because performance-focused scenarios will alleviate the lower bandwidth of PMem by installing as many DIMMs as possible.

Kernel-level page migration shows lower performance than our framework, although outperforming memory mode for

TABLE VII
FUNCTION BREAKDOWN OF RELATIVE AVERAGE IPC AND LOAD ACCESS LATENCY OF CLOVERLEAF3D WITH RESPECT TO MEMORY MODE

Function	IPC	Latency
advec_cell_kernel	122.6%	77.6%
calc_dt_kernel	201.8%	44.3%
flux_calc_kernel	211.8%	51.4%
pdv_kernel	163.7%	50.2%
viscosity_kernel	126.3%	76.8%
advec_mom_kernel	98.3%	118.9%
ideal_gas_kernel	43.6%	132.8%
clover_pack_message_top	78.4%	129.0%
clover_pack_message_front	77.3%	107.2%
reset_field_kernel	76.9%	123.0%
update_halo_kernel	74.3%	127.7%
accelerate_kernel	87.6%	97.2%
clover_pack_message_right	87.3%	94.1%

MiniFE and HPCG. Enabling the NUMA node backed by PMem poses a DRAM cost for page management metadata proportional to PMem size ($\sim$15GB in our case) that limits the DRAM left for application use, impacting its performance.

Performance differences when compared to ProfDP (with 12 GB DRAM limit) are marginal. Our approach is slightly faster for LULESH, where our highest speedup over memory mode is 1.07×, while the highest for ProfDP is 1.04×. MiniFE, HPCG and CloverLeaf3D show slightly higher performance for ProfDP; the largest difference is for CloverLeaf3D, where we obtain a speedup of 1.39× versus 1.44× with ProfDP. We could not obtain ProfDP results for MiniMD because HPCToolkit crashed. These results confirm that our methodology attains similar performance to the state–of–the–art ProfDP approach, while providing several advantages: (1) a single profiling run instead of three, (2) no need for using a custom tool to link the application (HPCToolkit requires linking with `hpclink`), and (3) efficient runtime methodology that enables users to deploy the devised data distribution without the need for changing the source code or even recompiling their application.

B. Average Memory Access Latency

We present a detailed examination of CloverLeaf3D to assess how the object placement generated by the HMem Advisor affects the average memory access latency and the application performance. We profiled an execution of this application when run with FlexMalloc to compare the performance metrics with those obtained in the initial profiling execution. The memory access latency, in CPU cycles, is obtained using the PEBS counters for loads (PEBS store data does not include access latency). We collected the `PAPI_TOT_INS` and `PAPI_TOT_CYC` counters to compute the instructions per cycle (IPC). This information is filtered to the duration of one of the application iterations (all iterations exhibit similar behavior), aggregated by the function performing the memory access, and averaged across MPI ranks. Table VII contains the computed IPC and memory access latency of the execution using FlexMalloc, as a percentage of the same data computed from the memory mode execution. Functions that account for $<$1% of the time are not included.

Most of the functions in Table VII (first two groups) show the expected inverse correlation between IPC and memory access latency, where a reduction in latency results in higher IPC, or vice versa. We observe both cases because the memory object placement from HMem Advisor may improve the performance of the functions accessing the objects that are placed into DRAM, but the functions accessing objects placed into PMem are penalized. When the cost heuristics are able to identify the most performance-relevant objects, the overall application performance improves, as it is the case of CloverLeaf3D. A few functions (third group) show unexpected relationships between IPC and memory latency; besides memory latency, other kinds of pipeline stalls may influence the IPC, whereas the sampling nature of the collected latency data may be injecting some distortion.

C. Impact of the Bandwidth-Aware Placement

To evaluate the bandwidth-aware memory object placement algorithm, we use LULESH and two full applications, LAMMPS and OpenFOAM. While for the applications discussed in Section VIII-A the main algorithm of HMem Advisor improves performance with respect to memory mode, for these two cases it is not able to outperform the baseline. This is mainly due to the more complex memory allocation and access patterns, which require a more advanced algorithm. On the other hand, applying the bandwidth-aware algorithm poses no noticeable difference on those simpler cases.

Both applications are executed with the largest DRAM limit that the available 16 GB permit. For OpenFOAM, the limit is 11 GB, whereas in the case of LAMMPS, we could use different limits for the main and bandwidth-aware algorithms: 14 GB and 16 GB, respectively. This means that the bandwidth-aware algorithm is being less aggressive trying to utilize all the DRAM available; otherwise, the experiments would be running out of memory. Each experiment is repeated 5 times to compute the average (the maximum RSD is 0.8%).

