Using Performance Attributes for Managing Heterogeneous Memory in HPC Applications

Brice Goglin, Andrès Rubio Proaño

Using Performance Attributes for Managing Heterogeneous Memory in HPC Applications

Brice Goglin Andrès Rubio Proaño
Inria, LaBRI, Univ. Bordeaux
Talence, France
{brice.goglin,andres.rubio}@inria.fr

***Abstract*—The complexity of memory systems has increased considerably over the past decade. Supercomputers may now include several levels of heterogeneous and non-uniform memory, with significantly different properties in terms of performance, capacity, persistence, etc. Developers of scientific applications face a huge challenge: efficiently exploit the memory system to improve performance, but keep productivity high by using portable solutions.**

In this work, we present a new API and a method to manage the complexity of modern memory systems. Our portable and abstracted API is designed to identify memory kinds and describe hardware characteristics using metrics, for example bandwidth, latency and capacity. It allows runtime systems, parallel libraries, and scientific applications to select the appropriate memory by expressing their needs for each allocation without having to re-modify the code for each platform. Furthermore we present a survey of existing ways to determine sensitivity of application buffers using static code analysis, profiling and benchmarking. We show in a use case that combining these approaches with our API indeed enables a portable and productive method to match application requirements and hardware memory characteristics.

***Index Terms*—heterogeneous memory, multi-level memory, NUMA, performance attributes, profiling, benchmarking, code analysis**

I. Introduction

The architecture of supercomputers has become more and more complex over the last years in order to be able to support the increasing computing needs. After multicore and NUMA architectures, the memory subsystem is now becoming heterogeneous with the advent of multi-level memories. A few years ago, the Intel *Knights Landing* Xeon Phi (KNL) [1] introduced a two-level memory architecture mixing normal memory (DRAM) and some low-capacity high-bandwidth memory (MCDRAM). Due to their very different characteristics, the choice between these memories became essential for developers, since the performance of applications strongly depends on it. This complexity is found today in standard platforms with for example the emergence of non-volatile memories providing a large capacity but lower performance than the usual DRAM memory.

Many research projects addressed the management of the heterogeneous memory systems of KNL in HPC runtimes and applications. New memory management APIs were proposed as well as methods for determining the sensitivity of application buffers to high-bandwidth memory. Unfortunately, most of these approaches were specific to KNL. They cannot directly be applied to other heterogeneous memory platforms where the number, the locality and the characteristics of memory nodes are not known in advance. We believe there is a strong need for portable software solutions that do not assume any of these but rather expose memory information in an abstracted way to make applications portable again.

In this article, we introduce a new API to help manage the complexity of these memory architectures. It first consists in a generic and portable way to identify the different kinds of memories in a platform and their locality. Then it exposes a set of attributes describing the characteristics of these NUMA nodes, especially in terms of performance and capacity. This abstraction of the behavior of memory nodes provides application and runtime developers with a productive way to adapt their memory allocation to the hardware platforms without sacrificing the portability. We implemented this interface in hwloc, the *de facto* standard tool to expose the locality of hardware resources in high performance computing.

The remainder of this paper is organized as follows. Section II describes the hardware landscape of heterogeneous memory and how it is currently managed in software in the literature. Our proposal for identifying and describing memory kinds with a set of attributes is then presented in Section III before the implementation as a new hwloc API extension is detailed in Section IV. We then perform a survey of existing methods for determining the sensitivity of application buffers to different kinds of memory in Section V. We combine these ideas with our implementation in Section VI to confirm that our proposal indeed brings a portable and productive way to adapt buffer placement. Section VII discusses the limits of our approach before we present conclusions and future work.

II. Context and State of the Art

The increasing number of cores in processors puts the memory subsystem of computing platforms under severe pressure. This *Memory Wall* caused hardware architects to design more complex memory subsystems with multiple levels of heterogeneous memory. Hence, managing memory in HPC applications is getting more difficult since an extra step must now be performed for deciding where the applications should allocate each data buffer. We introduce these heterogeneous memory platforms in this section and discuss the solutions and issues for managing them in the HPC software stack.

A. High Bandwidth Memory

High-Bandwidth memory (HBM) is a new memory technology that appeared several years ago to alleviate the memory wall. This comes at the cost of much smaller capacities. Hence, system designers have to find a trade-off between performance and capacity. Multi-level memory is a popular way to address this trade-off, for instance by combining the low-capacity HBM with a slower but higher-capacity *normal* memory (DRAM).

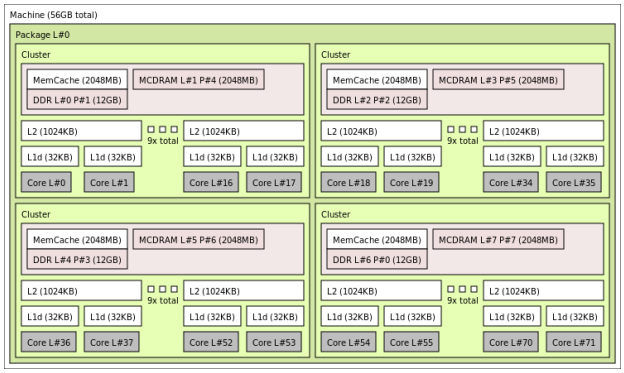


Fig. 1: Output of hwloc's `lstopo` tool on a Xeon Phi configured in SNC4/Hybrid50 mode. Each *SubNUMA Cluster* contains 18 cores with 2 local memories: 12GB of DRAM (behind a 2GB memory-side cache), and 2GB of MCDRAM.

The launch of the Intel *Knights Landing* Xeon Phi processors (KNL) in 2016 first brought multi-level memories to HPC. The traditional memory (DRAM) was accompanied with a faster but smaller memory (called MCDRAM, similar to HBM) [1]. Since it was not clear how such an architecture would be best used in terms of performance, the processor supported several configuration modes. The MCDRAM could be used as a hardware-managed cache in front of the DRAM (*Cache* mode), or both memories could be exposed to the developer so that he manually decides where to allocate (*Flat* mode). A hybrid mode was also supported (see Figure 1).

KNL introduced an important new trade-off between performance and productivity: the Cache mode is an automatic hardware-based way to benefit from MCDRAM performance and DRAM capacity, but its performance may be lower than the Flat mode if the application memory allocations are carefully tuned for this platform. This tuning requires identifying performance-critical data buffers to allocate the important ones on MCDRAM to benefit from high bandwidth, and let others benefit from the larger DRAM capacity [2].

