A survey of software techniques to emulate heterogeneous memory systems in high-performance computing

Clément Foyer, Brice Goglin, Andrès Rubio Proaño

A Survey of Software Techniques to Emulate Heterogeneous Memory Systems in High-Performance Computing

Clément Foyer Brice Goglin Andrès Rubio Proaño
Inria, LaBRI, Univ. Bordeaux
Talence, France
clement.foyer@univ-reims.fr,brice.goglin@inria.fr,andres.rubioproano@riken.jp

***Abstract*—Heterogeneous memory will be involved in several upcoming platforms on the way to exascale. Combining technologies such as HBM, DRAM and/or NVDIMM allows to tackle the needs of different applications in terms of bandwidth, latency or capacity. And new memory interconnects such as CXL bring easy ways to attach these technologies to the processors.**

High-performance computing developers must prepare their runtimes and applications for these architectures, even before they are actually available. Hence, we survey software solutions for emulating them. First, we list many ways to modify the performance of platforms so that developers may test their code under different memory performance profiles. This is required to identify kernels and data buffers that are sensitive to memory performance.

Then, we present several techniques for exposing fake heterogeneous memory information to the software stack. This is useful for adapting runtimes and applications to heterogeneous memory so that different kinds of memory are detected at runtime and so that buffers are allocated in the appropriate one.

***Index Terms*—Heterogeneous memory, Emulation, NUMA, Pirate, Throttling, Cache, Virtual Machine, ACPI, Runtime**

I. Introduction

The increasing computing power of processors brings significant changes in the memory subsystems. After adding several levels of caches between the (fast) CPU and the (slow) main memory, hardware architects are now looking into heterogeneous memory technologies to match the needs of applications. Indeed a single technology such as DRAM, currently used as main memory in most high-performance computing systems (HPC), provides a reasonably low latency but a limited capacity and a low bandwidth. Some systems such as the Fugaku supercomputer, formerly ranked #1 in Top500[1], chose to rather use a low capacity (32 GB) high-bandwidth memory (HBM) as main memory. However, some applications such as Graph500 may rather want low latency, while others prefer large capacities such as those provided by emerging non-volatile memory DIMMs (NVDIMMs).

Combining two of these technologies in a platform is an easy way to answer the needs of different applications. For instance Xeon processors from Intel support DRAM with large capacity NVDIMMs, and the Sapphire Rapids generation may even comes with embedded HBM. Several ARM vendors also plan to combine HBM and DRAM in their platforms. Moreover most upcoming CPUs will also support CXL, the new memory interconnect allowing to connect and share different kinds of memory and devices in a cache coherent manner. This hardware trend, however, brings significant changes to the way the software manages memory. Most HPC applications and runtimes are already able to manage non-uniform memory access (NUMA) by allocating their buffers on the memory near the CPUs that accesses them. Heterogeneous memory further complexifies this approach since buffers should be allocated in the right memory technologies. For instance, buffers heavily accessed in a streamed-manner usually benefit from HBM, while graph-based data structures rather prefer low latency. And the low capacity of HBM (and DRAM) may cause very large allocations to only fit in NVDIMMs.

Developers must prepare applications and runtimes to work with upcoming heterogeneous memory systems in a portable manner, even if those platforms are not available or accessible yet. Hence, we survey in this article many ways to emulate these hardware systems to ease the software development. We consider two kinds of emulation: First, *Performance Emulation* consists in exposing heterogeneous memory performance (bandwidth, latency, capacity) even if the underlying technologies is homogeneous. It helps the developer detect which part of its code and which data sets are sensitive to different memory performance. Second, *Environment Emulation* consists in exposing heterogeneous memory information from the hardware and operating system even if the performance is homogeneous. This helps the developer adapt its code to identify, locate and use the existing memory technologies in the system. We focus on solutions available on Linux since it is used by the vast majority of platforms in HPC.

The remainder of this article is organized as follows. First, we present the current and future heterogeneous memory systems in Section II as well as the ways to expose them in the firmware and to manage them in software. Then, many solutions to implement *Performance Emulation* for different memory characteristics are introduced in Section III. Finally, several strategies for implementing *Environment Emulation* are

[1]https://top500.org/system/179807/

Preprint submitted to Parallel Computing,
accepted version available at http://dx.doi.org/10.1016/j.parco.2023.103023

presented in Section IV.

II. Hardware Landscape

We describe in this section the current and upcoming heterogeneous memory platforms in high-performance computing. Then, we explain how this hardware heterogeneity impacts application performance and how it is managed in software.

A. Heterogeneous Memory Platforms in HPC

The first widely available heterogeneous memory system in HPC was the Intel *Knights Landing* Xeon Phi (KNL) launched in 2016. This many-core processor came with 16 GB of embedded Multi-Channel DRAM (MCDRAM, similar to HBM) offering a bandwidth of 400 GB/s, while the larger off-package DRAM only reached about 100 GB/s [1]. The MCDRAM could be used as a separate NUMA domain requiring the allocator to decide between low-capacity high-bandwidth and large-capacity low-bandwidth. The MCDRAM could also be configured as a hardware-managed cache in front of the DRAM. This *Cache* mode is a trade-off between performance and simplicity since the application does not need to be modified to explicitly allocate each buffer in the appropriate kind of memory [2].

KNL is now discontinued and no platforms with a similar memory subsystem was available in HPC until recently. However, several CPU vendors announced the return of integrated high-bandwidth memory in future (exascale) platforms. First, the Intel Xeon *Sapphire Rapids* optionally embeds 64 GB of HBM2e [3]. Combined with the usual off-chip DRAM, its memory subsystem is very similar to KNL. Then, several exascale initiatives announced their plan to use ARM Neoverse V1 CPUs with both HBM and DRAM [4]. SiPearl's *Rhea* processor will be used in the *European Processor Initiative*. In South Korea, ETRI (*Electronics and Telecommunications Research Institute*) will develop the *K-AB21* CPU. In India, the *Centre for Development of Advanced Computing* (C-DAC) is designing the *Aum* processor. All these developments employ HBM in one form or the other.

The emergence of non-volatile memory (NVDIMMs) also brought a different kind of heterogeneity. Indeed, this memory technology can be used as persistent storage but also as large-capacity main memory. Instead of being the large-capacity slow memory in KNL, DRAM is considered here the low-capacity fast memory. This technology is available in Intel Xeon platforms since the Cascade Lake generation as *Optane DataCenter Persistent Memory Modules*. However, Intel is phasing out this product which was not widely used in HPC yet despite showing interesting benefits for performance and capacity on some applications [5].

Another way to bring heterogeneity is to combine the main host memory with I/O device memory. Modern peripherals, for instance high-performance network interfaces, often expose part of their memory to the CPU. However, this is not designed to extend the main memory but rather provide more efficient or flexible transfers since the CPU may directly access this I/O memory. GPGPUs are a special case where the device memory is used for storing application data during GPU computation. Vendors are designing hardware protocols to manage memory in a cache-coherent manner so that both CPUs and GPUs may compute using data stored either in host memory or GPU memory. POWER9 platforms implement this approach over the CAPI and NVLink protocols interconnecting the host and NVIDIA GPUs: each GPU memory is exposed as a separate NUMA node in the host. Not only it creates a NUMA platform where each NUMA node is only local to a single CPU or a single GPU, but also a heterogeneous memory system since GPU memory is HBM with very different performance from the host DRAM, especially when accessed remotely across the NVLink bus [6]. NVIDIA is continuing this road by combining an ARM CPU and a Hopper GPU in the so-called *Grace Hopper* single package, where the CPU LPDDR5 memory and the GPU HBM3 memory are shared in a cache-coherent way [7]. However, for now, NVIDIA recommends allocating GPU memory with CUDA, hence making the software management highly non-portable to other heterogeneous memory platforms.

