Phase-based Data Placement Optimization in Heterogeneous Memory

Jannis Klinkenberg, Clément Foyer, Pierre Clouzet, Brice Goglin, Emmanuel Jeannot, Christian Terboven, Anara Kozhokanova

Phase-based Data Placement Optimization in Heterogeneous Memory

Jannis Klinkenberg* ✉, Clément Foyer†, Pierre Clouzet‡, Brice Goglin‡, Emmanuel Jeannot‡, Christian Terboven*, Anara Kozhokanova*
*Chair for High Performance Computing, RWTH Aachen University, Aachen, Germany
Email: {j.klinkenberg, terboven, kozhokanova}@itc.rwth-aachen.de
†Université de Reims Champagne-Ardenne, CEA, LRC DIGIT, LICIIS, Reims, France
Email: clement.foyer@univ-reims.fr
‡Inria, Univ. Bordeaux, LaBRI, Talence, France
Email: {pierre.clouzet, brice.goglin, emmanuel.jeannot}@inria.fr

***Abstract*—While scientific applications show increasing demand for memory speed and capacity, the performance gap between compute cores and the memory subsystem continues to spread. In response, heterogeneous memory systems integrating high-bandwidth memory (HBM) and non-volatile memory (NVM) alongside traditional DRAM on the CPU side are gaining traction. Despite the potential benefits of optimized memory selection for improved performance and efficiency, adapting applications to leverage diverse memory types often requires extensive modifications. Moreover, applications often comprise multiple execution phases with varying data access patterns. Since the capacity of the "fastest" memory is limited, relying solely on fixed data placement decisions may not yield optimal performance. Thus, considering allocation lifetimes and dynamically migrating data between memory types becomes imperative to ensure that performance-critical data for each phase resides in fast memory.**

To address these challenges, we developed a workflow incorporating memory access profiling, optimization techniques and a runtime system, which selects initial data placement for allocations and performs data migration during execution, considering the platform's memory subsystem characteristics and capacities. We formalize the optimization problems for initial and phase-based data placement and propose heuristics derived from memory profiling metrics to solve it. Additionally, we outline the implementation of these approaches, including allocation interception to enforce placement decisions. Experiments conducted with several applications on an Intel Ice Lake (DRAM+NVM) and Sapphire Rapids (HBM+DRAM) system demonstrate that our methodology can effectively bridge the performance gap between slow and fast memory in heterogeneous memory environments.

***Index Terms*—Heterogeneous memory, Non-volatile memory, High-bandwidth memory, Heuristics, HPC, Data placement**

I. Introduction

In all modern computing systems, the performance gap between compute and memory continues to spread, particularly in the face of multi-core and accelerated systems. In consequence, the memory subsystem is changing: the evolution of the cache hierarchy is followed by the introduction of new memory technologies as new kinds of memory in computing systems. In the context of HPC, this has been pioneered by combining traditional main memory (typically DDR) with a significantly smaller amount of high-bandwidth memory (HBM) on the CPU side. More recent systems include large portions of byte-addressible slower non-volatile memory (NVM) besides the traditional DDR. Combining multiple kinds of memory results in a heterogeneous memory system and these new kinds of memory offer new and different trade-offs between capacity and speed. Additionally, many modern systems are equipped with accelerators such as GPUs, which provide their own memory (often HBM as well), which increasingly often is also accessible by the host as shared memory. However, for this work, we will focus on heterogeneous memory on the CPU-side only.

Programming for such systems with heterogeneous memory creates new challenges, of which three stick out. First, for each memory allocation it has to be considered what the right kind of memory would be, given application performance as the optimization goal. Since faster memory typically is smaller, putting all data in the fastest memory is often not possible for real applications. Consequently and second, a subset of all allocated data objects has to be selected to be placed in the fastest memory. A data-driven and optimization-guided selection of data objects that should be placed in the faster memory should improve performance over a greedy first-come-first-served approach, that will place first appearing allocations into that memory as long as there is capacity left. Further, if the runtime system supports data migration between different kinds of memory, performance could be improved even further, as the most critical data for the performance of the current application phase could be put into the faster memory. So third, an improved strategy would have to divide the application in phases and manage the movement of data between memory kinds based on the optimization result of each phase.

In this work, we present the extended functionality of the H2M runtime system [1] that efficiently and automatically manages data placement and data migration on systems with heterogeneous memory. The following contributions stick out:

- We formalize the data placement optimization problems

This work was supported by the French National Research Agency (ANR) in the frame of the ANR-DFG H2M project (ANR-20-CE92-0022-01). Parts of this work have been funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – 446185093.

based on previously collected memory profiling data and a characterization of modern heterogeneous memory architectures.
- We model and perform the data placement by taking individual application phases into account.
- We provide an implementation sketch of the H2M extensions, which are available as open source, and evaluate our approach on two recent systems with heterogeneous memory.

The rest of this paper is structured as follows. Section II provides background information on memory characteristics. Section III presents the new approach of our work, introducing the workflow and placement optimization heuristics. In Section IV, we evaluate our work, followed by a discussion of related work in Section V. We draw our conclusions in Section VI.

II. Background on Memory Characteristics

The gap between memory bandwidth and processor throughput widens. Hence, memory subsystems of modern computing systems contain a deep hierarchy of memory technologies [2]. Besides multiple levels of caches, the traditional main memory (DRAM) may now be combined with *High-Bandwidth Memory* (HBM) and/or *Non-Volatile Memory* (NVM), while these technologies have different benefits and limitations. We first introduce the current and upcoming heterogeneous memory platforms before describing how they may be managed, e.g., with the hwloc library.

We will use the term *memory technology* to refer to a concrete technology or product, such as HBM. *Memory kind* will be used to refer to memory with a specific, distinct property, such as the largest capacity.

A. Hardware Landscape

Heterogeneous memory first appeared widely in high performance computing in 2016 with the Intel *Knights Landing* Xeon Phi processor (KNL). It embedded high-bandwidth MCDRAM while still supporting traditional off-package DRAM [3]. They could be used as explicit separate NUMA domains requiring the allocator to decide between low-capacity high-bandwidth and large-capacity low-bandwidth memory [4].

KNL is now discontinued and no similar platforms have been used in HPC until Intel released the *Sapphire Rapids* Xeon last year, which optionally embeds 64 GB of HBM2e [5]. Combined with the usual off-chip DRAM, its memory subsystem is very similar to KNL. The same trend is followed by vendors involved in several exascale initiatives, especially with ARM Neoverse V1 CPUs, exposing both DRAM and HBM. SiPearl's *Rhea* processor will be used in the *European Processor Initiative*. In South Korea, ETRI will develop the *K-AB21* CPU. In India, the C-DAC is designing the *Aum* processor.

A different kind of heterogeneity was brought by the emergence of non-volatile memory (NVM). This memory technology may be used as persistent storage or as large-capacity main memory. Hence, when combined with NVM, DRAM is considered the low-capacity (but fast) memory. Intel is phasing out this product (*Optane DCPMM*) which is not widely used in HPC despite showing interesting benefits for performance and capacity on some applications [6]. However, as explained below, NVM will come back as standard CXL devices in the near future.

Moreover, modern platforms are able to share GPU memory with the host CPU in a cache-coherent manner. NVIDIA exposed GPU HBM to POWER processors through the NVLink protocol. The recently released *Grace Hopper* superchip continues this road by sharing the ARM CPU LPDDR5 memory and the GPU HBM3 cache-coherently [7]. AMD (e.g., MI250X on Frontier) and Intel are following the same path with their newest platforms. This effectively exposes heterogeneous memory to applications, although GPU memory is still preferably allocated using custom GPU APIs instead of standard system calls.