B. Non-Volatile Memory

Non-Volatile Memory is an emerging technology that will supposedly close the gap between memory and storage. It provides users with non-volatile memory DIMMs (NVDIMMs[1]) that may be used either as fast persistent storage, or as high-capacity memory. When used with Intel Xeon processors, 2 configuration modes are supported. *2-Level-Memory* exposes NVDIMMs as normal/volatile memory and DRAM acts as a hardware-managed cache in front of them (similar to KNL *Cache* mode). *1-Level-Memory* exposes DRAM as the main memory and NVDIMMs as storage, which the operating system can convert into additional NUMA nodes (see Figure 2, similar to KNL *Flat* mode). [2]

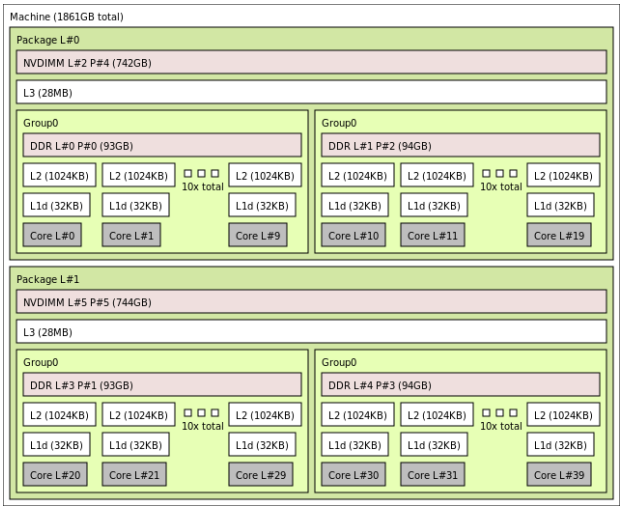


Fig. 2: Output of hwloc's `lstopo` tool on a dual Xeon 6230 with 384GB of DRAM (96GB per *SubNUMA Cluster* of 10 cores) and 1.5TB of NVDIMMs (768GB per CPU). NVDIMMs are configured in *1-Level-Memory* and exposed to applications as additional NUMA nodes.

These platforms bring the trade-off between performance and productivity again: Is it better to let the hardware manage the DRAM as a cache? How should applications decide what to allocate on DRAM and what to allocate on NVDIMMs?

C. Heterogeneous Memory in Future HPC Platforms

With KNL being obsolete nowadays, one may wonder whether HBM is still a relevant memory technology for HPC. HBM remains the best solution to feed CPUs with increasing number of cores. It is used in the Fugaku pre-exascale supercomputer. However, this platform does not combine HBM with any other kind of memory, hence there is no actual need for performance/productivity trade-off. However, the HBM capacity may be too low to avoid a combination with another kind of slower but larger memory, either DRAM-like or NVDIMM. Two ARM HPC-AI initiatives already revealed that they will combine on-package HBM with off-package DDR5: ETRI's K-AB21 processor, and SiPearl's Rhea exascale processor.

HBM is also used in GPUs and other powerful co-processors. Platforms such as POWER9 may expose NVIDIA V100 GPU memory as separate host NUMA nodes, and GPUs are able to access the host DRAM. This uniform view of the entire platform memory implies that buffers may be allocated anywhere and accessed from both CPUs and GPUs.

[1] Intel Optane DataCenter Persistent Memory Modules in our platform.

[2] A mix is also supported if NVDIMMs are partitioned.

Hence, applications need to decide where to allocate buffers depending on performance and capacity requirements and where they will be primarily accessed from.

Additionally some more exotic kinds of memory may emerge in the future. For instance, network-attached memory has been proposed to offer very high-capacity to compute nodes in disaggregated environments where memory resources are allocated to jobs on demand. Such technologies also likely trade-off performance for capacity (allocating buffers first requires to check whether the remote memory is appropriate).

D. State of the Art

Based on the memory landscape we described in previous sections, we believe that there is a clear need for software solutions for managing these new heterogeneous memory platforms. The basic way to allocate on specific kinds of memory is to bind the entire process on a memory node. Unfortunately, this cannot work for HBM-enabled platforms because the HBM capacity is usual too low to store all data buffers. Hence, there is a need to determine performance critical buffers and only allocate them on small memory nodes. AutoHBW [3] proposed to allocate buffers in HBM or DRAM depending on their size without having to modify the application code. However, this is only a convenience solution that still requires to identify sensitive buffers and their size for a specific run.

Since the appearance of KNL, several interfaces have been proposed to allocate explicitly in fast or slow memory, for example memkind [3]. However, this API was designed for KNL. It hardwires the difference between HBM and conventional memory instead of providing explicit performance-related criteria that would also work on platforms with DRAM and slower memory, e.g. NVDIMMs. Moreover, it does not take NUMA locality into account, which means slow local memory cannot be compared with fast remote memory.

Some more advanced strategies take into account the access patterns of the applications. Servat [4] and Narayan [5] (MOCA) use a post-mortem analysis of memory accesses based, for example, on hardware counters. FLEXMALLOC [6] similarly proposes to replace dynamic allocations at runtime. SICM [7] rather proposes both a low-level allocation API and a high-level data management interface.

All these approaches require knowledge of which NUMA nodes are *fast* before adapting the memory allocation. However, this **identification and characterization** step is not discussed in the literature. Indeed, the KNL configuration was known in advance and could be hardwired in applications. However, this is no longer possible with generic platforms with HBM and/or NVDIMMs. Users have to manually describe the characteristics of each memory node whenever they start using a new platform. SICM [7] rather performs an *Architecture Profiling* step to guess the memory organization but this still hinders productivity and portability.

As explained later in Section III-B, we believe this identification step should be performed automatically during the execution and it must support any type of memory. After identification, there is a need for **exposing performance characteristics** to describe each memory kind in terms of capacity, bandwidth, latency, etc.

E. Proposal

Applications could be considered as a set of memory buffers that must be allocated somewhere. Each buffer may lead to different performance when allocated in different kinds of memory. Hence, they have an **affinity for some kind(s) of memory**, just like they may have an affinity for some location depending on which CPU accesses them. This requires an **analysis of the application behavior** to determine the affinity of the data buffers.

The method that we describe in the rest of the paper proposes to implement these steps to provide application developers with a high productivity environment for supporting heterogeneous memory in a portable manner: (i) Describe application needs, (ii) Gather hardware characteristics, (iii) Match them when allocating buffers.

Such a method can be either implemented directly in the application, or in the runtime system that executes it. For instance, OpenMP already plans to expose different kinds of memory to applications [8].

III. Exposing the Memory Characteristics

We describe in this section how to identify memory kinds and expose their characteristics so that runtimes and applications know where to allocate performance-critical buffers.