All these platforms expose significant memory performance discrepancies: MCDRAM and HBM have much higher bandwidth than DRAM, but their latency is often higher, especially when accessed through additional buses (e.g., NVLink). NVM has significantly higher latency and lower bandwidth, but offers larger capacity. Also capacity usually decreases when bandwidth increases, and they have no correlation with latency. TABLE I summarizes the approximate characteristics of these technologies.

However the main reason for heterogeneity in future memory subsystems is that vendors are developing new interconnects with generic support for different kinds of memory. The POWER10 processor will use the *Open Memory Interface* (OMI) which allows connecting DRAM, HBM and NVDIMMs [8]. Gen-Z, another memory interconnect, is now part of the *Computer Express Link* (CXL) initiative [9]. CXL is being adopted by many vendors, and even already supported by latest server processors from Intel (*Sapphire Rapids*) and AMD (*Genoa*). This will bring standard support for NVDIMMs to many processor models [10]. CXL CPUs enable heterogeneity by combining DRAM in usual DDR slots with CXL *Memory Expander* devices connected to PCIe slots. The latency overhead for accessing memory across a direct CXL link is announced in the range of 100-200 nanoseconds. CXL bandwidth is limited by PCIe Gen5 (4 GB/s per lane) with up-to 16 lanes per device (64 GB/s total), but multiple CXL devices may be used simultaneously. However the variety of technologies and the ability to use cables of different length or even switches will bring much more diverse heterogeneity since the interconnect topology will impact the end-to-end bandwidth and latency.

B. How Heterogeneous Memory is exposed by the Firmware

The Advanced Configuration and Power Interface (ACPI) specifies how the hardware information is exposed to the operating system [11]. It includes listing many hardware

TABLE I: Bandwidths, latencies and capacities of different memories when accessed from all cores of a single modern CPU socket. NVIDIA GPU refers to a POWER9 CPU accessing the GPU memory across CAPI and NVLink. CXL DRAM refers to a DRAM memory expander directed connected to the CPU (no CXL switch). All these values are very approximate since they depend on the actual CPU model, number of memory channels and DIMMs, frequencies, etc.

Technology	Bandwidth (GB/s)	Latency (ns)	Capacity (GB)
DRAM (DDR5)	200	100	100s
1 NUMA hop away	*150*	*200*	*100s*
KNL MCDRAM	400	120	16
HBM2	800	120	64
NVDIMM	50	300	1000
NVIDIA GPU	50	200	50
CXL DRAM	50	250	1000

resources such as CPUs and memories so that the OS may find out what it may actually configure and use. Many ACPI tables exist and at least four of them are now involved in the description of the memory subsystem, as shown in TABLE II.

TABLE II: Examples of information provided by ACPI tables on heterogeneous memory systems.

Name	Example of Contents
SRAT	NUMA node #3 contains physical memory from 0x20000 to 0x2ffff and it is local to CPU cores #10–19.
SLIT	CPU cores from NUMA node #1 read from NUMA node #2 memory with relative latency 21.
HMAT	CPU cores from NUMA node #1 read from NUMA node #2 with 80 ns latency and 100 GB/s bandwidth.
NFIT	Memory from 0x400000 to 0x5fffff is non-volatile and interleaved across 4 NVDIMMs.

First, the SRAT table (*System Resource Affinity Table*) says which CPU cores and/or memory ranges belong to each proximity domain, basically building the concept of NUMA node.

Then, the *System Locality Information Table* (SLIT) exposes a relative latency matrix for accesses from cores in one domain to memory in another domain. This allows to represent the topology of the NUMA interconnect but this table has several drawbacks. First, it was rarely correctly implemented by CPU vendors because it was not significantly used by operating systems.[2] Most vendors would mark nodes as remote without specifying the actual distance, making the entire NUMA topology flat. Second, CPU-less memory nodes cannot be properly represented since they cannot perform memory accesses as reported by the table. Third, heterogeneous memory performance cannot only be exposed through latency. For instance HBM currently has higher bandwidth but also slightly higher latency than standard DRAM, which means SLIT would expose HBM as slower. KNL platforms had to hardwire unusual values in the table so that default memory allocations do not use MCDRAM by default, while still allowing applications to detect which NUMA node is MCDRAM.

These limitations of SLIT led to the addition of the HMAT table (*Heterogeneous Memory Attribute Table*). HMAT clarifies the difference between CPU proximity domains (called *Initiators*) and memory proximity domains (*Target*). It describes memory performance in terms of latency and bandwidth, possibly separating read and write accesses, and may also expose information about memory-side caches. HMAT is able to properly describe the memory subsystem of KNL platforms, but it is only implemented in recent platforms such as those based on Intel *Ice Lake* processors. It provides a way to tell applications which memory targets are fast or slow without explicitly saying whether they are HBM, DRAM or NVDIMMs.

Finally, ACPI also provides a *NVDIMM Firmware Interface Table* (NFIT) that describes specific properties of non-volatile memory DIMMs.

These static ACPI tables describe devices whose possible existence, location and performance is known by the firmware a priori.[3] Some additional tables may also exist for optional devices with different possible interconnection, performance, etc. For instance, CXL exposes the *Coherent Device Attribute Table* (CDAT) which may provide HMAT-like information for CXL memory devices.

Additionally, the attribute of each physical memory ranges are also listed in either the legacy BIOS E820 table or in the modern UEFI memory map. The latter is notably used to specify *Soft Reserved* ranges of memory that should not be used by default memory allocations but rather reserved to specific drivers or applications. This is the modern recommended way to expose performance-differentiated memories such as HBM. Combined with HMAT, these attributes may now tell applications that a memory node is faster and that it is actually HBM.

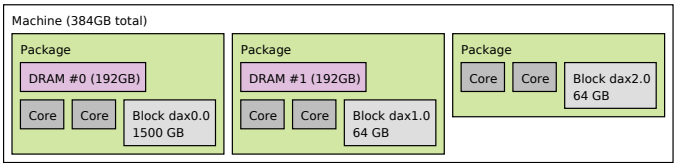
Thanks to all these firmware tables, the operating system only exposes standard DRAM by default. Non-volatile memory (NVDIMMs), soft-reserved memory (HBM) and CXL memory may be hidden (offline) or available through specific interfaces.[4] On Linux, DAX (*Direct Access*) special files may be used to expose them as depicted on Fig. 1a. Applications have to memory-map those files to gain access to the target memory. However, DAX devices may also be converted into *system-ram* mode. Once reconfigured, each target memory appears as an additional dedicated NUMA node on the side of the standard DRAM NUMA nodes (Fig. 1b).

Similarly, if the system was initially configured to keep HBM and CXL memory offline instead of as DAX devices, the administrator may online them to make them available as additional NUMA nodes (identical to Fig. 1b).

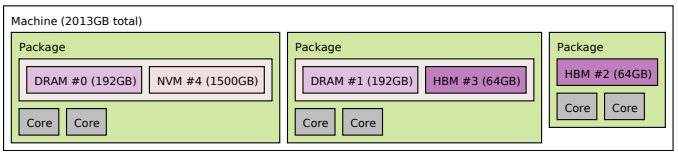
[2]Large NUMA platforms such as the SGI Altix UV used in Fig. 2 are a rare example of very well defined SLIT table.

[3]Devices that are soldered on the motherboard or that may only be connected directly in specific slots, e.g. DIMM slots.

[4]The default behavior depends on the Linux kernel version and configuration.



(a) By default, HBMs and NVDIMMs may be reserved as DAX special files (`dax0.0` for NVDIMMs, `dax1.0` and `dax2.0` for HBMs). Only DRAM memory is exposed as actually usable NUMA nodes (#0 and #1). Alternatively, the system could have been configured to expose HBMs as offline memory (unusable) instead of DAX devices.



(b) After `daxctl reconfigure --mode=system-ram`, HBMs and NVDIMMs are converted from DAX files to additional NUMA nodes (HBMs #2 and #3, and NVM #4 here).

Fig. 1: `lstopo` graphical outputs of a fictitious heterogeneous platform with 5 NUMA nodes (2 DRAM, 2 HBM, 1 NVM) depending on their configuration on Linux.