All these platforms and technologies exhibit significant memory performance variances: MCDRAM and HBM have much higher bandwidth than DRAM, but their latency is in some cases higher, especially when accessed through additional buses (e.g., NVLink). NVM has significantly higher latency and lower bandwidth, but offers larger capacity. Hence, there is no best target for all needs: Capacity usually decreases when bandwidth increases, and they have no correlation with latency.

However, the main reason for the heterogeneity of future memory subsystems is new interconnects with generic support for different kinds of memory. The *Compute Express Link* (CXL), already adopted by many vendors, is set to become the dominant standard. It will bring standard support for NVM to many processor models [8]. This variety of technologies and the ability to use cables of different length or even switches will bring much more diverse heterogeneity. Indeed, the interconnect topology may impact the end-to-end bandwidth and latency.

B. Software Management of Heterogeneity with hwloc

In the KNL era, software support for heterogeneous memory was lacking. Fortunately, only very few hardware configurations were available (single socket, single HBM capacity, only combined with DRAM, known performance). Hence software developers were able to tweak their applications specifically for KNL. Nowadays, with different platforms supporting different kinds of memory, the software stack has to explicit enumerate the existing NUMA nodes, identify whether they are HBM, DRAM or NVM and discover their performance before applying placement decisions.

This issue of identifying which NUMA nodes are fast or slow, and how, is actually not handled by most research works on heterogeneous memory. They assume the user will manually tell the software stack which NUMA nodes are of which technology. As we believe more portability is needed on the path to higher productivity, we implemented the required abilities in *hwloc* [9], the *de facto* standard tool for describing the topology of parallel platforms and managing the locality

of tasks and data buffers. hwloc 2.3 brought an API to query the performance of memory nodes (either measured through benchmarks, or obtained in the ACPI HMAT table), find the best memory target for a given metric, sort them, etc. [10] We contributed in hwloc 2.8 an extension that combines performance information with the hardware knowledge (E820 special purpose flag for HBM, CXL information, etc.) to find out which node is DRAM, HBM, CXL, etc. We believe that combining such static information about memory technologies with performance information (bandwidth, latency) is the key to answering the needs of both users and runtimes. In the next sections, we will show how to use these features to decide where to allocate memory.

III. Methodology and Approach

Although there are low-level (and sometimes also vendor-specific) libraries available to allocate memory in HBM, DRAM or NVM, users typically have to explicitly and manually take care of utilizing them and deciding where to place allocations by modifying their source code. However, in many cases, developers lack specialized knowledge in HPC and prefer not to tailor their applications solely to a specific system architecture.

In contrast, our work aims at providing users with higher-level abstractions and a vendor-neutral and architecture-independent solution to manage the data placement on heterogeneous shared memory systems. Our approach is based on abstracting or intercepting memory allocations to enable developers to express additional hints or requirements (also called *traits*) for individual allocations that describe what attributes these data objects have or how the data is used and accessed throughout the application execution. By leveraging these *traits* along with insights about the system's topology, memory layout and memory characteristics, the runtime system efficiently manages data allocation or data movement by selecting a suitable physical storage. Allocation abstraction and interception lay the foundation for the methodology presented in this work and will be detailed in the subsequent sections.

Further, during the runtime initialization, our library employs low-level hwloc APIs to automatically identify and categorize NUMA domains on-the-fly, eliminating the need for pre-existing assumptions and making it easily portable across numerous modern architectures.

A. Workflow

Although our runtime offers the option to provide additional allocation hints, it remains uncertain whether a programmer always possesses the expertise to know these requirements or has a profound understanding of the underlying architectural features or data access patterns. Therefore, we developed a data-driven workflow, as illustrated in Fig. 1, to assist users in identifying and specifying suitable allocation traits that, in this work, will be used to optimize data placements and data migrations in applications.

The workflow comprises the following steps. First, the original application is executed once along with a memory profiler, capturing information about allocations, memory accesses and application phase transitions happening during the execution. Next, the resulting profiling data is fed to our *placement optimizer* that incorporates knowledge about the available memory kinds and their distinct performance characteristics and capacity limits to optimize the placement decisions for individual allocations. This will be detailed in Section III-D. The resulting placement decisions are saved in a JSON configuration file and, in subsequent executions, finally exploited by the original unmodified application where allocation calls are *intercepted* on an `LD_PRELOAD` basis.

B. Allocation Abstraction, Interception and Phase Transitions

As of now, our solution supports *allocation abstraction* as well as *allocation interception*. In the *abstraction* case, the programmer replaces regular calls to `malloc` / `free` in C or `new` / `delete` in C++ with the corresponding abstraction counterparts from the library, which allow expressing additional traits. Traits can either contain descriptive hints, which will be exploited by an *allocation strategy* at run time to guide the memory kind selection, or prescriptive hints that directly prescribe where a certain allocation should be done. In our previous work [1], we detailed different variants for allocation abstraction along with their specific advantages and limitations. Nevertheless, all abstraction variants still require the user to moderately modify the application source code and replace allocation and de-allocation API calls. Especially when applying the proposed workflow to larger existing codes or when working with C++ applications that heavily rely on standard containers, this is not desirable and can quickly become more complex. Further, allocation *abstraction* cannot distinguish between separate allocations that have been issued from the same source code line but from different call stack paths. As an example, consider that an allocation is done in $funcC$, which might be called from different locations or paths such as $main \rightarrow funcA \rightarrow funcC$ or $main \rightarrow funcB \rightarrow funcC$. Allocation interception however is able to detect the difference and treat those allocations separately. Therefore, in this work, we focus on an *allocation interception* solution to minimize manual code modification as much as possible, while yielding comparable or better results.

To further support different execution phases, the only minimal code modification that needs to be done by developers is placing `phase_transition()` calls into the program (e.g., between two loop nests, functions, or parts of the simulation code) to split the application into separate execution phases as shown in Listing 1. Such minor modifications can easily be done by the code developers, who often know the structure of their programs. During the memory profiling step, we utilize this information to identify and distinguish memory accesses of individual application phases. During productive runs, these calls act as entry points to our runtime system and can trigger data migrations between memory kinds if desired to improve overall phase execution times. The *allocation interception* case

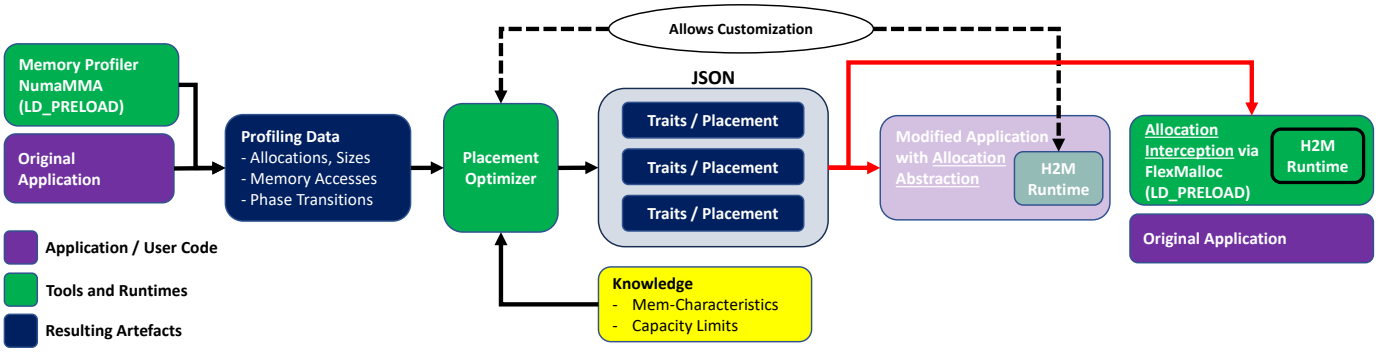


Fig. 1: Overview of the data-driven workflow for generating data placement optimizations. The resulting traits or placement decisions can be utilized by the allocation abstraction or interception layers to select the desired memory space during allocation or migrate data between application phases. In this work, we focus on allocation interception.

```
// variable declaration and memory allocation
size_t N; // array size
double* data = (double*) malloc(sizeof(double)*N);

// first hotspot or execution phase
#pragma omp parallel for schedule(static)
for(size_t i = 0; i < N; i++) { /* do work */ }

// call to mark transition between application phases
h2m_phase_transition("start of foobar");

// second hotspot or execution phase
double result = foobar(data);
```

Listing 1: Example for a phase transition within an application source code.

relies on a modified version of FlexMalloc[1] [11]. Here, we defined three memory allocators to reflect the three memory spaces in the H2M runtime (high bandwidth, low latency and large capacity) that can explicitly be targeted by users in FlexMalloc configuration files. However, for this work, we defined a fourth allocator which forwards all intercepted calls to the runtime that is then able to allocate to any memory space.