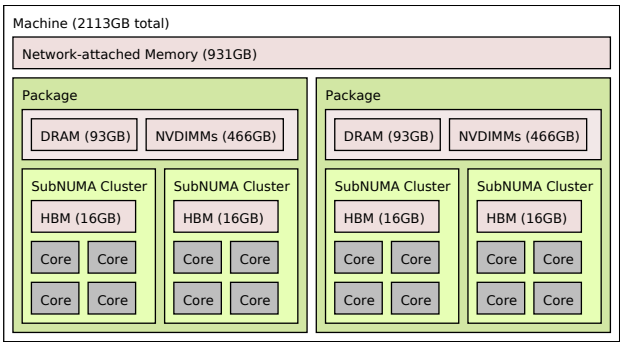


Fig. 3: Output of hwloc's `lstopo` tool on a fictitious platform with several kinds of memory: Each CPU package has local NVDIMM and DRAM NUMA nodes. Each SubNUMA Cluster in those packages also has a HBM. And a network-attached memory[3]is also connected to the entire machine.

Most parallel computing runtimes rely on hwloc [9] to discover the topology of HPC platforms. It builds a hierarchy of objects based on inclusion and physical locality. Since hwloc 2.0, memory objects are attached to the CPU hierarchy to show which cores are local to a given NUMA node [10], hence properly modeling of multi-level memory architectures.

Four types of memories[4] are available in Figure 3. A program running on a core therefore has 4 local NUMA

[3]Examples of network-attached memory are presented at https://kove.net and https://www.deep-projects.eu/hardware/memory-hierarchies/49-nam.

[4]Currently, commercial platforms support up to 3 simultaneously.

nodes to allocate its memory buffers. Hence, there is a need to provide a simple way to choose between them. At the processor level (*Package*), hwloc exposes DRAM memory (conventional memory) before NVDIMMs because DRAM is usually the default allocation node. However this choice does not meet the needs of all applications, for example for capacity reasons. Bandwidth-bound applications may also prefer allocating to HBM by default. Unfortunately, hwloc cannot easily expose the HBM before the DRAM on this platform because they are not attached to the same locality (*SubNUMA Cluster* vs *Package*). Moreover, there was previously no hwloc API to expose performance information, such as latency or bandwidth. We describe later in this article how we extended hwloc to provide applications with such information.

A. Identifying Memories

Identifying memory kinds for instance consists in understanding which NUMA nodes are (fast) HBM, (normal) DRAM, or high-capacity (slow) NVDIMMs. On a platform such as Figure 2, how does an application know that first 2 NUMA nodes are DRAM while the other are NVDIMMs? It could be 2 DRAMs and 4 HBMs instead. This was hardwired in applications on KNL but high productivity on different modern platforms now requires a portable solution.

hwloc identifies kinds of memory by querying Linux kernel drivers but it is only meant for debugging with human-readable information. This identification is not explicitly exposed by hardware ACPI tables or operating systems such as Linux[5] because performance may vary from one technology to another.[6] The actual safe way for user-space to select the appropriate memory node for allocation is to compare performance attributes. This is the reason why we proposed to characterize memories by a set of attributes and metrics [11].

B. Characterizing Memories with Performance Metrics

1) Bandwidth: With architectures like KNL including high-capacity memory (DRAM) and high bandwidth memory (MCDRAM), bandwidth sensitive applications should be able to allocate data buffers in the most appropriate memory.

We propose to **characterize memory targets with a bandwidth value** (or even one for *Read* or one for *Write* if necessary). These values let us rank memories when bandwidth is the priority criterion for an allocation.

On a system with HBM, DRAM and non-volatile memory (NVDIMM), for example, we would rank as follows:

$$HBM_{BW} > DRAM_{BW} > NVDIMM_{BW} \qquad (1)$$

2) Latency: Bandwidth and latency may not be correlated in practice. For example, on KNL, HBM and DRAM have similar latencies[7] while their bandwidth is very different. In addition, the latency is considered very important for HPC applications nowadays [12]. For instance, graph-based or *Pointer Chasing*-type applications benefit much more from low latency than from high bandwidth.

Hence, we also **characterize memories by latency** (possibly separating *Read* and *Write* latencies). It would lead, when latency is the priority criterion, to the following order: [8]

$$DRAM_{Lat} \simeq HBM_{Lat} > NVDIMM_{Lat} \qquad (2)$$

On this KNL platform, the application will not know if it should allocate on DRAM or HBM since their priority are similar. But it can look at other criteria such as the capacity to finalize its choice.

3) Capacity: The capacity of different memories is obviously an important criterion for choosing where to allocate, particularly when the memories are small, for example 16GB of HBM on KNL. The order by capacity is usually:

$$NVDIMM_{Cap} > DRAM_{Cap} > HBM_{Cap} \qquad (3)$$

If several applications are running on the same machine, their dynamic behavior could moreover impose to consider the available capacity rather than the total capacity.

IV. Implementation in hwloc

We implemented the aforementioned ideas as a new hwloc API extension. As is shown in Figure 4, it exposes the memory characteristics and ordering according to our proposed criteria, for example the latency[9]. The API is based on *Initiators* that perform accesses to *Memory Targets*[10]. Hence, an application usually first selects the targets that are local to the core(s) where it runs (*NUMA Affinity*), and then compares their values for some attributes (*Memory Kind Affinity*). An *Attribute* corresponds to the priority criteria when deciding where to allocate. This API could actually also be used for homogeneous NUMA platforms since latency or bandwidth indicate whether NUMA nodes are close or far away from cores.

The function `get_best_target()` determines the best target for an allocation from a given core using the ranking for the given attribute (while a `get_best_initiator()` finds the cores with best performance for a given target). If allocating fails (for example if the memory is full), or if more complex decision criteria are needed (e.g. weighted latency and bandwidth), or if NUMA-locality is not strictly required, one may fall back to `get_value()` for manually comparing targets. Moreover, allocating parts of arrays in different targets or migrating is possible using existing hwloc features combined with this new API.

Figure 5 presents an example of information available through this interface. hwloc provides several default attributes that are already supported by the operating systems (capacity and locality) and by some platforms (bandwidth and latency).

[5]https://lkml.org/lkml/2019/3/25/1020

[6]Some NVDIMM technologies are not slower than DRAM.

[7]But the latencies of HBM and DRAM depend on the concurrency load.

[8]This order compares the priority of memories for latency-sensitive allocations, and not the latency itself (the **weaker** latency is, the more priority that memory has, the further left it appears in our equation).

[9]The full interface is available in https://github.com/open-mpi/hwloc/blob/master/include/hwloc/memattrs.h and released in hwloc version 2.3.

[10]*Initiator* and *Target* are the modern terms replacing the obsolete *NUMA node* composed of both CPU cores and memory.