C. Impact of Heterogeneous Memory on the Software Stack

Current heterogeneous memory platforms have well known performance characteristics. Capacity usually goes up from HBM to DRAM and then to NVDIMMs. On KNL, HBM bandwidth is about 4× higher than DRAM, while latency is similar [2]. On Xeon with Optane NVDIMMs, latency is 2–4× higher for NVDIMMs against DRAM, while bandwidth is 2–10× lower. Bandwidth-bound compute kernels should place their buffers in HBM if possible. Latency-bound kernels should avoid NVDIMMs, and may want to avoid wasting HBM capacity since DRAM latency is similar. However, it is hard to generalize these strategies to future platforms given the variety of configuration exposed in Section II-A. Hence, there is a need for HPC runtimes to describe the hardware configuration in a portable manner instead of having every application manually hardwire which memory target to use.

With the release of KNL, the `memkind` library was proposed to discriminate NUMA nodes between DRAM and HBM, and ease memory allocations in specific technologies [12]. It was later extended to support NVDIMMs. The OpenMP standard now leverages this idea with specific allocators for bandwidth, latency and capacity [13]. A more abstracted approach consists in exposing performance characteristics of memory nodes so that applications may match them with their needs: a bandwidth-bound application prefers DRAM over NVDIMMs even if the machine does not have HBM [14].

Once the hardware memory subsystem is abstracted, developers have to identify which application's data buffer is actually sensitive to bandwidth or latency. Servat [15] and Narayan [16] (MOCA) use a post-mortem analysis of memory accesses based on hardware sampling. SICM [17] annotates the program before profiling. These approaches provide ways to determine which data set to allocate in which kind of memory.

These tools indeed help application developers manage heterogeneous memory in their code, but it is currently possible to test them on very few different platforms as explained in Section II-A, namely KNL and Xeon with NVDIMMs. Hence, there is a need for ways to emulate different platforms with different combinations of memory technologies and different performance characteristics. First, developers must test their application with different hardware bandwidths and latencies to identify which kernels and data buffers are sensitive to memory performance. We present techniques for *Performance Emulation* in Section III. Second, they need to adapt their code to allocate on the appropriate memory target, which means the underlying hardware, the operating system and the runtime must expose information about heterogeneous memory. We describe techniques for *Environment Emulation* in Section IV.

III. Performance Emulation

Adapting applications to heterogeneous memory requires to identify which data buffers are accessed in latency-sensitive or bandwidth-sensitive patterns. Static analysis, profiling and benchmarking are the main strategies for detecting such sensitivity [14].

Benchmarking consists in comparing the performance of the application depending on where each buffer was allocated during different executions. This solution is the easiest but it requires to actually run the code on platforms with heterogeneous memory performance. Moreover, different heterogeneities are needed to better understand whether a buffer is significantly sensitive to bandwidth or latency. Hence, there is a need for techniques to make hardware memory performance more or less heterogeneous, for bandwidth and/or latency.

We detail in the next sections several software solutions for exposing different memory performance and heterogeneous memory on Linux. We compare in terms of flexibility (ability to emulate different bandwidths and/or latencies), of granularity (performance of all memory accesses are impacted, or only some of them), and of technical difficulties for deploying them in practice (technical complexity, privileges required, runtime overhead, and availability).

A. Simulators

Many hardware simulators are available to execute applications on some customizable virtual platforms. Cycle-accuracy for executing instructions inside the CPU cores and accessing memory banks through caches enables predicting the performance of applications. This is actually one of the very few techniques that allow simulating hardware faster than the actual hardware running the simulator: CPU cycles can be slowed down so that memory accesses appear fast. Simulator are also very flexible since they allow to build highly customizable platforms with different numbers or kinds of CPU cores, memory, caches, etc. with tunable latencies and throughputs.

However, the runtime overhead is usually very high. The SESC simulator [18] for instance runs hundreds of times slower than a native platform [19]. The memory subsystem is responsible for a significant part of the this overhead [20]. Lighter technologies allow to only simulate part of the hardware. SimpleScalar or QEMU may simulate user-space code while system calls are emulated without cycle accuracy. Simulators also allow to simplify the hardware for instance by removing memory caches. However, these lighter strategies cannot predict application performance accurately anymore.

Simulating heterogeneous memory platforms to find out the sensitivity of applications obviously requires to simulate CPU instructions and memory accesses precisely. Hence, reducing the accuracy of the hardware modeling is not an option, and the simulation overhead will remain inconvenient. Thus, simulators remain a limited solution for studying the performance of large HPC parallel applications under different heterogeneous memory conditions.

B. Compiler Techniques

Instead of simulating heterogeneous memory hardware, another approach consists in modifying software memory accesses at compilation or execution time. Making access apparently slower or faster is indeed a way to test the application behavior if the actual memory performance were different.

Slowing down a specific access could for instance consists in adding useless instructions before it. Accelerating accesses can be performed by removing them, by adding prefetching, or by retargeting them to a constant memory address so that they hit the cache. This has been used in compilers to find lower-bounds or bottlenecks of executions [21], but it may also be performed by patching the binary to replace instructions [22].

The advantage of this approach is that it may focus on very specific parts of the application or even specific memory accesses. For instance, affecting phases that are memory-latency-bound allows to simulate a memory with higher latency without changing its bandwidth. One drawback is that the performance change cannot be precisely customized since the cost of an access is replaced with the cost(s) of specific instructions or accesses. Moreover replacing loads and stores can modify the behavior of the reminder of the application if it depends on the input values.

This strategy has also been used at coarse granularity to slow down entire phases without targeting specific accesses. For instance the Quartz simulator [23] was used to inject delays to simulate accesses to slow non-volatile memory [24]. The approach is more simple but it prevents from slowing down specific memory accesses, or accesses to specific NUMA nodes in the platform.

C. NUMA Distance

The NUMA distance may be used to simulate heterogeneous memory since accessing local memory is usually faster than memory further away in the machine. In today's multi-socketed architecture, there are actually often two levels of NUMA hierarchy. First is the inter-socket level where cores of a CPU socket have slower access to memory of other CPUs. Then the intra-socket level where the local memory is actually split between 2 or 4 sets of cores (Intel *SubNUMA Clusters*, AMD *Nodes per Socket*, etc.).

TABLE III: Memory latency and bandwidth measured with Intel Memory Latency Checker on a server with two AMD Zen2 EPYC 7702 CPUs. Each CPU has 64 cores (2 GHz) split into 4 NUMA nodes. Initiator is bound to cores of NUMA node #0.

	First CPU				Second CPU			
Target NUMA	0	1	2	3	4	5	6	7
Latency (ns)	100	116	125	129	223	227	220	222
Bandwidth (GB/s)	40	40	39	38	22	22	22	21

TABLE III presents the bandwidth and latencies on a platform with intra- and inter-socket NUMA distances. Bandwidth decreases by about 1.8× when accessing inter-socket memory, while latency increases by about 2×. This phenomenon has been very commonly used to emulate a heterogeneous memory system: the local memory is the fast target while the remote socket memory is the slow target [23], [25]. The intra-socket NUMA topology shows much smaller performance differences, up to 30 % for latency and 5 % for bandwidth. Hence, this strategy may hardly simulate more complex heterogeneous memory systems with very different target performance (for instance 3 kinds of local memory, or 2 sockets with 2 kinds each).

This strategy may, however, be extended to larger platforms with many NUMA nodes with very different distances. SGI Altix systems have been used in HPC to assemble tens or even hundreds of CPUs into a single shared-memory machine with a complex NUMA topology. Two CPUs are usually assembled in a blade, and blades are connected through a ring or hypercube network (*SGI NUMALink*). Hence, a wide variety of NUMA distances is available, possibly leading to a wide variety of latencies and bandwidths between CPUs and remote memory banks.