Individual allocations are identified based on the call stack, the chain of function calls that leads to the allocation point in the code. This identification either uses the debugging symbols, allowing the tracking of functions by the name of the source file and the line of the instruction, or it relies on the stackframe state at allocation time. The stackframes form a list of pointers locating the return value address of each function called that led to the allocation. This list can be obtained by a call to the standard function `backtrace(3)`. At run time, we intercept calls to allocation functions using `LD_PRELOAD`, which allows us to redirect calls to `malloc`[2], `realloc` or `posix_memalign`, for example, to the FlexMalloc interception layer which then redirects calls to our corresponding runtime allocator. This allocator considers placement decisions from the JSON file resulting from the analysis presented in Section III-C to request memory in the appropriate memory space. To find entries for the desired memory allocation, the JSON file also provides stackframe information for each allocation, which has been recorded during the memory profiling step and against which we do the matching. Neither this analysis nor the interception of allocations require any change from the application whose behavior is to be changed. Through our experiments, we have observed that the overhead associated with our technique, involving allocation interception, obtaining placement decisions from JSON files, and redirecting the allocation process, is under 1 % of the execution time.

[1]Available at https://gitlab.inria.fr/h2m/flexmalloc

[2]Allocation calls in Fortran or to C++ equivalent (`new`) ultimately result in a call to libc's `malloc`

C. Memory Access Profiling

For profiling memory allocations and accesses in an application run, we use NumaMMA [12], a sampling-based profiler that is utilizing hardware counters such as *Precise Event Based Sampling* (PEBS) on Intel and *Instruction Based Sampling* (IBS) on AMD architectures. NumaMMA operates on an `LD_PRELOAD` basis and collects various information about memory allocations and their memory accesses (load and stores), as shown in Table I. The collected profiling data can be dumped and serves as an input for the subsequent placement optimization step. More details about NumaMMA are provided in related previous work [1], [12]. Note that the profiling step only needs to be performed once in order to optimize the placement and migration decisions for subsequent executions.

D. Metrics, Heuristics and Placement Optimization

In this section, we will detail the metrics and heuristics that are applied by our placement optimizer to decide which allocations will be placed in the limited faster or the larger slower memory. Note that all metrics used in the subsequent equations can directly be calculated using information from the collected profiling data (i.e., Table I) and underlying memory characteristics of the systems. In that context, we distinguish between the following fundamental scenarios:

Per memory allocation	Per memory access
	thread that performed access
allocation id	timestamp
start address and size	allocation id
(de-)allocation timestamps	offset accessed
callsite address + file and line	access type (load or store)
callstack addresses and offsets	level (cache or main memory)
	weight (latency) for access

TABLE I: Profiling data collected by NumaMMA and provided in dumps.

- **Initial data placement (IDP)** considers all allocations and all memory accesses of the *complete* application profile to decide which data objects should reside in the faster memory. Once data has been allocated there it will not move and stay there for its entire lifetime. Here, we implemented the following two versions: while the first version will consider all allocations and their memory accesses from the start until the end of the program, the second version will also take individual object lifetimes (via allocation and de-allocation timestamps) into account.
- **Phase-based data placement (PBDP)**, in contrast, operates at a *per-phase* granularity, considering memory accesses and allocations from the upcoming P execution phases (typically $P = 1$) to take placement decisions. If desirable data objects are already allocated in slower memory, optimizing performance may involve shifting less important objects to slower memory and prioritizing the migration of high-priority objects to faster memory.

1) Estimating Performance Improvement: To determine the appropriate location for mapping a data object, we must first calculate a score that reflects the advantage of placing the object in the faster memory. To determine this, we estimate the access time for an object i (in read mode) for memory M as follows:

$$\text{Acc}_i^{\text{read,M}} = \frac{N_i^{\text{read}} \cdot \text{Lat}_M}{\text{CL/sizeof(double)}} \cdot (1 - R_i) + \frac{N_i^{\text{read}} \cdot \text{CL}}{\text{BW}_M^{\text{read}}} \cdot R_i \quad (1)$$

Here, N_i^{read} represents the number of read accesses to object i as monitored by the memory profiler; $\text{CL} = 64$ is the cache line size in bytes; $\text{BW}_M^{\text{read}}$ is the read bandwidth for memory M, which depends on the number of utilized threads; R_i is the cache hit ratio for object i; Lat_M is the access latency for memory M. By weighing the terms for bandwidth and latency with the cache hit ratio, we can put more emphasis on the latency part if accesses to that object often cause cache misses. This metric requires information about access latencies and bandwidth scaling of the different memory kinds, which needs to be measured beforehand, e.g. using the Memory Latency Checker [13]. Symmetrically, we can determine $\text{Acc}_i^{\text{write,M}}$ and with that the overall access time for a given memory M as follows: $\text{TAcc}_i^{\text{M}} = \text{Acc}_i^{\text{read,M}} + \text{Acc}_i^{\text{write,M}}$.

Finally, the estimated improvement score V_i for data object i is defined as:

$$V_i = (\text{TAcc}_i^{\text{SlowMem}} - \text{TAcc}_i^{\text{FastMem}}) \times \text{Freq}_\text{S} \quad (2)$$

Due to the sampling approach, this value would underestimate the execution time improvement. Hence, we need to multiply it with the sampling frequency Freq_S to achieve values in an appropriate order of magnitude.

2) Initial Data Placement Optimization: Based on the improvement scores V_i for each data object, we now need to select in which memory objects should be allocated. We have two choices: either allocate buffers to the fast memory (but limited in capacity) or to the slow memory (supposedly large enough for all objects). To this end, we model the problem as a *0/1 knapsack problem* where we consider the following inputs:

- Let n be the number of buffers that will be allocated by the application.
- Let S_i, for $i \in [1..n]$, be the size of buffer i, and $V_i > 0$ represents the score when this buffer is mapped to fast memory (as computed by Eq. 2).
- Let W be the capacity of the fast memory.

Next, we will determine the decision variable x_i for the allocation, where $x_i = 1$ if buffer i is allocated in the fast memory, and $x_i = 0$ if it is allocated in slow memory. To do so, we will solve the following integer linear programming (ILP) task:

$$\max_x \quad \sum_{i=1}^{n} x_i V_i \quad (3)$$

$$\text{s. t.} \quad \sum_{i=1}^{n} x_i S_i \leq W \quad (4)$$

$$x_i \in \{0, 1\}. \quad (5)$$

The objective function (Eq. 3) states that we want to maximize the gain of mapping objects to fast memory. The constraints state that the sum of the sizes of the objects allocated to fast memory cannot exceed the available memory capacity (Eq. 4), and that each object must be allocated either to fast or to slow memory (Eq. 5).