Get the array of memory targets that are local to a given initiator:

```
hwloc_get_local_numanode_objs(topology, initiator, &nr, &targets)
```

Get the best memory target (and its value) for the given initiator and attribute:

```
hwloc_memattr_get_best_target(topology, attribute, initiator, &best_target, &target_value)
```

Get the value of an attribute for the given memory target and initiator:

```
hwloc_memattr_get_value(topology, attribute, target, initiator, &value)
```

Fig. 4: Summary of main hwloc API functions for manipulating memory attributes. Initiators are either sets of logical processors (CPU-set) or specific objects. Targets are hwloc *objects* of type NUMA node.

```
$ lstopo --memattrs
Memory attribute #0 name 'Capacity'
  NUMANode L#0 = 99786076160
  [...]
  NUMANode L#5 = 798863917056
Memory attribute #2 name 'Bandwidth'
  NUMANode L#0 = 131072 from Group0 L#0
  NUMANode L#1 = 131072 from Group0 L#1
  NUMANode L#2 =  78644 from Package L#0
  NUMANode L#3 = 131072 from Group0 L#2
  NUMANode L#4 = 131072 from Group0 L#3
  NUMANode L#5 =  78644 from Package L#1
Memory attribute #3 name 'Latency'
  NUMANode L#0 = 26 from Group0 L#0
  NUMANode L#1 = 26 from Group0 L#1
  NUMANode L#2 = 77 from Package L#0
  NUMANode L#3 = 26 from Group0 L#2
  NUMANode L#4 = 26 from Group0 L#3
  NUMANode L#5 = 77 from Package L#1
```

Fig. 5: Extracts from hwloc's `lstopo` reporting memory attributes on the Xeon platform depicted by Figure 2. NUMA nodes 2 and 5 are NVDIMMs while others are DRAM. Capacity is exposed in bytes (96GB for DRAM nodes, 768GB for NVDIMMs). Bandwidth is in MB/s (128GB/s to local DRAM, 76GB/s to local NVDIMMs). Latency is in nanoseconds (26 to local DRAM, 77 to local NVDIMMs). This platform only exposes performance attributes for accesses to local memory. `Group0` are *SubNUMA Clusters* (halves of processors).

These attributes also directly provide support for implementing the corresponding OpenMP 5.0 allocators and memory spaces such as `omp_high_bw_mem_space` [8].

TABLE I: Status of memory attributes in hwloc. Values may be discovered natively by hwloc when supported by the OS and hardware. Or external sources may feed them to hwloc.

Attributes	Native Discovery	External Sources
Capacity, Locality	Always supported	Unneeded
Bandwidth, Latency	On **most** platforms	Benchmarks
R/W Bandwidth, Latency	On **some** platforms	Benchmarks
Persistence, Endurance, Power	Under investigation	
Custom Metrics	N/A	User-specified

Table I shows many other attributes supported in hwloc. Separate bandwidth or latency for reads and writes may be dynamically added when supported by the hardware. Other attributes are being considered for addition when the relevant hardware information is available, for instance the power consumption or the endurance of non-volatile memory.

The API also lets users create attributes for metrics characterizing memories under specific circumstances. For instance a custom attribute may be added to hwloc to describe the performance of a STREAM Triad kernel by combining Read and Write bandwidths obtained from `get_value()`.

A. Discovering Attribute Values

In this section, we describe how to obtain the performance values to characterize memory targets. There are two main sources: hardware-provided information and benchmarking.

1) Hardware Information: hwloc uses the *Heterogeneous Memory Attributes Table* (HMAT) introduced in the revision 6.2 of ACPI specification[11] which should be available on future platforms to describe complex memory hierarchies.

Platform vendors can expose in this table theoretical latency and bandwidth between all the initiators (set of cores) and all the memory targets (NUMA nodes). For instance, on a platform with both HBM and DRAM, cores could access their local HBM at 500 GB/s with a 100 ns latency, or their local DRAM at 100 GB/s and 110 ns, while other CPU cores access this HBM at 300 GB/s and 150 ns. Latencies and bandwidths may optionally be specified independently for read and write accesses but current platforms rarely expose these yet.

We contributed to the exposure of these tables in the `sysfs` virtual file system starting in Linux 5.2. However, this is currently limited to the performance of local accesses (see Figure 5). Hence, it is for instance currently impossible to compare the local DRAM with the HBM of another processor.

2) Benchmarking: Until the ACPI HMAT table is properly implemented in all platforms, hwloc may use experimentally measured attribute values.

For example, the performance of DRAM on Intel *Cascade Lake* has been empirically measured at around 80 GB/s and 285 ns of latency while the performance of NVDIMM was 10 GB/s and 860 ns. [13]. Although, these values depend on the number of threads accessing memory, their access pattern, etc., they are sufficient to rank or compare the memories in our proposed API. Many benchmarks can be used for this purpose, for instance STREAM [12] for bandwidth under different access

[11] https://uefi.org/specifications

[12] http://www.cs.virginia.edu/stream/

patterns, Lmbench for latency[13], and Google Multichase for bandwidth and latency[14]. Moreover, separate values for reads and writes can be obtained and fed to hwloc when the ACPI tables do not provide them.

B. Heterogeneous Memory Allocator

Once hwloc obtained performance metrics of memory targets either from the hardware or from benchmarking, they may be queried by applications either as explicit values or as a ranking of best targets for some criteria.

However, modifying applications using this new low-level hwloc API implies significant work for end-users[15] even if it consists in only replacing `malloc` with an allocation call on the preferred memory target. Hence, we developed a higher-level memory allocator for simple use-cases in applications. We rather expect the hwloc API to be used in advanced runtime systems and/or by advanced users with complex allocation criteria[16], with the need to handle local and remote memory, memory migration, etc.

Our experimental memory allocator may be summarized with a single function `mem_alloc(..., attribute)` which allocates on the best local memory target for the specified attribute, for instance Bandwidth, Latency or Capacity (as listed in Table I). If the best target is full, the allocator can easily fallback to next ones according to the ranking for this attribute. If the attribute is not available on the platform, the allocator may also fallback to other similar attributes, for instance *Bandwidth* instead of *Read Bandwidth*. This first step is an easy and productive modification of application allocation calls towards more performance.

It looks similar to what has been proposed in the literature, for instance in memkind [3]. However, the key difference is that our attribute specifies what is important for the application (e.g. Bandwidth) without hardwiring it to a specific kind of memories (e.g. HBM). Our approach is more portable since it may for instance return DRAM on a platform with DRAM and NVDIMMs but no HBM.

We believe this approach is an easy and productive way to bring portable heterogeneous memory support to applications. Indeed, it only requires modifying each memory allocation by specifying the attribute that describes the requirements of the buffer. Moreover, the code modification step could still be avoided by intercepting and recognizing allocation calls (as proposed in *auto-hbwmalloc* [4]) to add sensitivity hints.

The actual difficulty rather lies in deciding the appropriate criteria for each buffer. We discuss this in the next section.

V. Finding the Best Allocation Criteria

The interface and allocator presented in the previous section offer an easy way for application to request specific kinds of memory. In this section we now look at how to decide what kind of memory an application should request for each buffer.

[13]http://www.bitmover.com/lmbench/

[14]https://github.com/google/multichase

[15]One has to first initialize the hwloc topology, etc.

[16]If the memory access pattern is 2 reads for 1 write, one may build its own target ranking by combining read/write bandwidths from the API.

