Exploring Data Migration for Future Deep-Memory Many-Core Systems

Swann Perarnau,* Judicael A. Zounmevo,* Balazs Gerofi,† Kamil Iskra,* Pete Beckman*
*Argonne National Laboratory. {swann, jzounmevo}@anl.gov, {iskra, beckman}@mcs.anl.gov
†RIKEN Advanced Institute for Computational Science. bgerofi@riken.jp

***Abstract*—Upcoming high-performance computing (HPC) platforms will have more complex memory hierarchies with high-bandwidth on-package memory and in the future also non-volatile memory. How to use such deep memory hierarchies effectively remains an open research question.**

In this paper we evaluate the performance implications of a scheme based on a software-managed scratchpad with coarse-grained memory-copy operations migrating application data structures between memory hierarchy levels. We expect that such a scheme can, under specific circumstances, outperform a hardware-managed cache while requiring a lot less effort than would a scheme managed entirely by the application programmers.

Because suitable hardware is not yet generally available, we propose and benchmark several existing hardware configurations that can be used as approximations, including non-uniform memory access (NUMA) systems and memory on accelerators. We then evaluate data migration mechanisms currently available on Linux systems, such as `move_pages` and `memcpy`. We also design a best-case-scenario HPC benchmark to explore how the memory locality and parallelism of applications can be improved by data migration.

We find that NUMA systems can be a reasonable approximation platform, especially when auxiliary load mechanisms are employed. Memory migration mechanisms inside the Linux kernel turn out to significantly lag behind a plain user-space memory copy, even after we level the playing field as much as possible. Our dedicated application benchmark demonstrates a significant performance benefit of doing memory migrations—approaching the measured difference in the memory bandwidth—provided that the ratio of worker threads to migration threads is chosen well.

I. Introduction

Future high-performance computing (HPC) platforms are expected to feature compute nodes with hundreds of hardware threads and significantly more complex memory hierarchies [1]. Intel's Knights Landing processor, soon to be made available, already hosts up to 288 hardware threads and a new level in the memory hierarchy: the high-bandwidth multichannel DRAM (MCDRAM) located on the processor package. This memory is expected to deliver more than five times the bandwidth of regular DRAM [2], which is now pushed farther in the memory hierarchy. With only 16 GiB in the best case, however, Knights Landing's MCDRAM may be too small to fit many HPC applications.

This presents a dilemma: how should a deep-memory hierarchy be managed, and who should be managing it? Should applications be statically allocating individual data structures explicitly at the most suitable hierarchy level depending, for example, on data size or expected access pattern? Should the *fast* memory be instead managed dynamically, as a cache, and if so, at what granularity (cache line size, page size, whole data structure) and who should be responsible for the policy and the mechanism (application, runtime, OS kernel, hardware)? In general, we have little faith in static schemes managed solely by the applications; beyond heroic efforts of few and far between "ninja" programmers, the history of computing is littered with examples of powerful yet complex mechanisms being misused or simply unused. On the other hand, hiding the fast memory as a transparent, hardware-managed new level of cache, while highly practical in that it requires no changes to the applications, strikes us as a lazy way out that misses the opportunity to perform advanced optimizations that could benefit the applications more than the necessarily basic and inflexible cache replacement policies implemented in hardware.

In this paper we explore the *middle ground* between these two extremes. Our focus is on using the fast memory as a software-managed scratchpad, with coarse-grained memory-copy operations migrating application data structures between memory hierarchy levels to best match the dynamically changing working sets of application kernels. We expect that with sufficient information about the application — obtained through programming directives in the source code, feedback from the compiler analysis phases, and dynamic performance data from the runtime system — a middleware-managed solution should, under specific circumstances, be able to outperform a hardware-managed cache while requiring a lot less effort than would a scheme managed entirely by the application programmers. Clear parallels exist between this approach and the partitioned global address space (PGAS) programming model or out-of-core computations.

Application workload that could benefit from such a scheme needs to exhibit certain characteristics. Chief among them, its memory access pattern must be predictable enough that the data can be prefetched ahead of time, effectively hiding the cost of migration. Beyond that, its access pattern must be sufficiently localized that when a portion of application's data is migrated from slow to fast memory, the majority of subsequent memory accesses will hit the latter. Of course, the problem needs to be bandwidth-bound when running in slow memory so that a bottleneck exists that can be overcome by moving data to fast memory. The dynamic working set size must be "just right," fitting in fast memory but being

2168-9253 2016
U.S. Government Work Not Protected by U.S. Copyright
DOI 10.1109/CLUSTER.2016.42

too large to fit in the CPU cache. Software-managed solutions bring their own overheads. We assume no hardware assistance in the form of, for example, a DMA engine, so we will need to take a portion of compute resources away from the application to perform data migration in the background. The effectiveness of such a solution will depend on both the application characteristics (thread count, data parallelism, bandwidth requirements) and hardware characteristics (core count, bandwidth and latency of each memory level).

Unfortunately, without access to suitable hardware (general-purpose processors with on-package memory are not available yet) identifying how those characteristics will impact the effectiveness of data migration is difficult. This paper describes early experiments, using current hardware to approximate the features of such a deep-memory system, to explore the performance of data migration and its usefulness to HPC applications.