We use integer linear programming because it provides a uniform framework for the migration problem (later described in Section III-D4). Even though solving this problem using ILP can be computationally intractable in theory, in our case, the number of objects is limited (typically tens or hundreds), and the time to solve the initial data placement is negligible (less than 1 second for 1000 data objects) and performed post-mortem.

Although this optimization presents a good basis for initial data placement, often not all buffers have a lifetime that spans the entire duration of the application. Throughout the application's execution, several buffers might be allocated while others are freed again. Consequently, the initial allocation strategy needs to be re-evaluated, also taking object lifetimes and the dynamic nature of memory usage into account.

Algorithm 1 Recursive Initial Data Placement Optimization

```
sorted ← SortByLifetimeDesc(allocs)
remain ← Copy(sorted)
inFast, inSlow ← ∅, ∅
while remain ≠ ∅ do
    cur ← remain.pop()                ▷ Remove first allocation
    overlap ← FindOverlapping(cur, sorted)
    set, limit ← GetOptSet(overlap, inFast, inSlow)
    result ← RunOptimization(set, limit)
    inFast.extend(result)
    for alloc ∈ result do
        remain.remove(alloc)
    end for
    if cur ∉ result then
        inSlow.append(cur)
    end if
end while
```

3) Extended Initial Data Placement using Object Lifetimes: Our first approach to mitigate the limitation of the previous optimization strategy is to extend the initial data placement procedure by also respecting allocation and de-allocation timestamps of individual data objects that can be extracted from the memory profiling dumps. This allows identifying cases where allocations have been placed in the faster memory and are de-allocated. The freed-up capacity can then be used for later allocations that require faster memory.

Our solution involves solving the knapsack optimization problem (Eq. 3 to 5) in a recursive fashion as described in Algorithm 1, by gathering profiling data from allocations with an overlapping lifetime and conducting separate knapsack optimization passes for those sets, until placement decisions for all allocations have been found. For each optimization pass, we check which of the overlapping allocations have already been decided to go into fast memory and accordingly reduce the maximum available capacity of that memory.

4) Phase Based Data Placement: A second approach is to consider that an application may comprise different execution phases, where each execution phase might access different data or exhibit different data access patterns to buffers. In such cases, it can be beneficial to migrate some buffers to optimize data access times for the subsequent phase(s). However, migrating data objects has a cost that depends on several factors (hardware characteristics, buffer size, etc.). Hence, we need to find a trade-off between placing highly used buffers in fast memory and paying the cost of migrating objects between fast and slow memory. Moreover, when considering the data placement, we also need to account for new objects that are not yet allocated.

To address this case, we extend the ILP formulation of Section III-D2 for phase-based data migration by adding a set of new variables:

- Let a_i, be a binary variable stating if the data object i is already allocated ($a_i = 1$) and we therefore need to account for the data movement cost or not ($a_i = 0$).
- Let y_i, be a binary variable describing whether buffer i is currently in fast memory ($y_i = 1$) or slow memory ($y_i = 0$). Of course, the value of this variable is meaningful only if $a_i = 1$.
- Let $C_i^{f \to s}$ (resp. $C_i^{s \to f}$) be the costs to move all the pages of buffer i from fast to slow (resp. slow to fast) memory. Again, the value of this variable is meaningful only if $a_i = 1$. The cost to move a buffer from one memory to another is given by the size of this buffer divided by the memory copy bandwidth from the source memory to the target memory which has also been measured and provided to the optimizer.

In this case, we only need to change the objective function of the above ILP (Eq. 3) to:

$$\max_x \sum_{i=1}^{n} x_i V_i - a_i \left(y_i (1 - x_i) C_i^{f \to s} + (1 - y_i) x_i C_i^{s \to f} \right) \quad (6)$$

This new objective function expresses our goal of maximizing the benefit gained from assigning a buffer to fast memory while accounting for the costs associated with transferring buffers that were already allocated ($a_i = 1$) from fast to slow memory if they were initially in fast memory, as well as the costs of moving from slow to fast memory in the reverse situation.

5) Dealing with more than 2 kinds of memory: If there are more than 2 kinds of memory, we can apply the above framework recursively. We first use the knapsack formulation to decide which memory objects are allocated to the fastest (and smallest memory), then we reapply it to the remaining buffers to select those to be allocated in the second fastest (and smallest) memory, etc. To compute V_i in Eq. 2 we can employ a conservative approach, assuming that buffers not allocated in the fastest memory will be allocated in the second-fastest memory.

IV. Evaluation

A. Applications

To evaluate our approaches, we investigated several well-known HPC benchmark suites and selected shared-memory codes for which we could observe a significant performance difference between runs, where all data is allocated in either the faster or slower memory. These code and proxy applications comprise:

- **LULESH** [14], [15] that is designed to simulate the behavior of materials under extreme conditions, such as those found in shock wave physics or hydrodynamics.
- **miniFE** [16] as a representative for unstructured implicit finite element codes.
- **XSBench** [17], which serves as a proxy application for full neutron transport applications like OpenMC [18].
- **NPB-BT** [19] from the *NASA Advanced Supercomputing Parallel Benchmarks*. As BT and SP actually have a very similar behavior and run similar computations, we decided to only show BT in this work.

- **SPEC OMP2012** [20]: out of the OpenMP benchmarks provided by the Standard Performance Evaluation Corporation (SPEC) we selected *351.bwaves*.

Further, we employ a synthetic iterative *Multi-Phase* benchmark[3] as a representative for codes that have a dedicated data allocation and initialization phase at the start and comprise multiple execution phases, which either use different data objects or exhibit varying access patterns to the same data objects.

The benchmark allows to define the number and size of buffers to allocate, which can subsequently be used by the following set of computational kernels: a naïve non-blocked DGEMM kernel, a dot product kernel as well as a daxpy and stencil kernel. Each execution phase will execute a configurable number of randomly selected kernels. Similarly, data objects for the kernels are also randomly selected. However, to ensure reproducibility and a fair comparison, the random number generator is always initialized with the same seed.

For each program, we analyzed the predominant hotspots with common profiling tools such as Score-P or Intel VTune once to identify individual execution phases and accordingly placed `phase_transition()` calls in the codes. The computation kernels of XSBench and miniFE only consist of a single large region. Hence, there are no phases available for those applications. Further, we reviewed the allocation behavior and found the following: some codes are allocating all data in a dedicated step at the start and those objects are alive for the complete execution. Others allocate part of the data items for specific execution phases only. Choosing an appropriate approach might depend on those characteristics as well as the duration of individual phases. Data migration between longer phases might be worth, whereas frequent migrations between short phases might be too costly. Table II lists applications, parameters and their corresponding characteristics.

B. Setup and Architectures

For our experiments, we relied on two platforms.The first is a dual-socket Intel Xeon Gold 6338 (Ice Lake) with 32 cores each, running at 2.00 GHz. It is equipped with 512 GB of DRAM and 2 TB of *Intel Optane DC Persistent Memory* (NVM) which is exposed as a separate NUMA domain per socket. The Memory Latency Checker [13] reports idle latencies of 76 ns for DRAM and 247 ns for NVM, and an achievable memory bandwidth of 321 GB/s for DRAM and 52 GB/s for NVM.

The second platform, shown in Fig. 2, is a dual-socket Intel Xeon Max 9460 (Sapphire Rapids), each comprising 40 cores running at 2.2 GHz. We configured the processor in SNC-4 mode with flat memory. Hence, each Sub-NUMA cluster is composed of 10 cores and has access to 64 GB of DRAM and 16 GB of high-bandwidth memory (HBM). Here, idle latencies for DRAM and HBM are 98 ns and 121 ns, while bandwidths are 496 GB/s and 1329 GB/s, respectively. On both platforms, we disabled Hyperthreading.