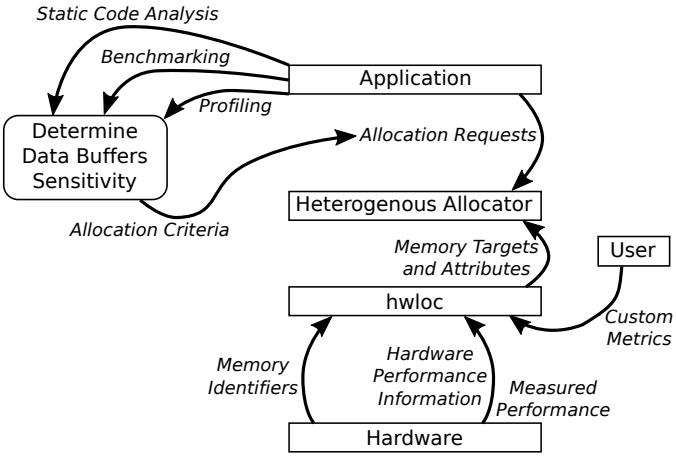


Fig. 6: Data buffer sensitivity may be determined by benchmarking, profiling and static code analysis. It is then used by modifying the application to use our heterogeneous allocator.

Our general framework is described on Figure 6. Analyzing the application behavior allows to determine the sensitivity of the important buffers. This sensitivity is exposed to the runtime as additional criteria in allocation requests. These criteria are then matched with memory attributes gathered by hwloc from the hardware or custom metrics specified by the user.

Some developers know their code well enough to guess which buffers are latency and/or bandwidth-bound thanks to years of optimizing cache affinity, prefetching, tiling, etc. However, there is a need for a more productive framework for non-experts in hardware architectures and code optimization. We survey below three main methods for determining this sensitivity. Benchmarking and Profiling are offline methods that require a first run for analysis, before determining the allocation criteria that can be used for future runs, assuming those runs have similar behavior (they may depend on the input data). Static code analysis does not suffer from this limitation.

A. Benchmarking

The simplest strategy for determining sensitivity is to bind the entire process to each kind of memory consecutively and compare the overall performance of each run. Unfortunately, this approach only works if all buffers have the same sensitivity or if there is a single performance-critical buffer. In the general case, one should rather compare the performance of all possible placements of every buffer in the application. This will often lead to a combinatorial explosion since N buffers lead to 2^N possible placements (if there are 2 kinds of memory), which might be reduced by identifying buffers that are obviously not performance critical.

This approach is still used extensively in the literature for showing the sensitivity of some benchmarks to certain memory attributes [2]. Kernels with regular access patterns and streamed accesses (as shown by the benchmark STREAM [14]) often show greater sensitivity to bandwidth. Those with a random access patterns and/or pointer indirection often show more sensitivity to latency.

B. Profiling

Profiling is a more complex strategy which performs an analysis of the execution using counters and/or instrumentation to identify in detail the bottlenecks, hot spots, etc. Many tools have been proposed to display such information graphically and link execution traces to the corresponding lines of code and memory allocations. Analyzing memory-related counters therefore helps identifying which buffers are limited by bandwidth or latency and when/where they were allocated.

This approach has already been used in hybrid systems with DRAM and NVDIMM for identifying sensitivity in terms of latency and energy [15]. An extension of the roofline model also added latency information and improved hints for NUMA memory placement [16]. We will present a use case integrated with our heterogeneous allocator in Section VI-B.

C. Static Code Analysis

Static code analysis consists in studying the source code, for instance during compilation. This step may be used for finding bugs or for providing the compiler and/or runtime with addition information about the program, for instance what is going to happen in the future with a data buffer.

Compilers have been trying to reduce latency of memory accesses for a long time by inserting software prefetching for specific applications and/or specific platforms [17], [18]. Scratchpad memory management may also be performed statically [19]. We believe that this kind of work should allow compilers to detect latency- or bandwidth-sensitive kernels and hence provide runtime systems with sensitivity hints. For instance, streamed/linear accesses to contiguous buffers can be detected and marked as bandwidth sensitive, without requiring the user to manually benchmark or profile its application.

This approach has been proposed for different languages to detect frequently used data and memory access patterns, and decide to allocate in HBM or DRAM [20]. Variables are annotated with priority values, and the compiler statically replaces malloc calls to allocate high priority buffers to HBM.

In our approach, the plan is rather to have the compiler insert annotations in the code to tell the runtime where to allocate each buffer. This may be performed by replacing allocation calls with our heterogeneous allocator. However, providing information to the runtime is generally a more flexible approach since it lets the runtime take decisions with more information about the overall execution. Static analysis is not considered in the rest of this paper because compilers are not ready to provide such hints yet.

VI. Use Case

Our method can be applied to provide memory allocations that respect the affinities and needs of computational tasks. We describe in this section how we successfully apply benchmarking and profiling to the Graph500 application[17] that uses irregular memory accesses. We used Graph500 version 3.0.0 over MPI. The performance is measured by an harmonic average of *Traversed Edges per Second* (TEPSe+8).

We carried out experiments on two servers. The Xeon server contains 2 Xeon *Cascade Lake* 6230 processors (20 cores each) with 192GB of DRAM and 768GB of NVDIMM each.[18] The KNL server contains a single Xeon Phi *Knights Landing* 7230 processor (64 cores) configured in SNC-4 Flat modes (processor split in 4 clusters, each with 24GB of DRAM and 4GB of HBM).[19] Since Graph500 requires 2^n tasks by default, we used 16 processes so that it fits inside a single Xeon CPU or KNL cluster. Only memories that are local to these CPU cores are used.

A. Benchmarking

TABLE II: Graph500 performance in Traversed Edges per Second (TEPSe+8).

Graph Size	DRAM	NVDIMM
2.15 GB	3.423	2.056
4.29 GB	3.459	2.067
8.59 GB	3.481	2.084
17.18 GB	3.343	2.107
34.36 GB	2.990	1.044

(a) Xeon: 16 MPI processes on a single processor using its local DRAM or NVDIMM.

Graph Size	HBM	DRAM
2.15 GB	0.418	0.415
4.29 GB	0.402	0.396

(b) KNL: 16 MPI processes on a *SubNUMA Cluster* using its local HBM or DRAM.

Graph500 performance depending on the memory placement of the entire process is presented in Table II. On the Xeon, the DRAM provides results between 1.5 and 3 times higher. This confirms that this application should specify either latency or bandwidth as a priority in our heterogeneous allocator since Xeon DRAM is faster for both metrics.

However, on KNL, DRAM results are very close to HBM. The latency of both memories is actually similar while the bandwidth is very different (90GB/s against 350 approximately). This result shows that the bandwidth criterion is in fact not suitable for this allocation (the gain is too weak to justify consuming the low HBM capacity).