For LAMMPS, we can see in Table VIII that there is almost no difference between the speedup with respect to memory mode of the main and the bandwidth-aware algorithm. We further analyzed this application with the VTune and Paraver [5] profilers, and observed that the room for improvement is limited. Aggregate statistics from VTune reveal that only 29.2% of stalls may be related to memory access latency, and the DRAM cache experiences a hit ratio of 63.5%. Comparing these metrics with the applications discussed in Section VIII-A, LAMMPS is markedly the least memory-bound code. From the analysis with Paraver, we observe that for the bulk of the compute iteration time, most of the working set fits into L2. The overhead of our executions originates in the MPI communication phases that occur in between each iteration. This suggests that memory objects related to MPI communications are being placed into PMEM, which introduces delays in the critical path. Since the communication phases represent a small proportion of the iteration time, the lower number of samples captured affects the capacity of HMem Advisor to identify the relevance of these objects. Nevertheless, even in this unfavorable case, the bandwidth-

TABLE VIII
SPEEDUP OF OPENFOAM AND LAMMPS W.R.T. MEMORY MODE

	OpenFOAM	**LAMMPS**
Main algorithm Ld+St	0.65	0.97
Bandwidth-aware Ld	1.03	0.96
Bandwidth-aware Ld+St	1.06	0.97

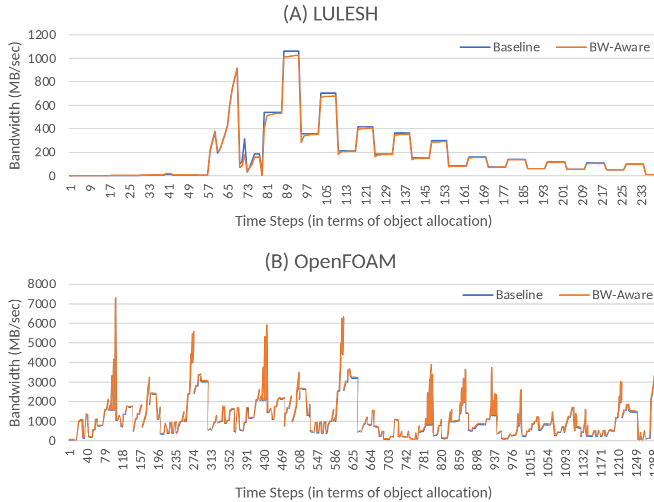

Fig. 7. PMem bandwidth usage with the main HMem Advisor algorithm (baseline) and the bandwidth-aware algorithm.

aware algorithm does not introduce any performance penalty, and the slowdown of our framework is kept below 4%.

The bandwidth-aware placement algorithm provides significant improvement for OpenFOAM and LULESH. OpenFOAM improves by 6.1% over memory-mode (whereas the base algorithm had a 2x slowdown), while for LULESH the improvement increases from 7% to 19%. Since our algorithm tries to reduce PMem bandwidth demand by moving demanding objects into DRAM, Figure 7 presents the effectiveness of our algorithm towards this objective for LULESH and OpenFOAM. For LULESH, as soon as objects causing high bandwidth demand (referring to Figure 3, most of the bandwidth demanding objects are allocated in the start of the high bandwidth phase) are moved to DRAM, improvement in bandwidth perfectly follows the bandwidth demand curve. This also demonstrates that most of the high bandwidth demand is originated by the small number of objects allocated at the start of the phase. OpenFOAM exhibits a much more complex memory access pattern and its bandwidth demand varies considerably throughout the execution. Nonetheless, our algorithm effectively reduces some of the demand by moving objects from high bandwidth regions.

D. Impact of the Call-Stack Format

We leverage OpenFOAM to showcase the impact of the call-stack format in runtime performance and memory consumption. The speedup with respect to memory mode for the *bandwidth-aware Loads+stores* experiment using human-readable call-stacks is 0.66, where we would lose almost all the improvement provided by the bandwidth-aware algorithm.

This overhead is originated by two factors. First, the additional space in DRAM required for the debug information needed to generate human-readable call-stacks reduces the DRAM limit that can be used by the HMem Advisor to 9 GB. This large difference is due to the fact that the same data is loaded in each MPI process, 16 in this case. Second, when using the BOM format, we avoid translating the call frame memory addresses to source file and line pairs. The lower DRAM limit seems to be the main contributor to overhead, and we planned to confirm this by storing the debug information in PMEM. However, with the current memkind, FlexMalloc cannot control allocation calls when already handling an allocation call from the application. Nevertheless, the new call-stack format we devised solves these issues while still allowing the matching of call-stacks across different executions.