Fig. 2 shows how bandwidth and latency evolve when increasing the distance from the initiator CPU and the target memory. We have five different inter-socket performance profiles, which correspond to the theoretical latency of the NUMA interconnect. When accessing data located in NUMA nodes attached to different sockets, the performance heavily decreases.

Hence, this platform could be used to emulate a heterogeneous memory by considering that the local memory is HBM (27 GB/s and 100 ns), memory attached to the other CPU in the blade is DRAM (10 GB/s and 450 ns), and memory far-away in the platform is slow NVDIMM (7 GB/s and 800 ns). As long as the platform is not shared with other applications that could cause congestion of the NUMA interconnect by using other NUMA nodes, this heterogeneous performance should be reproducible.

Unfortunately, although this setup is interesting for testing applications, it hardly matches current heterogeneous memory systems: HBM and DRAM bandwidths are about 10× higher,

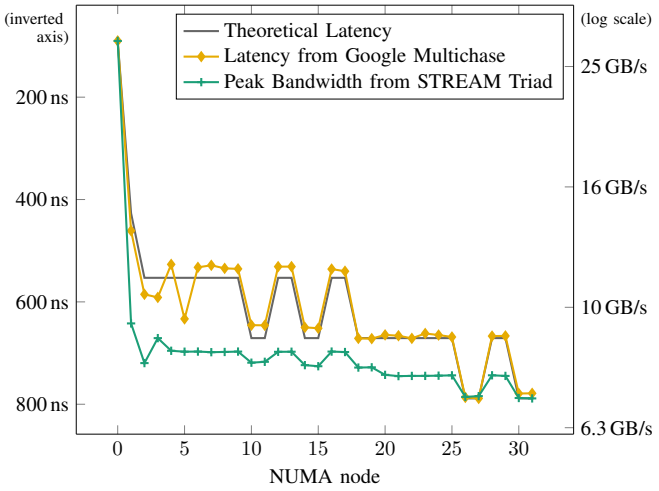


Fig. 2: Latency and bandwidth between NUMA nodes of a SGI Altix UV2000 platform composed of 128× Intel Xeon E5-4650L (2.6 GHz, 8 cores). Initiator is bound to cores of NUMA node #0. The theoretical latency is read from the ACPI SLIT table.

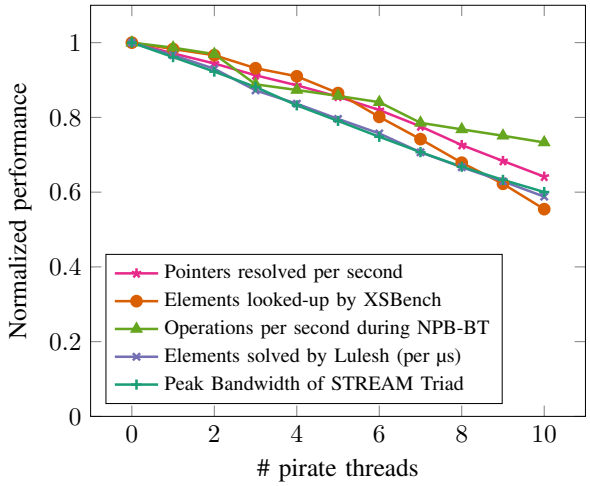


Fig. 3: Effects of a bandwidth pirate (STREAM Triad) using $0 \leq N \leq 10$ cores on applications (Pointer-chasing, XSBench, NPB-BT, Lulesh and STREAM Triad) running on 10 other cores of the same Xeon 6230 processor and accessing its local NUMA node. The remaining $10 - N$ cores are idle.

while DRAM latency should be slightly lower than HBM. Moreover, a significant limitation of this strategy is that latency and bandwidth are always both affected by the NUMA distance penalty: it is not possible to emulate a system with one memory target preferred for bandwidth (i.e. HBM) and another one for latency (i.e. DRAM). Additionally, one must remember that these large NUMA platforms are increasingly rare.[5]

D. Pirate Applications

Pirating consists in using a separate program that consumes resources to restrict what is actually available to the target application. This has been proposed as a way to steal memory bandwidth or cache capacity [26], hence emulating a slower memory subsystem.

Fig. 3 shows how running a loop of STREAM kernels (bandwidth-bound) on some cores indeed reduces significantly the performance of applications running on other cores of the same CPU package. The reduction follows the number of cores used by the pirate; hence, it allows the user to tune the available bandwidth for the target application. In this case, the application and pirate run on cores that share a L3 cache and NUMA node (as well as the memory controller). It is not clear which of these shared resources causes the application slowdown. This actually depends on the hardware platform. For instance, an AMD processor would allow to share the NUMA node without sharing a cache.[6] A pirate may also apply contention on the NUMA interconnect. For instance an application running of the first CPU and accessing memory on another NUMA node may be slowed down by a pirate accessing memory in the reverse direction.

While pirating is easy to deploy, it has several drawbacks:

- Fig. 3 shows that latency-bound applications (Pointer-Chasing and XSBench) are also impacted, but in slightly different ways (the impact is more significant for large numbers of pirate threads). This may prevent from building custom memory heterogeneity since bandwidth and latency cannot be impacted independently.
- The impact of the pirate depends on the application behavior. It cannot be easily programmed to always consume a specific amount of bandwidth as the hardware memory bandwidth is shared by all cores performing memory accesses. A pirate running a STREAM kernel on half the cores may steal half the bandwidth of a bandwidth-bound application using the other half of the cores. However, it will steal much more bandwidth if the application does not access memory intensively.
- The pirate must reserve a significant share of the cores to produce a visible impact. The share of stolen bandwidth is usually proportional to the share of reserved cores. For instance, only about 20 % of the memory bandwidth may be stolen by 4 pirating cores on our 20-core CPU.

Finally, one may wonder if reserving half the CPU cores to steal half of memory bandwidth is actually useful to studying heterogeneous memory. This strategy basically splits the entire machine in half, hence exposing a twice-smaller platform with same relative CPU/memory performance. However, one has to remember that reserved cores may run a pirate or just remain idle. Hence the bandwidth available to the application may vary between the entire platform bandwidth and its half, as shown on Fig. 3. Changing the available bandwidth while the number of application threads is constant allows to study the

[5]SGI was acquired by HPE in 2017 and current platforms (*HPE Superdome Flex*) are limited to 32 CPU sockets.

[6]AMD EPYC3 *Milan* processors have L3 caches shared by 8 cores while each NUMA node is local to at least 16 cores.

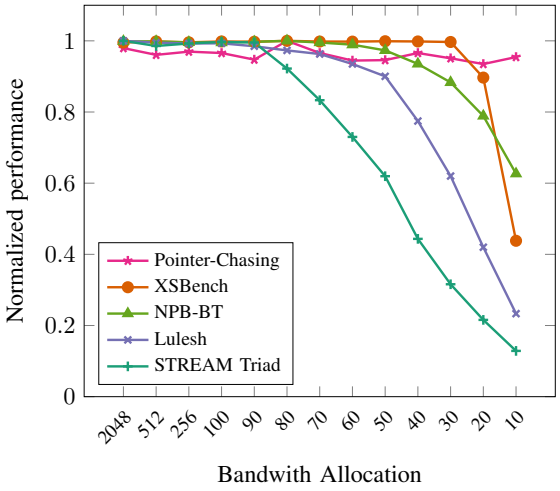


Fig. 4: Effects of bandwidth throttling on applications (Pointer-chasing, XSBench, NPB-BT, Lulesh and STREAM Triad) running on 8 threads on a 32-core AMD EPYC 7502 (2.5 GHz) and accessing its local NUMA nodes. *Bandwidth allocation* is a unit-less value between 1 and 2048 given to the processor to set the bandwidth available to a set of processes or cores.

application under different memory performance profiles. This is useful for determining if an application or kernel is sensitive to memory bandwidth.