These results confirm what was expected: The Graph500 application is rather limited by latency because it performs memory accesses with indirections during the graph traversal.

On an application limited by bandwidth, for example STREAM on Table III, bandwidth is obviously the criterion that should be passed to our allocator.

Thus, with our heterogeneous allocator and the bandwidth and latency criteria, we are able to allocate memory for two very different architectures. Combined with the capacity

[17]https://graph500.org/

[18]Contrary to Figure 2, *SubNUMA Cluster* is disabled. There is a single DRAM node per CPU (and a single NVDIMM per CPU) with 20 local cores.

[19]Contrary to Figure 1, memory-side cache is disabled. The entire MCDRAM is used as a separate 4GB NUMA node per cluster.

TABLE III: STREAM Triad throughput in GB/s depending on the optimized criteria. *Best Target* corresponds to the local NUMA node that the allocator found most appropriate for this criteria.

Optimized Criteria	Best Target	Total allocated memory for arrays 22.4GiB	89.4GiB	223.5GiB
Capacity	NVDIMM	31.59	10.49	9.46
Latency	DRAM	75.06	75.24	–

(a) Xeon with 20 threads on a single processor using its local DRAM (192GB) or NVDIMM (768GB) in 1-Level-Memory mode.

Optimized Criteria	Best Target	Total allocated memory for arrays 1.1GiB	3.4GiB	17.9GiB
Bandwidth	HBM	85.05	89.90	–
Latency	DRAM	29.17	29.17	29.16

(b) KNL with 16 threads on a *SubNUMA Cluster* using its local HBM (4GB) and DRAM (24GB).

criterion, this work allowed us to dynamically adapt the allocations according to the needs of applications and according to the actual available memory. This achievement would not be possible with existing interfaces because the application would only be able to request HBM explicitly instead of requesting a memory with good *Latency*. HBM allocations are not possible on the Xeon, while HBM allocations on KNL would consume MCDRAM without actually needing it for better performance. This shows that our work provides **same performance as manual tuning while remaining portable**, hence providing superior productivity.

B. Profiling

We now present profiling experiments on the Xeon platform to further dig into the details of data buffer sensitivity to bandwidth or latency. We used the Intel VTune Profiler whose *Memory Access* analysis tool can help determine hot memory objects in a program. However, we believe similar results could be obtained with many other profiling tools.

We presume that we do not know the latency-sensitivity of Graph500 main buffers and we profile the execution when allocating them on DRAM and on NVDIMMs separately. Again, only cores of a single processor are used together with their local memory. The old version 2.1.4 of Graph500 is used here for OpenMP support, because profiling a single process is easier (however profiling the more recent MPI version is also possible). As seen earlier on Table IIa, allocating the entire process on DRAM brings about two times better performance than on NVDIMMs.

First, the summary of the memory access analysis gives information about the overall application sensitivity to latency or bandwidth, Table IV presents the relevant information for our work. *DRAM Bound* and *Persistent Memory Bound* rely on Intel-specific counters indicating whether many cycles are spent accessing DRAM or NVDIMMs, hence showing CPU stalls and latency issues. *DRAM Bandwidth Bound* and *Persistent Memory Bound* rather show bandwidth issues as a percentage of elapsed time. For Graph500, VTune shows an indicator flag on the DRAM Bound parameter meaning that the application is latency sensitive, especially when running on NVDIMMs because this memory has a high latency.

Second, to identify the sensitive data buffers, the memory access analysis may provide details as shown in Figure 7a: which kind of memory is being used, the used bandwidth, and the list of buffers ordered by importance. Additionally, we can identify the corresponding source code touching these buffers. It is clear from this figure that the relevant buffer is allocated in `xmalloc` on line 31 (callstacks may also be displayed). *LLC Miss Count* is important here because it is the last and longest-latency in the memory hierarchy before main memory meaning that this latency cannot be avoided.

Thanks to this analysis we can now modify Graph500 to allocate this buffer with the latency attribute in our heterogeneous allocator, and get the same optimized performance on different platforms.

We performed the same analysis with STREAM on Table IV and Figure 7b. VTune shows the indicator flag on the parameter *DRAM Bandwidth Bound* or *PMem Bandwidth Bound* parameter depending on where memory is allocated. This shows that the overall application is rather sensitive to bandwidth, as expected[20]. The in-depth analysis of important buffers may then be performed as earlier.

C. Summary of the Use Case

Benchmarking and profiling are two methods that open the black box of memory attributes which applications can be sensitive to. Benchmarking may easily provide a general idea of the sensitivity of the application. Profiling requires more time for an in-depth analysis, but modern tools are able to give very useful information. Although, this information still requires human intervention, it allows to determine which buffers are actually important to performance and their sensitivity. This is a critical step towards applying the proper allocation criteria, either using our heterogeneous allocator, or in the runtime.

We believe that this approach brings higher productivity since this sensitivity information can be passed to allocators in a portable manner without having to hardwire information about existing memory kinds into the application code.

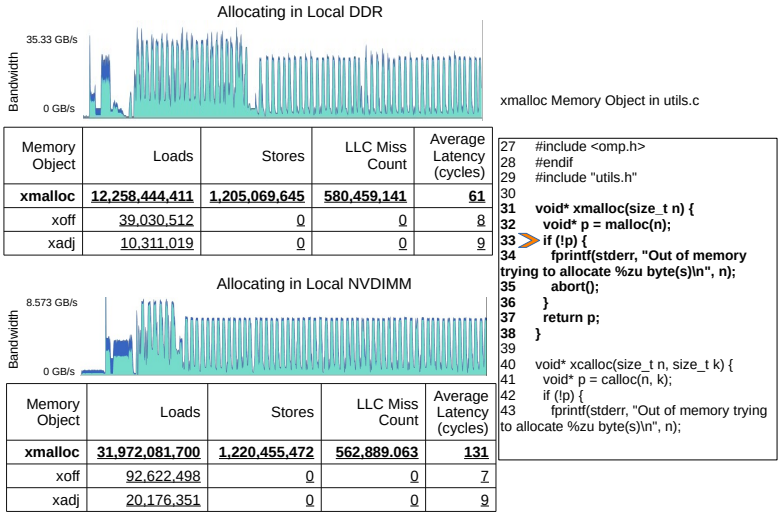
VII. Discussion

We showed in the previous sections that it is possible to combine application analysis with our interface to allocate each buffer on its appropriate target memory. However, the problem of target capacity needs to be discussed because it may conflict with buffer affinities. Indeed, one cannot allocate two 10GB buffers on a 16GB MCDRAM on KNL. Most implementations deal with this issue in a *First Come First Served* approach. Buffers that could not be allocated where requested may either be entirely allocated on slower memories (DRAM on KNL), or at least partially. Such hybrid allocations

[20]The tool even recommends to allocate to HBM if available.