[3]Available at: https://gitlab.inria.fr/h2m/h2m/-/tree/main/examples/codes/multiple_phases

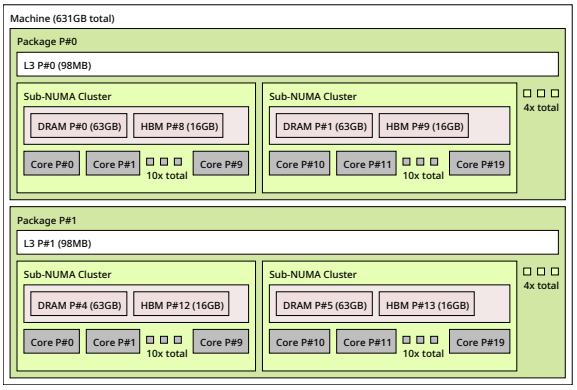


Fig. 2: Topology of the Sapphire Rapids platform.

The runtime as well as all applications were compiled with GCC 11.4.0 and `-O3` optimization. To effectively highlight the distinctions in memory technologies and minimize the influence of NUMA effects, the experiments are deliberately confined to 16 threads on a single socket on Ice Lake and a single Sub-NUMA cluster on the Sapphire Rapids. In order to achieve consistent and reproducible outcomes for parallel applications runs, we bind OpenMP threads to physical cores using `OMP_PLACES` and `OMP_PROC_BIND`. Results presented in the following sections represent averages of 5 repetitions.

C. Experiments

1) Evaluating Allocation Interception Variants: For our experiments, we conducted tests with the aforementioned applications on both architectures using allocation interception and the following variants:

- **IDP-FCFS**: An initial data placement variant that employs a first-come-first-served approach.
- **IDP-RT**: An initial data placement variant that enforces the placement optimizations generated according to Section III-D2.
- **IDP-RT-LT**: An initial data placement variant that also considers individual allocation lifetimes according to Section III-D3.
- **PBDP-RT**: A phase-based data placement variant that enforces the placement optimizations generated according to Section III-D4.

Given that memory demands of those application runs are smaller than the available capacity of the fast memory, we explicitly enforced several capacity limitations that correspond to 25 %, 33 %, and 50 % of the individual application's memory footprint. Additionally, we report reference measurements, where all application data resides in the fast or slow memory. Results are depicted in Fig. 3.

Reviewing results on Ice Lake, we can see that XSBench and miniFE can profit from our *IDP-RT* solution, which correctly identifies the performance-critical data objects and prioritizes those for DRAM, achieving speedups of up to 2.48 × and 1.75 × over *IDP-FCFS*, respectively. Specifically in XSBench, *IDP-FCFS* will not place the most important item

Application	Parameters	# Inserted Transitions	# Effective Phases	Phase Length	Allocation Behavior	Suitable Approaches
XSBench (v20)	`-s large -l 100`	-	-	-	at start	IDP, IDP+Lifetime
miniFE 2.0	nx = ny = nz = 256	-	-	-	at start	IDP, IDP+Lifetime
LULESH 2.0	`-i 10 -s 200`	5	50	short-middle	at start and per phase	IDP+Lifetime, PBDP
NPB-BT (3.4.2)	CLASS = D	4	84	short	at start and per phase	IDP+Lifetime, PBDP
SPEC OMP 351.bwaves	double of size = train	5	94	short-middle	at start and per phase	IDP+Lifetime, PBDP
Multi-Phase	$30 \times$ 2400x2400 30 phases w/ 2 kernels	1	30	long	at start	PBDP

TABLE II: Parameters, problem sizes and further characteristics for codes tested during the evaluation. In addition, we indicate the number of phase transitions inserted, highlighting the minimal code modification effort involved, and suitable approaches that are expected to work well for those characteristics.

in fast memory due to the order of allocations. These findings and the performance are in line with what was achieved with the prior allocation abstraction approach [1], while requiring much less code modifications. As all allocations are done at the start and only one execution phase exists, there is no further gain using *IDP-RT-LT* or *PBDP-RT*.

Both 351.bwaves and LULESH allocate some data at the start but also further objects for specific execution phases. Hence, *IDP-RT* is not the best choice because it does not take allocation lifetimes into account during the placement optimization. Specifically for LULESH, we can observe poor performance compared to *IDP-FCFS*. Contrary, our extended *IDP-RT-LT* works much better, achieving a $1.25\times$ speedup over *IDP-FCFS* with $25\,\%$ capacity. Further, applying *PBDP-RT* gives another small performance boost leading to speedups of up to $1.38\times$. 351.bwaves presents a similar picture, although *PBDP-RT* only shows better performance than *IDP-RT-LT* in some scenarios. Although NPB-BT has quite similar characteristics, individual phase durations are shorter. Considering the next $P = 5$ upcoming phases yields good outcomes but is not able to outperform *IDP-RT-LT* as it results in identical placement decisions.

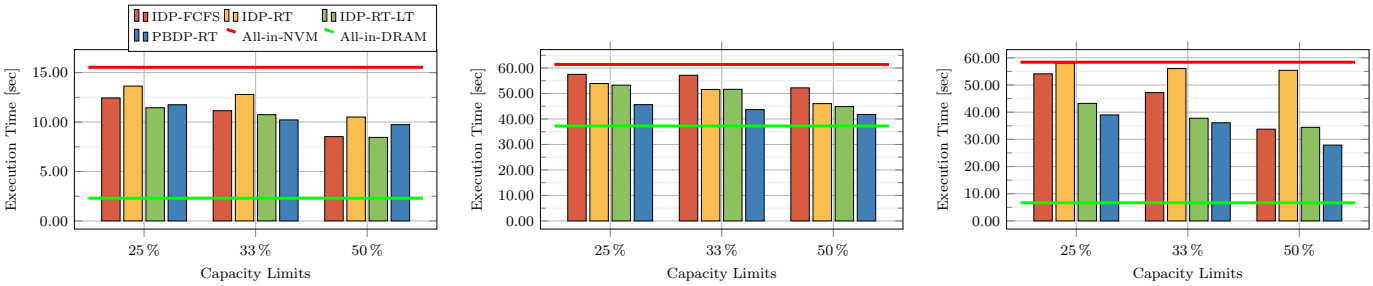
Multi-Phase allocates all data at the start but uses different buffers in every phase. As individual phase durations are much longer here, dynamically migrating data in between phases really pays off, outperforming other variants. As a comparison, considering $P = 2$ upcoming phases for *PBDP-RT* achieves an average speedup of $1.27\times$, while *IDP-RT-LT* only yields $1.13\times$. As an example, logs for a capacity limitation of $25\,\%$ report that *PBDP-RT* performed 19 (blocking) data migrations that overall took $2.35\,\mathrm{s}$ to ensure that phase-critical data resides in the faster memory. We are currently also working on non-blocking approaches to overlap migration costs with useful work (i.e., execution of other phases) to further boost performance.

Results on the Sapphire Rapids platform illustrate that the performance margin between the reference versions where all data is either in DRAM or HBM is much smaller, providing only limited potential for improvements. Although, in theory, HBM provides higher bandwidth, latency to DRAM is better in most instances. Hence, there is no single best memory for all cases. Codes that show access characteristics that are more sensitive to latency will therefore not work much better or might even experience performance declines. Similar behavior has already been observed with KNL's MCDRAM and reported in our related previous work [1].