E. Bandwidth Throttling

POWER processor have had the ability to throttle memory for a long time. Throttling DRAM allows to decrease power consumption while keeping performance unchanged if the application bandwidth requirements match the throttling [27], [28]. Other architectures such as Intel processors allow to change the memory frequency in the BIOS. Combined with CPU frequency tuning, it changes the performance ratio between CPU and memory [29]. Some more recent processors even allow to configure throttling per channel or DIMM [27], [30]. However, in practice, applications may only allocate in specific NUMA nodes, not in specific DIMMs. Hence, these features allow to configure different NUMA nodes with different performance and to allocate application buffers in the appropriate one. For instance, the Quartz simulator uses DRAM throttling to emulate non-volatile memory's (low) bandwidth [23].

The drawback of these DRAM throttling techniques is that they usually focus on frequency rather than bandwidth, hence latency may significantly increase when bandwidth decreases. Additionally, they require at least root access, or even a reboot to change the BIOS configuration. Given that monitoring and partitioning access to shared resource is important on virtualized hosts, vendors recently improved and facilitated throttling configuration. The Intel *Resource Director Technology*, AMD *Quality of Service* extension, and ARM *Memory Partitioning and Monitoring* are hardware features exposed to users through the *Resource Control* subsystem of recent Linux kernels[7][8]. They allow to explicitly partition cache and memory bandwidth between sets of processes or cores [31].

Fig. 4 shows how this hardware bandwidth throttling is able to limit memory performance. Bandwidth-bound applications such as STREAM or Lulesh are significantly affected, while latency-bound programs such as pointer-chasing or XSBench are not or very slightly affected. Contrary to the pirate technique, bandwidth throttling can affect bandwidth without latency, which is important for testing applications under different memory performance profiles. Moreover, its impact is well controlled and it does not depend on the memory access pattern of the application. Also, hardware throttling does not require to waste cores for running pirate threads. This is the reason why we used bandwidth throttling to detect application's sensitivity to bandwidth in order to adapt applications to upcoming heterogeneous platforms [32].

Hardware bandwidth throttling is only available on recent processors (Intel Skylake, AMD Zen2, etc). Compared to former DRAM throttling techniques, it has the important advantage of only requiring root access when creating the partition where applications will run. The actual (re)configuration can be applied by normal users without rebooting. It may even be modified at runtime between phases of a single application. However, the throttling is applied to all memory accesses, it cannot distinguish different target NUMA nodes. Hence, it does not build a heterogeneous platform but allows to test application under different memory performance profiles.

F. Cache Disabling, Partitioning or Locking

A last set of techniques for modifying memory access performance consists in modifying the behavior of the caches.

First, memory regions may be marked as uncachable. Disabling caches significantly increases latency and decreases bandwidth, hence exposing a much slower memory (at least for applications that usually benefit from caches). On former x86 architectures, ranges of physical memory could be marked uncachable by administrators through the *Memory Type Range Register*[9]. This is now deprecated in favor of the *Page Attribute Table* which applies to pages of virtual memory. PAT is supported on all CPUs nowadays, but it requires a dedicated kernel driver, which makes it very inconvenient for non-privileged end-users willing to test application performance under custom memory performance.

An opposite strategy consists in making sure some data buffers are always cached. It guarantees cache hits, thus increasing bandwidth and decreasing latency. These cache locking or pseudo-locking techniques are available in many modern processors [33]. They are often used to improve the predictability of performance, but may also simulate high-performance memory. However, this does not exactly match HBM characteristics since current HBMs do not have better latencies than DRAM.

[7] `resctrl` virtual file-system.
[8] ARM software support is still missing in Linux as of kernel 6.2.
[9] Accessible through the `/proc/mtrr` special file.

Another approach consist in changing the available capacity of caches (usually the shared L3) thanks to cache partitioning [34]. Reducing cache capacity increases the likeliness of cache misses, hence decreasing average bandwidth and increasing average latency. This feature is available in most modern architectures (together with memory bandwidth throttling presented in Section III-E). Caches are usually divided into slices (between 10 and 20 on current architectures), each slice being available to a single partition of processes. Hence, it is possible to configure the cache with a granularity of 5 or 10 %. However, given that all memory accesses go through the cache, it is not possible to make this approach specific to some data buffers or some NUMA nodes.

Contrary to marking memory uncachable, cache partitioning and cache pseudo-locking are reasonably easy to use on Linux since administrator privileges are only required for creating the partitions and assigning processes in the `resctrl` virtual filesystem.

G. Summary of Performance Emulation

Performance emulation consists in modifying the performance of memory accesses to study the execution of an application on different memory technologies and on heterogeneous memory platforms. Even if the runtime does not actually know that memory is heterogeneous, it enables testing the performance and behavior of the code on different hardware.

TABLE IV compares the advantages and drawbacks of eight software techniques presented in the previous sections. The simulator is the most flexible approach but its slowness makes it hardly usable for studying large applications. Most approaches only allow to slow down memory access for both latency and bandwidth; for these cases, "*follows Bandwidth*" in the table means that latency and bandwidth cannot be impacted separately. Although it is useful for simulating NVDIMMs over DRAM, it effectively prevents from precisely simulating a platform with HBM and DRAM (different bandwidths but similar latencies). Overall latency is currently difficult to manipulate without simulator or compiler-based techniques.

The granularity of the changes, ranging from the entire application to individual NUMA node or even individual access, is also something worth considering. If all application accesses are slowed down, it is only possible to study the application on different homogeneous platforms with different performance. This is useful to identify data buffers and kernels that are sensitive to latency or bandwidth. However, a NUMA node granularity is required for testing actual heterogeneous environments since one will usually emulate one NUMA node as fast memory and another one as slow memory.

IV. Environment Emulation

Emulating heterogeneous memory systems by modifying memory access performance allows to measure the impact of such architectures on applications. However, developing a portable application requires to adapt its behavior to the actual hardware organization. The number of NUMA nodes varies from one machine to another, the availability of HBM, NVDIMM or CXL may also vary, as well as their actual location within the platform. HPC applications and runtimes have to detect all these resources at runtime to decide which one to use. We need ways to expose heterogeneous systems to verify that these codes detect HBM, DRAM and NVDIMMs correctly and adapt their behavior to their existence and location. We call this *Environment Simulation*: we want to expose a heterogeneous memory system from the operating system to the applications even if memory access performance is not heterogeneous.

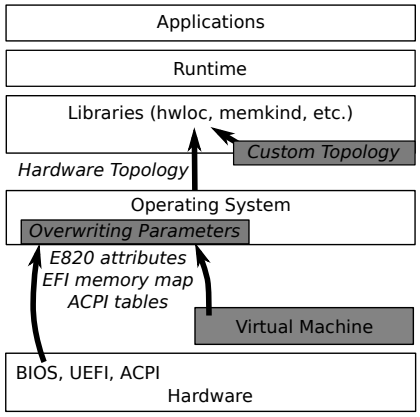


Fig. 5: Emulating heterogeneous memory in the software stack. Grey boxes indicate solutions for changing the hardware information. ACPI tables or E820/EFI memory attributes may be overwritten in the Linux kernel. An intermediate virtual machine or user-space libraries may also expose completely different hardware to upper software layers.

Emulating a heterogeneous memory environment means that the upper layers, either the application or the HPC runtime, must think it is running on heterogeneous hardware. It requires lower layers, usually the hardware or operating system, to expose fake heterogeneous memory information, even if the physical platform is actually homogeneous. As depicted on Fig. 5, this may be performed by modifying the information exposed by the firmware of the platform (ACPI tables and memory attributes in E820/EFI tables) or by low-level HPC libraries such as hwloc, or by using an intermediate virtual machine.

We detail all these strategies in the next sections and compare them in terms of flexibility (ability to add or remove NUMA nodes, to change their kind of memory, etc.), difficulty (technical complexity and privileges required), and overhead (at configuration, and at runtime).