TABLE IV: Extracts from the VTune Profiler execution summary for Graph500 and STREAM Triad using DRAM or NVDIMM.

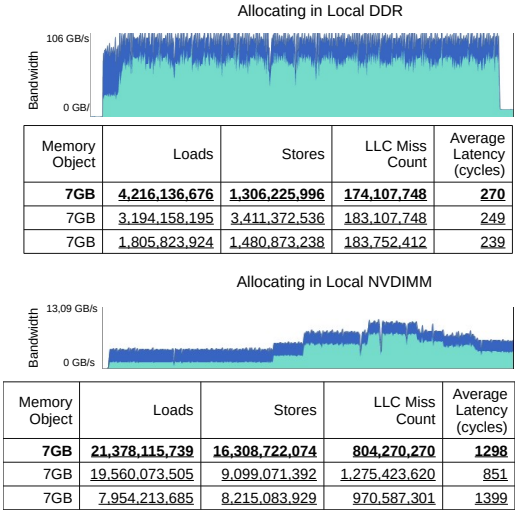
Application	Target	DRAM Bound in % of Clockticks	PMem Bound in % of Clockticks	DRAM Bandwidth Bound in % of Elapsed Time	PMem Bandwidth Bound in % of Elapsed Time
Graph500	DRAM	29.0%	0.0%	0.0%	0.0%
Graph500	NVDIMM	63.0%	60.9%	0.0%	0.0%
STREAM Triad	DRAM	63.3%	0.0%	80.4%	0.0%
STREAM Triad	NVDIMM	43.7%	17.0%	0.3%	2.1%



Memory Object	Loads	Stores	LLC Miss Count	Average Latency (cycles)
xmalloc	**12,258,444,411**	**1,205,069,645**	**580,459,141**	**61**
xoff	39,030,512	0	0	8
xadj	10,311,019	0	0	9

Memory Object	Loads	Stores	LLC Miss Count	Average Latency (cycles)
xmalloc	**31,972,081,700**	**1,220,455,472**	**562,889,063**	**131**
xoff	92,622,498	0	0	7
xadj	20,176,351	0	0	9

(a) Graph500 memory allocations and counters (on the left), and the source code where the the main buffer allocated (on the right).



Memory Object	Loads	Stores	LLC Miss Count	Average Latency (cycles)
7GB	**4,216,136,676**	**1,306,225,996**	**174,107,748**	**270**
7GB	3,194,158,195	3,411,372,536	183,107,748	249
7GB	1,805,823,924	1,480,873,238	183,752,412	239

Memory Object	Loads	Stores	LLC Miss Count	Average Latency (cycles)
7GB	**21,378,115,739**	**16,308,722,074**	**804,270,270**	**1298**
7GB	19,560,073,505	9,099,071,392	1,275,423,620	851
7GB	7,954,213,685	8,215,083,929	970,587,301	1399

(b) STREAM Triad memory objects and counters.

Fig. 7: Extracts of the Memory Access graphic interface in the Intel VTune Profiler. Execution with memory in DRAM (top) is compared to NVDIMM (bottom). The Read bandwidth is represented in turquoise, while the Write bandwidth is in blue (aggregated on top or Read).

across two kinds of memory are possible with the *Preferred* NUMA allocation policy of operating systems. Unfortunately, Linux does not support all combinations of *Allocate on node X if possible, or Y otherwise*[21]. Moreover, partial allocations will cause parts of a data buffer to be processed faster than others, leading to irregular application performance that may make task scheduling and profiling more difficult.

We believe that these capacity conflicts should be managed by using priorities: *Allocate buffer X on HBM first, and then buffer Y if possible*. In case of allocation failure, our heterogeneous allocator indeed falls back to other targets according to their ranking. Late allocations of performance sensitive buffers should thus be moved earlier when possible. Or the caller may query NUMA node capacity from hwloc to make sure HBM capacity will not be used earlier. Things might unfortunately get more difficult when application buffer sizes (or even the number of allocations) are not known in advance.

Such problems have already been studied intensively for hybrid storage, e.g. with SSD and HDD [21]. Proposed solutions include the analysis of block access patterns so that critical blocks are moved to the SSD while others remain on HDD. Read-intensiveness may also be used to prioritize some blocks for SSD. This leads us to think that analyzing the application memory access patterns is an interesting way to port this work to heterogeneous memory. Simple criteria such as file size have also been used to prioritize small files on SSD by assuming that those metadata are more important to the overall performance [22]. However, as discussed earlier, these criteria are hardly usable for memory buffers: some small buffers may be indirection blocks in graph (require low latency) and some large buffers may be streaming buffers (require high bandwidth) but they could also be non-important.

Memory migration could be a solution to avoid capacity issues when important buffers are not used during the same application phase. hwloc is already able to migrate memory buffers between memory targets that may be selected using our new API. However, this operation is quite expensive in operating systems [23]. Hence, it should likely be avoided unless the application behavior changes significantly between phases, either by using different buffers, or by using the same buffers with different access patterns.

[21]The preferred NUMA node must have a lower index than fallback nodes. This is for instance not possible on KNL if the preferred node is MCDRAM: KNL MCDRAM nodes always have higher index than DRAM (so that default allocations do not go to MCDRAM by mistake).

VIII. Conclusion and Future Work

As the memory subsystem of computing platforms becomes increasingly complex, there is a need for portable software solution for managing data buffers in HPC applications. The variety of emerging heterogeneous memory architectures where normal memory may be combined with high-bandwidth and/or non-volatile memory requires abstracted ways to compare these technologies before decide where to allocate. In this article, we presented an interface to manage the complexity by exposing hardware characteristics.

Our approach focuses on first identifying the existing memory kinds in the platform and then exposing their abilities through a number of convenient attributes, such as bandwidth, latency, and capacity. This identification and characterization step was missing in existing approaches and it hindered productivity and portability by requiring users to benchmark every new platform and guess their memory organization.

We implemented these ideas in hwloc, the de facto standard tool for managing hardware topology in HPC software. We then showed that implementing a heterogeneous memory allocator enables applications to easily specify their needs for each allocation. This work rather focuses on productivity than on performance: Performance will not be superior to manual tuning of each allocation, but we expect a much better portability. Indeed, there is no need to explicitly depend on a specific technology (e.g. HBM) or a specific numbering of NUMA nodes (e.g. 4 DRAM nodes first then 4 HBMs) in the code anymore.

Our work may also be used in existing frameworks for managing heterogeneous memory such as SICM [7], FLEXMALLOC [6] and Hexe [24] to provide easy discovery of the hardware instead of requiring manual description and/or benchmarking. As some of these projects already use hwloc (for discovering some hardware information and binding tasks or memory), extending them to also gather memory performance characteristics from hwloc looks straightforward.