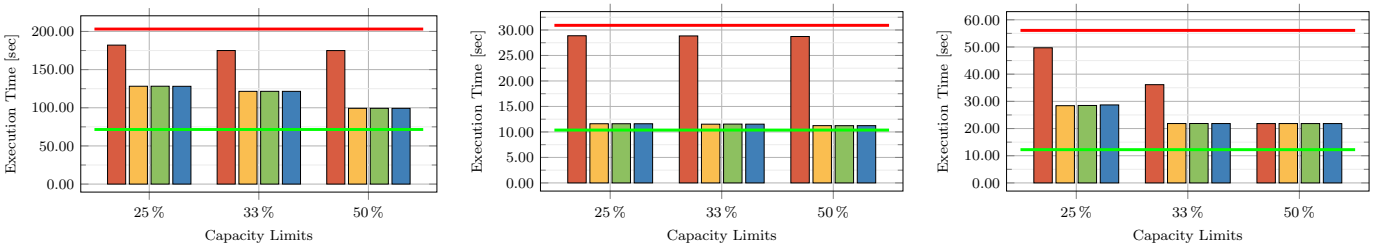
For LULESH and miniFE, our *IDP-RT-LT* and *PBDP-RT* approaches show better results than *IDP-FCFS* in almost all cases. For scenarios where the performance margin between HBM and DRAM is small, our variants are at least competitive with *IDP-FCFS*, and often reach outcomes close to the lower bound. For XSBench and Multi-Phase, we observe that placing all data in HBM is actually slower. That is a strong indicator that large portions of these codes are more sensitive to memory latency. Analyzing the JSON files revealed that, in those situations, our heuristics correctly identified that performance would be worse ($V_i \leq 0$). Hence, the ILP optimizer decided to set the corresponding $x_i = 0$, effectively allocating these data items in DRAM instead of HBM.

2) Tuning Phase-Based Data Placement: Next, we dive a bit deeper into phase-based data placement to highlight its strengths and weaknesses. As mentioned in Section III-D, the phase-based optimization heuristics take into account all profiling data (memory accesses) from allocations within the next P phases. This allows to put emphasis on heavily used data items in directly upcoming computational regions and identify performance-critical objects that should be placed in fast memory. However, depending on the value P this can also lead to a restricted view of future events and unfavorable placement decisions.

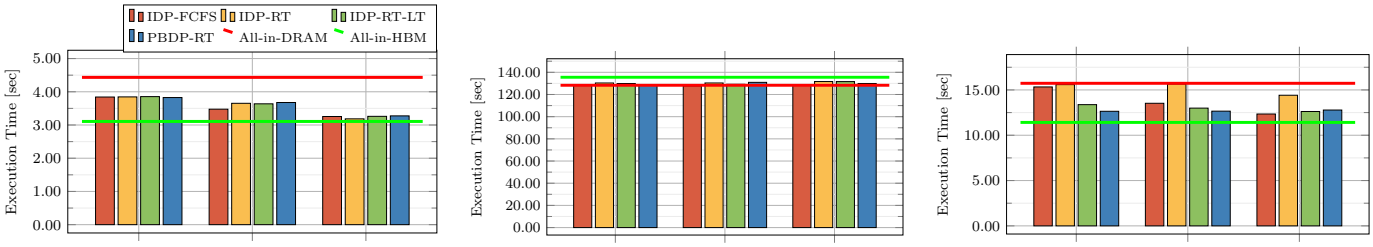
Lets consider the example illustrated in Fig. 4 and the case that only profiling data from the next phase is taken into account ($P = 1$). Optimizing the data placement in phase transition *T1* will then only respect allocations `A`, `B` and `C` and memory accesses overlapping with *Phase 1*, and allocate items accordingly, for example, by placing objects `B` and `C` in fast memory. If we now reach phase transition *T2*, and it turns out that the performance impact of objects `D` and `E` is much higher, there are two potential outcomes: 1) the estimated benefit of the new placement outweighs the costs for migrating objects, or 2) migrating data is too costly and `B` and `C` will stay in fast memory, which could negatively affect the performance of subsequent phases. In the latter case, choosing $P = 2$ would avoid the problem by directly prioritizing `D` and `E` for fast memory. As an example, NPB-BT on Ice Lake with $33\,\%$ memory capacity and $P = 1$ runs into that issue resulting in



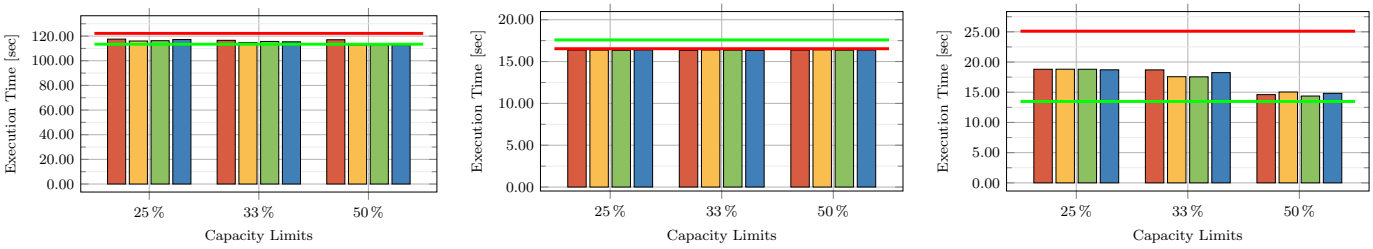
(a) SPEC 351.bwaves on Ice Lake (b) Multi-Phase on Ice Lake (c) LULESH on Ice Lake



(d) NPB-BT on Ice Lake (e) XSBench on Ice Lake (f) miniFE on Ice Lake



(g) SPEC 351.bwaves on Sapphire Rapids (h) Multi-Phase on Sapphire Rapids (i) LULESH on Sapphire Rapids



(j) NPB-BT on Sapphire Rapids (k) XSBench on Sapphire Rapids (l) miniFE on Sapphire Rapids

Fig. 3: Performance comparison of a first-come-first-served allocation scheme and allocation interception variants that utilize initial and phase-based data placement optimizations on Ice Lake and Sapphire Rapids.

203.19 s, while $P = 5$ takes better decisions and only runs in 121.47 s.

V. Related Work

A. Profiling Memory Access

Two fundamental approaches are employed to collect information about data accesses. The first approach entails tracking memory allocations and instrumenting individual memory accesses during the compilation phase. For example, the *AddressSanitizer* [21] in LLVM instruments memory accesses to detect various memory access errors. While this technique provides a detailed view, it generates substantial data and introduces significant overhead. Additionally, the instrumentation process may interfere with compiler optimizations, potentially altering the program's behavior compared to the uninstrumented executable [21].

The second approach also tracks memory allocations but employs a sampling technique, periodically interrupting program execution to gather information about memory accesses. This approach offers several advantages such as its applicability to unmodified executables, minimal impact on program behavior, and reasonable cost. Furthermore, most hardware

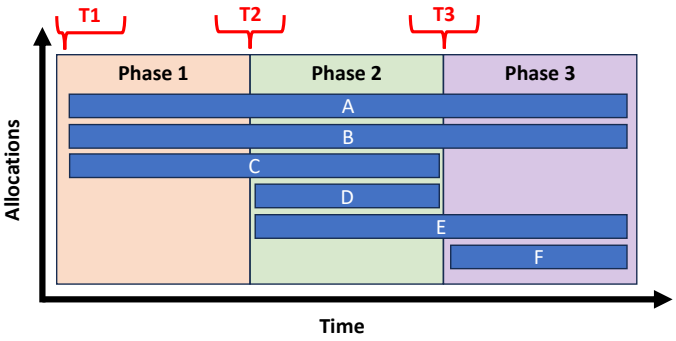


Fig. 4: Example illustration of allocation lifetimes, execution phases and phase transitions.

vendors support *Hardware Performance Counters* (HWC), which count hardware events like *Loads* and *Stores*. On recent architectures, these counters have been extended to provide information about which thread accessed a memory location and whether the access was serviced from cache or main memory, offering insights into realistic execution behavior. On Intel and AMD platforms, this feature is referred to as *Precise Event Based Sampling* (PEBS) and *Instruction Based Sampling* (IBS), respectively. Due to certain hardware design and caching choices (typically temporal stores), this information is limited for store events as cannot detect whether store accesses go to main memory, but it can be approximated by considering L1D misses, which is also done in this work.