A. Virtual Machines

Virtual machines are widely used for creating lots of individual servers on a single physical machine, but they were also created for emulating hardware one does not have access to. This provides developers with a way to access more kinds of hardware and even access architectures that do not exist yet. In fact Qemu [35] often supports new features even before they are available in real hardware (for instance CXL). Operating

TABLE IV: Summary of advantages and drawbacks of software techniques to make memory performance heterogeneous.

Emulation	Simulator (III-A)	Compiler (III-B)	NUMA Distance (III-C)	Pirate (III-D)
Impact on Bandwidth	Up or Down	Up or Down	Down only	Down only
Impact on Latency	Up or Down	Up or Down	follows Bandwidth	follows Bandwidth
Tunable Impact	Yes	No	Moderate	Moderate
Granularity	NUMA node	Instruction	NUMA node	NUMA node
Configuration Complexity	Hard	Hard	Easy	Moderate
Root privileges	No	No	No	No
Runtime overhead	Very High	—	—	—

Emulation	BW Throttling (III-E)	Uncachable (III-F)	Cache locking (III-F)	Cache partitioning (III-F)
Impact on Bandwidth	Down only	Down only	Up only	Down only
Impact on Latency	—	follows Bandwidth	follows Bandwidth	follows Bandwidth
Tunable Impact	Yes	No	No	Yes
Granularity	Application	Data Buffer	Data Buffer	Application
Configuration Complexity	Moderate	Hard	Hard	Moderate
Root privileges	Moderate	Yes	Moderate	Moderate
Runtime overhead	—	—	—	—

system developers may test drivers on Qemu so that the code is available publicly before actual products are launched.

```
qemu-system-x86_64 -machine pc,nvdimm=on,hmat=on
# 8 CPU cores and 4G of DRAM and 8GB total
smp cpus=8 -m 4G,maxmem=8G
# 4 NUMA nodes #0-3 with 2 CPU cores and 1GB each
-object memory-backend-ram,size=1G,id=dram0
-numa node,nodeid=0,memdev=dram0,cpus=0-1
-object memory-backend-ram,size=1G,id=dram1
-numa node,nodeid=1,memdev=dram1,cpus=2-3
[...]
# 2 NVDIMM nodes #4-5 with 2GB each
-object memory-backend-file,id=nvdimm1,size=2G
-device nvdimm,id=nvdimm1,memdev=nvdimm1,node=4
-object memory-backend-file,id=nvdimm1,size=2G
-device nvdimm,id=nvdimm1,memdev=nvdimm1,node=5
# NUMA distances (ACPI SLIT)
-numa dist,src=0,dst=0,val=10
-numa dist,src=0,dst=1,val=20
[...]
# memory access latency (ACPI HMAT), NVDIMMs slower
-numa hmat-lb,initiator=0,target=0,latency=10
-numa hmat-lb,initiator=0,target=1,latency=20
[...]
-numa hmat-lb,initiator=0,target=4,latency=100
-numa hmat-lb,initiator=0,target=5,latency=200
[...]
```

Fig. 6: Qemu command-line to build a virtual machine with 8 CPUs and 6 NUMA nodes. First 4 nodes have 2 CPUs and 1 GB of DRAM. Last 2 nodes have 2 GB of NVDIMM and no local CPU cores. Some attributes are omitted for clarity.

Fig. 6 presents an example of Qemu command-line for building a virtual machine with a custom NUMA topology and heterogeneous memory. Although most Qemu users only specify the number of CPU cores and amount of memory available in a virtual machine, many configuration options are available to tune the NUMA nodes, non-volatile memory, their distances, performance, etc. The length of these command-lines unfortunately grows quickly since the number of options to describe the NUMA topology increases with the square of the number of NUMA nodes[10]. Fortunately, this may be easily scripted. Configuring CXL devices or even interconnects in Qemu brings even more complexity with the requirement to define root ports, fixed memory windows, devices, switches, etc. Moreover, Qemu allows to provide entire ACPI tables as files instead of specifying each individual value on the command-line (see Section IV-B for more details about using custom ACPI tables). In theory, Qemu allows to emulate a system mixing DRAM, HBM ad NVDIMM but the difference between DRAM and HBM is currently only exposed implicitly through performance attributes in the HMAT ACPI table. We explain in Section IV-C how to ask the Linux kernel to explicitly mark some memory ranges as HBM so that applications may clearly identify them.

Virtual machines are usually slower than the physical host they run on [36]. The impact on HPC applications depends on the ratio of computation, memory access, I/O and communication. Fortunately, performance is not the goal of this discussion. Hence, virtual machines are an easy solution for emulating the environment of heterogeneous memory systems. It means applications will be tested to check whether their behavior is correct. For instance, the application should identify the NUMA node that is HBM and use it for allocating bandwidth-sensitive buffers. However, the virtual machine cannot be used for actually measuring the performance benefit of HBM on the application.

B. Replacing ACPI Tables

As explained in Section II-B, ACPI tables are the most important resource to describe the hardware topology including CPU cores and memories. Each platform has a firmware that exposes ACPI tables during boot. The Linux kernel allows to override these hardware tables with custom ones by placing them in the `initrd` boot file [37]. This is useful for temporarily avoiding firmware bugs until the vendor provides a firmware update, but also for modifying the CPU and memory hardware organization for development.

Tables are stored in hardware in the *ACPI Machine Language* binary format, which can be converted from/to human-readable *ACPI Source Language* with tools such as `iasl` as

[10]Some examples of Qemu configurations for simulating NUMA, HMAT, HBM, NVDIMMs and CXL used for hwloc development are available at https://github.com/open-mpi/hwloc/wiki/Simulating-complex-memory-with-Qemu.

```
                    Signature : "HMAT"
                              [...]
                Structure Type : 0001
                    Data Type : 00
  Initiator Proximity Domains # : 00000002
     Target Proximity Domains # : 00000004
               Entry Base Unit : 00000000000000c8
Initiator Proximity Domain List : 00000000
Initiator Proximity Domain List : 00000001
   Target Proximity Domain List : 00000000
   Target Proximity Domain List : 00000001
   Target Proximity Domain List : 00000002
   Target Proximity Domain List : 00000003
                         Entry : 01F4
                         Entry : 03E8
                         Entry : 01F4
                         Entry : 03E8
                         Entry : 03E8
                         Entry : 01F4
                         Entry : 03E8
                         Entry : 01F4
                             [...]
```

Fig. 7: HMAT table in ACPI Source Language. The table is made of multiple structures. The one shown here (*Structure Type 0001*) contains *System Locality Latency and Bandwidth Information* which lists the access latency (*Data Type* 00) between all initiator (2 CPUs) and all targets (4 memories). Values are on the *Entry* lines and will be multiplied by the *Entry base* unit to get latencies in picoseconds.

shown on Fig. 7. Modifying existing tables is then only a matter of replacing existing structures or values with new ones. ACPI specifications are useful for finding the appropriate fields to modify but `iasl` even annotates the source with names of structures and fields to make things understandable by humans. Modifications are indeed easy when moving some CPUs from one NUMA node to another, or even replacing NUMA distance values.

However, it is also possible to change the number of NUMA nodes by splitting existing ones. This requires to split address range structures in the SRAT table, associate new proximity domain values, and update SLIT, HMAT and possibly NFIT tables too. This complex work can fortunately be simplified with tools such as `daxctl split-acpi`. Then most attributes of new NUMA nodes (size, locality, distance, performance, type of memory, etc.) may be changed manually.

Another possibility consists in using tools such as Qemu to generate a virtual machine with the expected hardware memory subsystem (see Section IV-A). Qemu builds custom ACPI tables that may be exported for use on a physical machine.

Although this approach allows to add, split, merge and modify NUMA nodes by moving memory ranges between nodes, the overall memory capacity of the modified machine is still limited to the physical memory.

C. Modifying the Memory Ranges exposed by the OS

As explained in Section II-B, ranges of physical memory are also described either in the legacy BIOS E820 tables or in the modern UEFI memory map. The Linux kernel may be asked to tweak these tables to change the attributes of some ranges. In legacy BIOS mode, the `memmap` kernel boot parameter allows to mark a range of volatile memory as non-volatile [38]. In UEFI mode, the `efi_fake_mem` kernel boot parameter allows marking ranges as non-volatile or *soft-reserved*. This strategy was used for developing non-volatile memory-based storage solutions with DRAM when NVDIMMs were not available yet [39].