The remaining difficulty consists in determining the sensitivity of application buffers. We presented a survey of existing techniques and showed their applicability on a use case. There is still some work to integrate these techniques into a general framework to either modify the allocation requests in the code or have a compiler that automatically insert hints to the runtime for these allocations. We are working with some OpenMP developers to leverage our work into runtimes, especially through OpenMP memory spaces and allocators [8].

One remaining step is the management of many available memories, local or not. On a server with four Xeon processors with NVDIMMs, it is possible to have 8 NUMA nodes DRAM (each processor can be configured in 2 *SubNUMA Clusters* as in the Figure 3) and 4 NVDIMMs (one per processor). Although, Linux currently does not expose performance attributes for accessing the memory of other CPUs, hwloc is still able to expose them thanks to benchmarking. However, exposing metrics for all these NUMA nodes may not be very useful for applications because only local nodes are used in general (1 DRAM and 1 NVDIMM). However, if the application is irregular and the local DRAM is full, is it better to allocate in the local NVDIMM or in another DRAM? They might be full in the future too.

Another open question lies in the precision of the attribute values. If DRAM and HBM have close latencies (as on KNL), should we still expose their exact values, knowing that they are difficult to measure and can vary with the load? Is it actually worth exposing separate values for Read and Write? As usual, a good trade-off must be found between the precision of the model and its simplicity for end users. In addition, we are considering exposing information about *Memory-side Cache*[22] since they may cause application-observed performance to be different from our attribute values[23]

Finally, we are looking at other kinds of memory that can be exposed by certain peripherals (NVIDIA GPUs on POWER architecture, NVMe disks, memory attached to network, etc.). hwloc is already able to expose them but we may want to define additional attributes for describing different constraints, for example in terms of coherency or availability.

Acknowledgments

This work was supported in part by the French National Research Agency (ANR) in the frame of the ANR-DFG H2M project (ANR-20-CE92-0022-01). Some experiments presented in this paper were carried out using the PlaFRIM experimental testbed, supported by Inria, CNRS (LaBRI and IMB), Université de Bordeaux, Bordeaux INP and Conseil Régional d'Aquitaine (see https://www.plafrim.fr/).

References

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

[2] I. Peng, R. Gioiosa, G. Kestor, P. Cicotti, E. Laure, and S. Markidis, "Exploring the Performance Benefit of Hybrid Memory System on HPC Environments," in *IPDPS Workshops*, Jun. 2017.

[3] C. Cantalupo, V. Venkatesan, J. R. Hammond, and S. Hammond, "User Extensible Heap Manager for Heterogeneous Memory Platforms and Mixed Memory Policies," 2015. [Online]. Available: http://memkind.github.io/memkind/memkind_arch_20150318.pdf

[4] H. Servat, A. Pena, G. Llort, E. Mercadal, H.-C. Hoppe, and J. Labarta, "Automating the Application Data Placement in Hybrid Memory Systems," in *Proceedings of Cluster Computing*, Hawaii, USA, Sep. 2017.

[5] 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 IPDPS*. Vancouver, BC, Canada: IEEE, May 2018.

[6] A. J. Peña and P. Balaji, "Toward the efficient use of multiple explicitly managed memory subsystems," in *Proceedings of Cluster Computing*, 2014, pp. 123–131.

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

[8] J. Sewall, S. J. Pennycook, A. Duran, C. Terboven, X. Tian, and R. Narayanaswamy, "Developments in memory management in OpenMP," *IJHPCN*, vol. 13, no. 1, pp. 70–85, 2019.

[22] KNL in *Cache* mode uses MCDRAM as a cache in front of the DRAM. Xeons in *2-Level-Memory* use the DRAM as a cache in front of NVDIMMs.

[23] The ACPI HMAT that exposes memory access performance does not specify whether those accesses are cached on the memory side.

[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," in *Proceedings of the Euromicro PDP*, Pisa, Italia, Feb. 2010, pp. 180–186.

[10] B. Goglin, "Exposing the Locality of Heterogeneous Memory Architectures to HPC Applications," in *International Symposium on Memory Systems*, ser. MEMSYS'16. Washington, DC: ACM, 2016.

[11] E. A. León, B. Goglin, and A. R. Proaño, "M&MMs: Navigating Complex Memory Spaces with hwloc," in *MEMSYS19*. Washington, DC: ACM, Oct. 2019, pp. 149–155.

[12] R. Murphy, "On the Effects of Memory Latency and Bandwidth on Supercomputer Application Performance," in *2007 IEEE 10th International Symposium on Workload Characterization*, 2007, pp. 35–43.

[13] A. van Renen, L. Vogel, V. Leis, T. Neumann, and A. Kemper, "Persistent memory i/o primitives," in *Proceedings of the 15th International Workshop on Data Management on New Hardware*, ser. DaMoN'19, New York, NY, USA, 2019.

[14] J. D. McCalpin, "Memory Bandwidth and Machine Balance in Current High Performance Computers," *IEEE TCCA Newsletter*, Sep. 1995.

[15] H. Liu, R. Liu, X. Liao, H. Jin, B. He, and Y. Zhang, "Object-Level Memory Allocation and Migration in Hybrid Memory Systems," *IEEE Transactions on Computers*, vol. 69, no. 9, pp. 1401–1413, 2020.

[16] O. G. Lorenzo, T. F. Pena, J. C. Cabaleiro, J. C. Pichel, and F. F. Rivera, "3DyRM: a dynamic roofline model including memory latency information," *The Journal of Supercomputing*, vol. 70, no. 2, 2014.

[17] 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 *Proceedings of ICS*, New York, NY, USA, 2018.

[18] S. Ainsworth and T. M. Jones, "Software Prefetching for Indirect Memory Accesses," in *Proceedings of CGO*. IEEE Press, 2017.

[19] P. R. Panda, A. Nicolau, and N. Dutt, *Memory Issues in Embedded Systems-on-Chip: Optimizations and Exploration*. USA: Kluwer Academic Publishers, 1998.

[20] D. Khaldi and B. Chapman, "Towards Automatic HBM Allocation Using LLVM: A Case Study with Knights Landing," in *2016 Third Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)*, 2016.

[21] S. Mittal and J. S. Vetter, "A Survey of Software Techniques for Using Non-Volatile Memories for Storage and Main Memory Systems," *IEEE TPDS*, vol. 27, no. 5, pp. 1537–1550, 2016.

[22] B. Welch and G. Noer, "Optimizing a hybrid SSD/HDD HPC storage system based on file size distributions," in *IEEE MSST*, 2013, pp. 1–12.

[23] F. Broquedis, O. Aumage, B. Goglin, S. Thibault, P.-A. Wacrenier, and R. Namyst, "Structuring the execution of OpenMP applications for multicore architectures," in *Proceedings of IPDPS*, Atlanta, Apr. 2010.

[24] L. Oden and P. Balaji, "Hexe: A Toolkit for Heterogeneous Memory Management," in *Proceedings of ICPADS*, Shenzhen, China, Dec. 2017.