IX. CONCLUSION

In this paper we have proposed methodology for memory object placement in hybrid memory systems. Our methodology requires no source code modification, outperforms production solutions and is able to address, for the first time in the literature, complex production applications from unmodified binaries, while notably simplifying the workflow with respect to similar state–of–the–art solutions. Our contributions include novel object-matching runtime methodology and a bandwidth-aware placement algorithm. Our detailed experimentation, including 5 miniapplications and 2 production applications, reveals runtime improvements with respect to the state of the art, hardware-based methodology used in production, up to over 2x in miniapplications, and up to 6% for OpenFOAM, a complex production application, whereas we have not found any use case in which our methodology yields considerably lower performance than the baseline. Our experiments also reveal performance improvements with respect to a reference kernel-level approach, as well as potential for substantial DRAM reduction within negligible performance penalty, hence enabling more cost-efficient and energy-friendly computing platforms. We expect the presented methodology and our implementation to be easily applicable to upcoming systems based on HBM and DRAM, as well as those leveraging CXL memory pools.

ACKNOWLEDGMENTS

Special thanks to Harald Servat from Intel, for providing invaluable technical advise and inputs to this paper, and for developing FlexMalloc for ecoHMEM. This paper received funding from the Intel-BSC Exascale Laboratory SoW 5.1, the European Union's Horizon 2020 research and innovation program under the Marie Sklodowska-Curie grant agreement No. 749516, the EPEEC project from the European Union's Horizon 2020 research and innovation program under grant agreement No 801051, the DEEP-SEA project from the European Commission's EuroHPC program under grant agreement 955606, and the Ministerio de Ciencia e Innovación—Agencia Estatal de Investigación (PID2019-107255GB-C21/AEI/10.13039/501100011033).

REFERENCES

[1] AccelCom Group at BSC, "ecoHMEM: Software ecosystem for heterogeneous memory management," https://www.bsc.es/research-and-development/software-and-apps/software-list/ecohmem-software-ecosystem-heterogeneous, Dec. 2021.

[2] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent, "HPCTOOLKIT: Tools for performance analysis of optimized parallel programs," *Concurrency and Computation: Practice and Experience*, vol. 22, no. 6, pp. 685–701, 2010.

[3] L. Alvarez, M. Casas, J. Labarta, E. Ayguade, M. Valero, and M. Moreto, "Runtime-guided management of stacked DRAM memories in task parallel programs," in *International Conference on Supercomputing*, 2018.

[4] "Extrae," https://tools.bsc.es/extrae, Barcelona Supercomputing Center.

[5] "Paraver," http://tools.bsc.es/paraver, Barcelona Supercomputing Center.

[6] Y. Chen, I. B. Peng, Z. Peng, X. Liu, and B. Ren, "ATMem: Adaptive data placement in graph applications on heterogeneous memories," in *International Symposium on Code Generation and Optimization (CGO)*, 2020.

[7] 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 *European Conference on Computer Systems (EuroSys)*, 2016.

[8] 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 *International Conference on Archictecture of Computing Systems*, 2018.

[9] Free Software Foundation, "GNU Binutils," https://www.gnu.org/software/binutils.

[10] D. Hansen, "Memory tiering," https://lwn.net/Articles/802544, 2019.

[11] M. A. Heroux, D. W. Doerfler, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, and R. W. Numrich, "Improving performance via mini-applications," Sandia National Laboratories, Tech. Rep. SAND2009-5574, 2009.

[12] M. A. Heroux, J. Dongarra, and P. Luszczek, "HPCG benchmark technical specification," Sandia Natl. Lab. (SNL-NM), Tech. Rep., 2013.

[13] M. Hildebrand, J. Khan, S. Trika, J. Lowe-Power, and V. Akella, "AutoTM: Automatic tensor movement in heterogeneous memory systems using integer linear programming," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.

[14] Intel, "Intel Optane DC Persistent Memory product brief," www.intel.com/content/www/us/en/products/docs/memory-storage/optane-persistent-memory/optane-dc-persistent-memory-brief, 2019.

[15] Intel Corporation, "Intel Memory Latency Checker," http://software.intel.com/content/www/us/en/develop/articles/intelr-memory-latency-checker.html.

[16] "Memkind," https://memkind.github.io/memkind, Intel Corporation.

[17] Intel Corporation, "Intel(r) 64 and IA-32 architectures software developer manuals," http://software.intel.com/content/www/us/en/develop/articles/intel-sdm.html, 2021.

[18] J. Izraelevitz, J. Yang, L. Zhang, J. Kim, X. Liu, A. S. Memaripour, Y. J. Soh, Z. Wang, Y. Xu, S. R. Dulloor, J. Zhao, and S. Swanson, "Basic performance measurements of the Intel Optane DC Persistent Memory module," *arXiv:1903.05714*, 2019.