B. Data Allocation and Migration

Many *programming languages* require users to manage memory using language-provided facilities. For example, C++ offers the `new` / `delete` operators, and Fortran 90 provides the `ALLOCATE` / `DEALLOCATE` statements. Notably, C's `malloc` function is part of the Standard Library rather than the language itself. These built-in memory facilities were designed decades ago and may not effectively address the diverse memory configurations of today.

Efforts have been made to create *software libraries* that expose memory system functionality and provide specialized behavior. For instance, `libnuma` [22] allows querying memory locations (NUMA nodes) and specification of user-level preferences regarding page placement. However, it manipulates NUMA nodes without considering their characteristics, such as performance and capacity. It serves as the low-level interface to memory binding system calls on Linux. The *memkind* library [23] builds upon NUMA node functionality to enable allocations of high-bandwidth memory within C code. Initially designed for KNL, it has been later extended to discover NVM nodes, relying on work in hwloc for identifying generic HBM nodes portably. SICM [24] conducts architecture profiling to identify DRAM, HBM, or NVM memory nodes. This step is challenging as it must detect bandwidth and latency differences, as well as read/write asymmetry. Our opinion is that, utilizing hwloc to read performance attributes from the HMAT ACPI table is a more reliable way to identify and characterize different node types.

As elaborated in Section II-B, most research works on heterogeneous memory disregard the challenge of identifying which NUMA node is fast or slow. They assume prior knowledge of platform details and focus on optimizing memory placement. Servat et al. [25] use Extrae [26] to track memory allocations and PEBS to sample load and store instructions, analyzing these traces to determine which memory objects should be allocated in high bandwidth memory on KNL. Jordà et al. [27] also rely on Extrae and PEBS to track memory accesses. They prescribe initial data placement for data objects on a Xeon platform with Optane NVM through post-mortem optimization and allocation interception via an `LD_PRELOAD` mechanism, similar to what is done in this work. However, their heuristic solely focuses on a bandwidth-aware performance model to reduce CPU stall cycles. Contrary, our work takes both bandwidth and latency of underlying platforms into account. Narayan et al. [28] conduct a post-mortem analysis of memory accesses based on cache misses and reorder-buffer stall times to classify objects for generic heterogeneous memory platforms. Other works utilize PEBS [29]–[31] or other profiling methods [32] for data placement optimization, either post-mortem or online but it does not feature allocation interception. Moreover, some works are limited to specific programming models or program structures.

Our approach builds upon some of these concepts by incorporating traits that can be determined either dynamically by the runtime or provided explicitly by the user. We have the hypothesis that application developers often possess insights into which memory buffers may be sensitive to bandwidth or latency considerations, thereby we enable them to assist the runtime in making informed placement decisions. In cases where such user input is unavailable, we resort to established profiling and optimization techniques for suggesting suitable placements. Moreover, it is important to emphasize that our solution operates without making any assumption about the structure of the application itself or the underlying platform. This adaptability makes our approach highly versatile and capable of accommodating changes in the configuration of heterogeneous memory systems, or fluctuations in available memory capacities. This adaptability allows us to make use of fast memory as much as possible when it is available, while seamlessly transitioning to larger capacity memory when necessary.

C. Relation to OpenMP

OpenMP as the de-facto standard for shared memory parallel programming has addressed heterogeneous memory by incorporating allocation support for different kinds of memory. We have contributed to that as active members of the OpenMP Language Committee. In OpenMP, a *memory space* encapsulates a storage resource that is available in the system, in particular represented by different kinds of memory. For each of these spaces, there is also a pre-defined *allocator* available. A set of API routines to create and destroy memory spaces

and allocators, taking a set of traits as an argument, is part of OpenMP. The *allocator traits* can be employed to specify different aspects of an allocator's behavior. The memory kinds and the memory and allocator traits were defined to provide a reasonable level of abstraction (example: `omp_high_bw_mem_space` to select memory with high bandwidth) to be independent of individual memory technologies. In summary, the OpenMP memory management API provides a portable and vendor-neutral functionality to request allocation of memory in a certain memory kind and the selection of that memory kind based on the specification of certain properties as traits.

Our work follows the direction of these concepts by employing allocation routines, using traits to specify desired properties, and dealing with different system memories in the same way as OpenMP's kinds of memory. However, OpenMP's functionality is limited to selecting a certain kind of memory at an allocation request. Some tasks that are crucial to achieve high performance and efficiency on heterogeneous memory systems continue to be the sole responsibility of the programmer. This includes, in particular, dealing with limited capacity of the preferable (fast) kind of memory, as for most application codes the capacity of the high-bandwidth memory is too small to hold all data elements. In that case, the application requires an optimization strategy to select a subset of data to go into the preferable memory. In addition, such a strategy might want to change the selection of data over time by employing data migration between memory kinds, depending on application phases that exhibit changing characteristics. The solution of this complex optimization problem is addressed by our work and we hope that our findings will influence the further development of OpenMP in the future.

VI. Summary and Conclusion

In this work, we presented an extended methodology to automatically optimize initial and phase-based data placement on systems with heterogeneous memory. Our approach is based on abstracting or intercepting regular memory allocations to be able to provide additional requirements or hints (such as placement decisions) that can be exploited by the runtime system to select a suitable storage location. We presented a data-driven workflow that utilizes memory access profiling for existing applications and a placement optimizer. We formalized the data placement optimization problems for initial and phase-based data placement, facilitating metrics and heuristics based on the collected profiling data and considering individual application execution phases. Finally, we evaluated our approaches with six different applications on two recent heterogeneous memory systems, an Intel Ice Lake system (DRAM+NVM) and an Intel Sapphire Rapids system (HBM+DRAM), and demonstrated that our heuristics can efficiently manage the data placement on such architectures, outperforming a first-come-first-served approach in most instances. Our *IDP-RT-LT* strategy presents a viable and stable solution for most codes, while dynamic data migration at run time with *PBDP-RT* can provide further speedups for applications with coarse grained execution phases.

There are some natural directions for future work. Although information obtained from memory access profiling is valuable, it does not directly provide the information of which data objects are used together in performance-critical regions or in the same calculation. Our plan is to combine memory access profiling with regular tracing to identify such instances and improve our heuristics and placement decisions. Additionally, we are exploring an integration of non-blocking data migrations or prefetching mechanisms for our phase-based approach to hide transfer costs and overlap computation and data migration as much as possible. Currently, our main objective is optimizing the execution time. As our modular architecture of the placement optimizer allows customization of metrics and optimization objectives, an interesting aspect would be to combine performance-related and power-related heuristics to construct a combined optimization problem to reduce overall energy consumption.

Acknowledgment

We would like to thank Intel for providing access to the Sapphire Rapids platform. Further, We would like to thank François Trahay for modifications and for merging our changes into NumaMMA to better suit the requirements of our work.

REFERENCES

[1] J. Klinkenberg, A. Kozhokanova, C. Terboven, C. Foyer, B. Goglin, and E. Jeannot, "H2M: Exploiting heterogeneous shared memory architectures," *Future Generation Computer Systems*, vol. 148, pp. 39–55, 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X23002029

[2] R. Smith, "Hot Chips 2016: Memory Vendors Discuss Ideas for Future Memory Tech—DDR5, Cheap HBM, & More," Aug. 2016, [Online]. Available: http://www.anandtech.com/show/10589/hot-chips-2016-memory-vendors-discuss-ideas-for-future-memory-tech-ddr5-cheap-hbm-more.