When it comes to heterogeneous memory, any range changed as non-volatile or soft-reserved will be exposed as a DAX device on Linux. As explained earlier, these DAX devices may then be converted to *system-ram* and appear as separate NUMA nodes. This effectively allows to test the behavior of applications when the platform contains different kinds of memory. However, this approach is not as flexible as modifying ACPI tables: it may change the type of memory within a given NUMA node, but it cannot add, split or merge NUMA nodes[11].

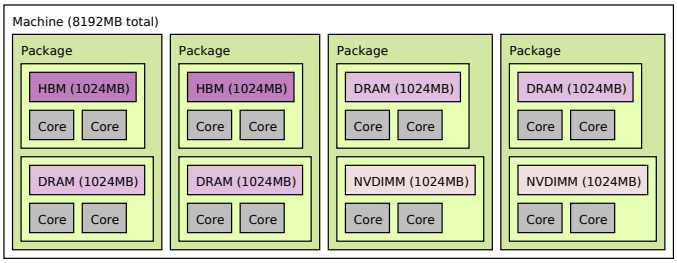


Fig. 8: `lstopo` graphical output of a platform with modified EFI memory attributes. Out of 8 DRAM NUMA nodes, 4 were changed into HBM or NVDIMMs. Capacity and core-NUMA locality cannot be changed with this strategy.

Hence, this strategy is only suitable when the platform contains enough physical NUMA nodes to convert some of them into different types. For instance a 4-socket platforms with 2 DRAM NUMA nodes in each CPU (8 nodes total) could be disguised as 2 sockets with HBM and DRAM and 2 other sockets with DRAM and NVDIMMs, as shown on Fig. 8. Fortunately many recent processors are able to expose 2 or 4 NUMA nodes, hence many dual-socket platforms easily provide enough nodes to meet this requirement.

Additionally, Linux allows administrators to mark some memory ranges as offline to artificially reduce the available memory capacity[12]. This is useful to verify the ability of applications to fallback to larger memory tiers when the HBM is full.

D. Modifying the runtime view of the hardware

Virtual machines, ACPI tables and E820/EFI attributes change the hardware features exposed by the operating system. However, HPC applications rarely directly talk to the operating system when it comes to managing heterogeneous memory.

[11]Marking half a DRAM NUMA node as non-volatile with the `memmap` parameter cannot result in one DRAM node and one NVDIMM node because they belong to the memory range of the same physical node.

[12]Writing in special files under `/sys/devices/system/memory/` changes the state of unused contiguous 2 GB chunks on x86 platforms.

They usually rely on intermediate software libraries, either thin layers such as `libnuma` or actual HPC runtime libraries such as `memkind` [12] or `hwloc` [40]. Those tools take care of identifying NUMA nodes in the hardware information exposed by the operating system. On Linux, this requires to parse hundreds of files in the `sysfs` virtual filesystem, a task that most application developers are happy to delegate.

`memkind` provides users with an option to expose some DRAM NUMA nodes as HBM. This allows testing HBM-applications on non-HBM platforms to verify that the bandwidth-sensitive buffers are indeed allocated on the (fake) HBM nodes. `hwloc` goes much further by allowing to load the topology of a completely different platforms (stored as a XML file), from CPUs to caches, NUMA and I/O peripherals. As hwloc is used by the majority of HPC runtimes, it provides a *de-facto* standard way to test algorithms on a wide variety of platforms without having access to them. Locality-aware heuristics such as top-down recursive partitioning for placing task and data may for instance be tested on different hierarchical architectures [41].

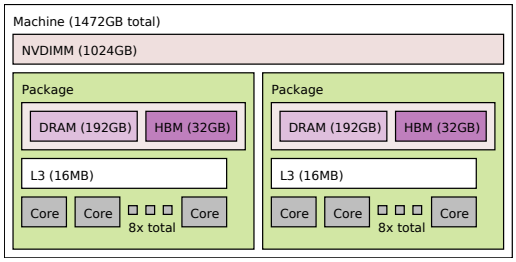


Fig. 9: `lstopo` graphical output of a fictitious 3-memory-kind machine. It was built with hwloc by describing the resource hierarchy as one machine containing one NUMA node and 2 CPU packages, and each package containing 2 NUMA nodes and one L3 cache shared by 8 cores. NUMA node types were specified afterwards (NVDIMM, DRAM and HBM).

Fig. 9 presents an example of non-existing platform built with hwloc. The procedure consists in modifying an existing platform topology, or creating a new one by describing the hierarchy of resources. hwloc then allows to remove some cores or NUMA nodes or modify their attributes (capacity, type of memory, etc.) [42]. Once the new hwloc topology is ready, the exported XML file may be loaded into any hwloc-aware HPC runtimes to simulate an execution on a different platform. Similar topologies may be obtained by modifying ACPI tables or using virtual machines but this methods is much user-friendlier.

This approach consists in having a library lie to the upper layers (runtime or application) when exposing hardware resources. As soon as the application actually uses the resources, the lies may become an issue: what if the application decides to run a task on the 7th CPU core and allocate its memory in HBM NUMA node #4 while the physical hardware has only 4 cores and 2 NUMA nodes? Hence, this approach may be risky unless the physical hardware contains at least as many cores and NUMA nodes as the virtual hardware exposed to the application. However, this approach is technically easy since it does not require any administration privileges (root access, modify the operating system boot parameters, rebooting, etc.) or any complex procedures such as modifying ACPI tables.

Additionally, such user-space techniques may be used to artificially limit the memory capacity. A runtime (or an intercepting preloaded library) may return an allocation failure before the backend memory is actually full. This is useful on Linux because *Control Groups* (the main way to limit resource usage in processes) currently do not allow to limit memory use inside a specific NUMA node.

E. Summary of Environment Emulation

Environment emulation consists in exposing a heterogeneous memory system to the application so that developers may update their code to identify different kinds of memory, allocate on the right target, implement placement heuristics, etc. It does not provide any way to check the application performance but offers an easy solution to verify its behavior.

We presented four techniques that lie either at the user level, or in the firmware, or in an intermediate virtual machine. TABLE V summarizes their drawbacks and advantages in term of flexibility, difficulty and overhead. Some of them still need to be combined since they do not provide enough flexibility alone (for instance ACPI tables must be combined with EFI attributes to expose HBM memory properly). We believe all these strategies allow developers to test their code under a wide variety of emulated platforms to prepare it for future heterogeneous memory systems.

V. Conclusion

Heterogeneous memory is a way to address the needs of different applications where latency, bandwidth and capacity requirements vary. Several hardware platforms with heterogeneous memory are going to be released in the next years on the road toward exascale. Application developers must start adapting their code to those platforms. We presented numerous software techniques that will help them deploy an environment to perform this development.

Performance emulation consists in modifying the performance of some memory accesses so that the application behaves as on a real heterogeneous system. This is useful for identifying which data buffers and compute kernels are sensitive to memory bandwidth and latency. Many hardware features such as bandwidth throttling or NUMA distance may be used to emulate such a platform, as well as actual hardware simulators or compiler-based techniques.

Environment emulation consists in exposing information about heterogeneous memory to the runtime even if performance is not heterogeneous. This is useful for verifying that the code is able to identify different kinds of memory such as HBM, DRAM and NVDIMMs, and allocate its sensitive buffers on the right one. This emulation may rely on virtual machines as well as modifying the physical host topology by replacing its ACPI tables or memory attributes.

TABLE V: Summary of advantages and drawbacks of software techniques to build heterogeneous memory environments.