[19] H. Jasak, A. Jemcov, and Z. Tukovic, "OpenFOAM: A C++ library for complex physics simulations," in *International Workshop on Coupled Methods in Numerical Dynamics*, 2007.

[20] I. Karlin, J. Keasler, and R. Neely, "LULESH 2.0 updates and changes," Lawrence Livermore National Lab. (LLNL), Tech. Rep. LLNL-TR-641973, August 2013.

[21] M. Laghari, N. Ahmad, and D. Unat, "Phase-based data placement scheme for heterogeneous memory systems," in *International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, 2018.

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

[23] G. Lloret-Talavera, M. Jorda, H. Servat, F. Boemer, C. Chauhan, S. Tomishima, N. N. Shah, and A. J. Peña, "Enabling homomorphically encrypted inference for large DNN models," *IEEE Transactions on Computers*, vol. 71, no. 5, pp. 1145–1155, 2022.

[24] K. T. Malladi, F. A. Nothaft, K. Periyathambi, B. C. Lee, C. Kozyrakis, and M. Horowitz, "Towards energy-proportional datacenter memory with mobile DRAM," in *International Symposium on Computer Architecture (ISCA)*, 2012.

[25] M. Marques, I. Kuzmin, J. Barreto, J. Monteiro, and R. Rodrigues, "Dynamic page placement on real persistent memory systems," *arxiv:2112.12685*, 2021.

[26] M. B. Olson, B. Kammerdiener, M. R. Jantz, K. A. Doshi, and T. Jones, "Portable application guidance for complex memory systems," in *International Symposium on Memory Systems (MEMSYS)*, 2019.

[27] M. B. Olson, T. Zhou, M. R. Jantz, K. A. Doshi, M. G. Lopez, and O. Hernandez, "MemBrain: Automated application guidance for hybrid memory systems," in *International Conference on Networking, Architecture and Storage (NAS)*, 2018.

[28] A. J. Peña and P. Balaji, "A framework for tracking memory accesses in scientific applications," in *43rd International Conference on Parallel Processing Workshops (ICPP-W)*, 2014.

[29] A. J. Peña and P. Balaji, "Toward the efficient use of multiple explicitly managed memory subsystems," in *IEEE Cluster*, 2014.

[30] S. Plimpton, "Fast parallel algorithms for short-range molecular dynamics," *Journal of Computational Physics*, vol. 117, no. 1, pp. 1–19, 1995.

[31] J. Ren, J. Luo, K. Wu, M. Zhang, H. Jeon, and D. Li, "Sentinel: Efficient tensor migration and allocation on heterogeneous memory systems for deep learning," in *International Symposium on High-Performance Computer Architecture (HPCA)*, 2021.

[32] H. Servat, "Flexible memory allocation tool for multi-tiered memory systems," https://github.com/intel/flexmalloc, 2022.

[33] H. Servat, J. Labarta, H. C. Hoppe, J. Giménez, and A. J. Peña, "Integrating memory perspective into the BSC performance tools," in *46th International Conference on Parallel Processing Workshops (ICPPW)*, 2017, pp. 231–232.

[34] H. Servat, J. Labarta, H. C. Hoppe, J. Giménez, and A. J. Peña, "Understanding memory access patterns using the BSC performance tools," *Parallel Computing*, vol. 78, pp. 1–14, 2018.

[35] H. Servat, A. J. Peña, G. Llort, E. Mercadal, H. C. Hoppe, and J. Labarta, "Automating the application data placement in hybrid memory systems," in *IEEE Cluster*, 2017.

[36] UK-MAC, "CloverLeaf3D," http://uk-mac.github.io/CloverLeaf3D.

[37] T. Vogelsang, "Understanding the energy consumption of dynamic random access memories," in *43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2010.

[38] S. Wen, L. Cherkasova, F. X. Lin, and X. Liu, "ProfDP: A lightweight profiler to guide data placement in heterogeneous memory systems," in *International Conference on Supercomputing (ICS)*, 2018.

[39] K. Wu, J. Ren, and D. Li, "Runtime data management on non-volatile memory-based heterogeneous memory for task-parallel programs," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC18)*, 2018.

[40] Z. Yan, D. Lustig, D. Nellans, and A. Bhattacharjee, "Nimble page management for tiered memory systems," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019.

[41] L. Zhang and S. Swanson, "Pangolin: A fault-tolerant persistent memory programming library," in *USENIX Annual Technical Conference*, 2019.