[3] A. Sodani, R. Gramunt, J. Corbal, H. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agarwal, and Y. Liu, "Knights Landing: Second-Generation Intel Xeon Phi Product," *IEEE Micro*, vol. 36, no. 2, pp. 34–46, Mar. 2016.

[4] T. Barnes, B. Cook, J. Deslippe, D. Doerfler, B. Friesen, Y. He, T. Kurth, T. Koskela, M. Lobet, T. Malas, L. Oliker, A. Ovsyannikov, A. Sarje, J. Vay, H. Vincenti, S. Williams, P. Carrier, N. Wichmann, M. Wagner, P. Kent, C. Kerr, and J. Dennis, "Evaluating and Optimizing the NERSC Workload on Knights Landing," in *2016 7th International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, Nov. 2016, pp. 43–53.

[5] A. Biswas, "Sapphire Rapids," in *2021 IEEE Hot Chips 33 Symposium (HCS)*. IEEE Computer Society, 2021, pp. 1–22.

[6] I. B. Peng, M. B. Gokhale, and E. W. Green, "System Evaluation of the Intel Optane Byte-Addressable NVM," in *Proceedings of the International Symposium on Memory Systems*, ser. MEMSYS '19. New York, NY, USA: Association for Computing Machinery, Oct. 2019, pp. 304–315.

[7] A. C. Elster and T. A. Haugdahl, "Nvidia Hopper GPU and Grace CPU Highlights," *Computing in Science & Engineering*, vol. 24, no. 2, pp. 95–100, 2022.

[8] D. Das Sharma and S. Tavallaei, "Compute Express Link 2.0 White Paper," Nov. 2020.

[9] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, and R. Namyst, "hwloc: A generic framework for managing hardware affinities in HPC applications," *Proceedings of the 18th Euromicro Conference on Parallel, Distributed and Network-Based Processing, PDP 2010*, pp. 180–186, 2010.

[10] B. Goglin and A. Rubio Proaño, "Using Performance Attributes for Managing Heterogeneous Memory in HPC Applications," in *The 23rd IEEE International Workshop on Parallel and Distributed Scientific and Engineering Computing (PDSEC 2022), held in conjunction with IPDPS 2022*. Lyon, France: IEEE, May 2022, pp. 890–899. [Online]. Available: http://hal.inria.fr/hal-03599360

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

[12] F. Trahay, M. Selva, L. Morel, and K. Marquet, "NumaMMA: NUMA MeMory Analyzer," in *Proceedings of the 47th International Conference on Parallel Processing*, ser. ICPP 2018. New York, NY, USA: Association for Computing Machinery, 2018, pp. 1–10.

[13] Intel, "Memory Latency Checker (MLC)." [Online]. Available: http://www.intel.com/software/mlc

[14] I. Karlin, A. Bhatele, J. Keasler, B. L. Chamberlain, J. Cohen, Z. DeVito, R. Haque, D. Laney, E. Luke, F. Wang, D. Richards, M. Schulz, and C. Still, "Exploring Traditional and Emerging Parallel Programming Models using a Proxy Application," in *27th IEEE International Parallel & Distributed Processing Symposium (IEEE IPDPS 2013)*, Boston, USA, May 2013.

[15] I. Karlin, J. Keasler, and R. Neely, "LULESH 2.0 Updates and Changes," Tech. Rep. LLNL-TR-641973, August 2013.

[16] P. T. Lin, M. A. Heroux, R. F. Barrett, and A. B. Williams, "Assessing a mini-application as a performance proxy for a finite element method engineering application," *Concurrency and Computation: Practice and Experience*, vol. 27, no. 17, pp. 5374–5389, 2015. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.3587

[17] J. R. Tramm, A. R. Siegel, T. Islam, and M. Schulz, "XSBench – the development and verification of a performance abstraction for Monte Carlo reactor analysis," in *PHYSOR 2014 - The Role of Reactor Physics toward a Sustainable Future*, Kyoto, 2014, pp. 1–12. [Online]. Available: https://www.mcs.anl.gov/papers/P5064-0114.pdf

[18] P. K. Romano and B. Forget, "The OpenMC Monte Carlo particle transport code," *Annals of Nuclear Energy*, vol. 51, pp. 274–281, 2013.

[19] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga, "The Nas Parallel Benchmarks," *The International Journal of Supercomputing Applications*, vol. 5, no. 3, pp. 63–73, 1991. [Online]. Available: https://doi.org/10.1177/109434209100500306

[20] M. S. Müller, J. Baron, W. C. Brantley, H. Feng, D. Hackenberg, R. Henschel, G. Jost, D. Molka, C. Parrott, J. Robichaux, P. Shelepugin, M. van Waveren, B. Whitney, and K. Kumaran, "SPEC OMP2012 – an application benchmark suite for parallel systems using OpenMP," in *Proceedings of the 8th International Conference on OpenMP in a Heterogeneous World*, ser. IWOMP'12. Berlin, Heidelberg: Springer-Verlag, 2012, p. 223–236. [Online]. Available: https://doi.org/10.1007/978-3-642-30961-8_17

[21] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "AddressSanitizer: A fast address sanity checker," in *2012 USENIX Annual Technical Conference (USENIX ATC 12)*. Boston, MA: USENIX Association, Jun. 2012, pp. 309–318. [Online]. Available: https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany

[22] A. Kleen, "A NUMA API for Linux," *Novel Inc*, 2005.

[23] C. Cantalupo, V. Venkatesan, J. Hammond, K. Czurlyo, and S. D. Hammond, "memkind: An Extensible Heap Memory Manager for Heterogeneous Memory Platforms and Mixed Memory Policies," Sandia National Laboratories (SNL-NM), Albuquerque, NM (United States), Tech. Rep., 2015.

[24] M. K. Lang, "Simplified Interface to Complex Memory (SICM) FY19 Project Review," Oct. 2019.

[25] 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 *Proceedings of the IEEE International Conference on Cluster Computing*, Hawaii, USA, Sep. 2017, pp. 126–136.

[26] BSC Performance Tools, "Extrae," https://tools.bsc.es/extrae.

[27] M. Jordà, S. Rai, E. Ayguadé, J. Labarta, and A. J. Peña, "ecoHMEM: Improving Object Placement Methodology for Hybrid Memory Systems in HPC," in *2022 IEEE International Conference on Cluster Computing (CLUSTER)*, 2022, pp. 278–288.

[28] A. Narayan, T. Zhang, S. Aga, S. Narayanasamy, and A. K. Coskun, "MOCA: Memory Object Classification and Allocation in Heterogeneous Memory Systems," in *Proceedings of the 2018 IEEE International Parallel and Distributed Processing Symposium*. Vancouver, BC, Canada: IEEE, May 2018, pp. 326–335.

[29] S. Wen, L. Cherkasova, F. X. Lin, and X. Liu, "ProfDP: A Lightweight Profiler to Guide Data Placement in Heterogeneous Memory Systems," in *Proceedings of the 2018 International Conference on Supercomputing*, ser. ICS '18. New York, NY, USA: Association for Computing Machinery, 2018, pp. 263–273.

[30] 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: Association for Computing Machinery, 2017, pp. 1–14.

[31] K. Wu, J. Ren, and D. Li, "Runtime data management on non-volatile memory-based heterogeneous memory for task-parallel programs," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, ser. SC '18. IEEE Press, 2019, pp. 1–13.

[32] M. Laghari, N. Ahmad, and D. Unat, "Phase-Based Data Placement Scheme for Heterogeneous Memory Systems," in *2018 30th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, 2018, pp. 189–196.