Emulation	Virtual machine (IV-A)	ACPI tables (IV-B)	EFI/E820 attributes (IV-C)	Runtime (IV-D)
Add/Remove NUMA nodes	Yes	Yes	No	No
Change type of memory	No explicit HBM	No explicit HBM	Yes	Yes
CXL memory	Yes	No	No	Yes
Difficulty	Moderate	Hard	Moderate	Easy
Require root privileges	No	Yes	Yes	No
Configuration overhead	VM reboots	Reboots	Reboots	—
Runtime overhead	Slower execution	—	—	—

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/).

Some experiments were performed on the SACADO MeSU platform (SGI Altix UV2000) at Sorbonne-Université.

REFERENCES

[1] 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.

[2] 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.

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

[4] "Arm welcomes you to SC'21," 2021, https://community.arm.com/arm-community-blogs/b/high-performance-computing-blog/posts/arm-returns-to-sc21.

[5] I. B. Peng, M. B. Gokhale, and E. W. Green, "System Evaluation of the Intel Optane Byte-addressable NVM," in *The Fifth International Symposium on Memory Systems Proceedings (MEMSYS19)*. Washington, DC: ACM, Oct. 2019.

[6] A. Li, S. L. Song, J. Chen, J. Li, X. Liu, N. R. Tallent, and K. J. Barker, "Evaluating Modern GPU Interconnect: PCIe, NVLink, NV-SLI, NVSwitch and GPUDirect," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 1, pp. 94–110, 2020.

[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] T. M. Coughlin and J. Handy, "Higher performance and capacity with omi near memory," in *2021 IEEE Symposium on High-Performance Interconnects (HOTI)*, 2021, pp. 68–71.

[9] M. Feldman, A. Norton, , and E. Joseph, "CXL and Gen-Z Consortiums Combine Forces," Jul. 2020, Hyperion Research.

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

[11] "Advanced Configuration and Power Interface (ACPI) and Unified Extensible Firmware Interface (UEFI) Specifications." [Online]. Available: https://uefi.org/specifications

[12] C. Cantalupo, J. R. Hammond, and S. Hammond, "User Extensible Heap Manager for Heterogeneous Memory Platforms and Mixed Memory Policies," no. 2, pp. 1–17, 2015. [Online]. Available: http://memkind.github.io/memkind/

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

[14] B. Goglin and A. R. Proa˜no, "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. [Online]. Available: http://hal.inria.fr/hal-03599360

[15] 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 the IEEE International Conference on Cluster Computing*, Hawaii, USA, Sep. 2017.

[16] 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.

[17] M. B. Olson, B. Kammerdiener, M. R. Jantz, K. A. Doshi, and T. Jones, "Portable Application Guidance for Complex Memory Systems," in *The Fifth International Symposium on Memory Systems Proceedings (MEMSYS19)*. Washington, DC: ACM, Oct. 2019.

[18] J. Renau, B. Fraguela, J. Tuck, W. Liu, M. Prvulovic, L. Ceze, S. Sarangi, P. Sack, K. Strauss, and P. Montesinos, "SESC simulator," January 2005, http://sesc.sourceforge.net.

[19] V. Weaver and S. McKee, "Are cycle accurate simulations a waste of time?" 12 2010.

[20] S. Li, R. S. Verdejo, P. Radojković, and B. Jacob, "Rethinking cycle accurate dram simulation," in *Proceedings of the International Symposium on Memory Systems*, ser. MEMSYS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 184–191.

[21] G. Fursin, M. F. P. O'Boyle, O. Temam, and G. Watts, "A fast and accurate method for determining a lower bound on execution time," *Concurrency and Computation: Practice and Experience*, vol. 16, no. 2-3, pp. 271–292, 2004.

[22] S. Koliaï, Z. Bendifallah, M. Tribalat, C. Valensi, J.-T. Acquaviva, and W. Jalby, "Quantifying performance bottleneck cost through differential analysis," in *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ser. ICS '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 263–272.

[23] H. Volos, G. Magalhaes, L. Cherkasova, and J. Li, "Quartz: A Lightweight Performance Emulator for Persistent Memory Software," in *Proceedings of the 16th Annual Middleware Conference*, ser. Middleware '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 37–49.

[24] S. Pelley, T. F. Wenisch, B. T. Gold, and B. Bridge, "Storage management in the nvram era," *Proc. VLDB Endow.*, vol. 7, no. 2, p. 121–132, oct 2013.

[25] I. B. Peng, S. Markidis, E. Laure, G. Kestor, and R. Gioiosa, "Exploring application performance on emerging hybrid-memory supercomputers," in *2016 IEEE 18th International Conference on High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, 2016, pp. 473–480.

[26] D. Eklov, N. Nikoleris, D. Black-Schaffer, and E. Hagersten, "Cache pirating: Measuring the curse of the shared cache," in *Proceedings of the International Conference on Parallel Processing*, 2011, pp. 165–175.

[27] H. Hanson and K. Rajamani, "What computer architects need to know about memory throttling," in *Computer Architecture*, A. L. Varbanescu, A. Molnos, and R. van Nieuwpoort, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 233–242.

[28] B. Li and E. A. León, "Memory Throttling on BG/Q: A Case Study with Explicit Hydrodynamics," in *6th Workshop on Power-Aware Computing and Systems (HotPower 14)*. Broomfield, CO: USENIX

Association, Oct. 2014. [Online]. Available: https://www.usenix.org/conference/hotpower14/workshop-program/presentation/li

[29] B. Li, E. A. León, and K. W. Cameron, "COS: A Parallel Performance Model for Dynamic Variations in Processor Speed, Memory Speed, and Thread Concurrency," in *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 155–166.

[30] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson, "System software for persistent memory," in *Proceedings of the Ninth European Conference on Computer Systems*, ser. EuroSys '14. New York, NY, USA: Association for Computing Machinery, 2014.

[31] J. Park, S. Park, and W. Baek, "CoPart: Coordinated partitioning of last-level cache and memory bandwidth for fairness-aware workload consolidation on commodity servers," in *Proceedings of the 14th EuroSys Conference 2019*, vol. 19. New York, NY, USA: ACM, 2019.

[32] C. Foyer and B. Goglin, "Using Bandwidth Throttling to Quantify Application Sensitivity to Heterogeneous Memory," in *Workshop on Memory Centric High Performance Computing, MCHPC'21*. St. Louis, MO, USA: IEEE, Nov. 2021. [Online]. Available: https://hal.inria.fr/hal-03356585

[33] S. Mittal, "A survey of techniques for cache locking," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 21, no. 3, may 2016.

[34] J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan, "Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems," in *2008 IEEE 14th International Symposium on High Performance Computer Architecture*, 2008, pp. 367–378.

[35] "QEMU: A generic and open source machine emulator and virtualizer." [Online]. Available: https://www.qemu.org/

[36] A. Kudryavtsev, V. Koshelev, B. Pavlovic, and A. Avetisyan, "Virtualizing hpc applications using modern hypervisors," in *Proceedings of the 2012 Workshop on Cloud Services, Federation, and the 8th Open Cirrus Summit*, ser. FederatedClouds '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 7–12.

[37] "Upgrading ACPI tables via initrd." [Online]. Available: https://www.kernel.org/doc/Documentation/acpi/initrd_table_override.txt

[38] "The kernel's command-line parameters." [Online]. Available: https://www.kernel.org/doc/Documentation/admin-guide/kernel-parameters.txt

[39] F. Chen, M. P. Mesnier, and S. Hahn, "A protected block device for Persistent Memory," in *2014 30th Symposium on Mass Storage Systems and Technologies (MSST)*, 2014, pp. 1–12.

[40] 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.

[41] E. Jeannot, G. Mercier, and F. Tessier, "Process Placement in Multicore Clusters: Algorithmic Issues and Practical Techniques," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 4, pp. 993–1002, 4 2014.

[42] "hwloc – How do I create a custom heterogeneous and asymmetric topology?" [Online]. Available: https://www.open-mpi.org/projects/hwloc/doc/v2.7.2/a00373.php#faq_create_asymmetric