The contributions of this paper are as follows. First, we present several available hardware configurations that can be used to test data migration schemes for deep-memory systems. Using state-of-the-art methods, we measure the configurations' latency and bandwidth. Second, we identify the various data migration mechanisms currently available on Linux systems and study their performance on our platforms, depending on the number of hardware threads dedicated to migration. Third, we design an HPC benchmark to explore how the memory locality and parallelism of applications can be improved by data migration. In particular, we study trade-offs between the number of threads dedicated to migration and to the application and the size of the datasets migrated back and forth.

The rest of this paper is organized as follows. Section II outlines the future many-core deep-memory platforms, and in particular Intel's Knights Landing. Section III describes the platforms we used to approximate such deep-memory architectures and our measurements of their characteristics. Section IV explains the current facilities available to an application to migrate data between physical memory locations and evaluates their performance. In Section V we present an application benchmark for our use case and explore how modifying its characteristics affects the performance. Section VI discusses related work. We conclude in Section VII with a summary and a brief look at future work.

II. Deep-memory Platforms

To the best of our knowledge, Knights Landing will be the first HPC-oriented, general-purpose processor to directly address two levels of memory beyond the CPU cache hierarchy. Consequently, this section focuses on the memory hierarchy of this CPU.

Knights Landing is the second generation of the Intel Xeon Phi many-core processor. It will be available either as a bootable host processor or as a PCIe coprocessor. Our work here focuses on the bootable host processor version. The processor will have up to 72 CPU cores, each capable of hosting four hardware threads.

The processor is organized in tiles of two cores each. Each core has an 8-way associative, 32 KiB L1 Icache and 32 KiB L1 Dcache. The L2 cache is 1 MB large, 16-way associative, coherent, and shared between the two cores of a tile. The first-level (unified) TLB offers 64 entries for 4 KiB pages while the second level, meant for data only, offers 256 entries for 4 KiB pages, 128 entries for 2 MiB pages, and 16 entries for 1 GiB pages.

The 16 GiB of on-board MCDRAM can be coupled with up to 384 GiB of 6-channel DDR4 memory. The MCDRAM is expected to deliver over 400 GB/s in peak bandwidth while the DDR4 will peak at about 90 GB/s; thus, the on-package memory will offer more than five times the bandwidth of the off-package DRAM. The MCDRAM can be used as follows [3]:

Cache A direct-mapped 64 B-lines cache, with some overhead in each line for tag information.

Flat A regular memory that extends the DRAM from the upper physical addresses end. In this mode, the MCDRAM is exposed as a second NUMA node.

Hybrid 25% or 50% of the MCDRAM used as cache while the rest extends the DRAM in the form of a separate NUMA node.

Based on the above, in the rest of paper, we make the following assumptions about the platform: two levels of DRAM hierarchy with a limited-capacity fast memory and a large-capacity slow memory; a five-times difference in bandwidth between the two tiers; and no hardware offloading for data migration. Given a lack of information on access latency differences between the two tiers, we assume no noteworthy improvements. The target we eventually want to compare with is the *Cache* mode of MCDRAM described above.

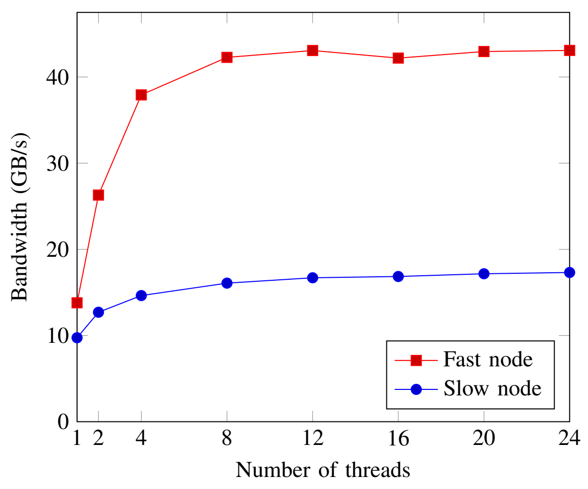
Other companies are also working in this area. Nvidia plans to incorporate high-bandwidth memory in its Pascal GPUs [4]. AMD already has a GPU product with high-bandwidth memory [5] on the market (Fiji) and is planning to release a hybrid Zen APU that incorporates such memory as well.

III. Approximation Platforms

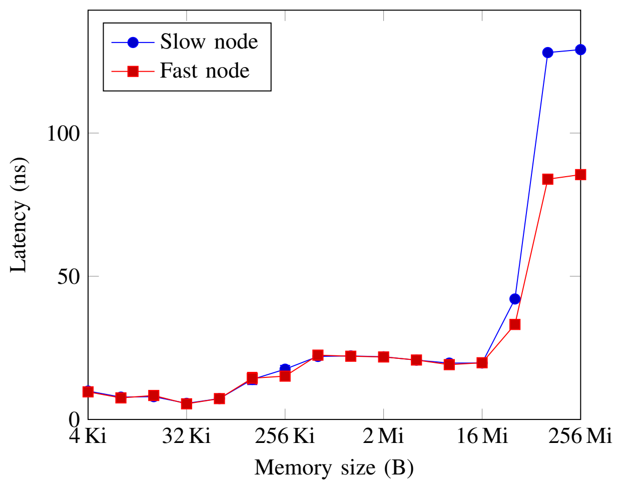
Although we do not have access to deep-memory architectures, and to Knights Landing in particular, we can evaluate the usefulness of data migration on platforms *approximating* their expected characteristics. For this purpose, we use three experimental setups: two based on a NUMA machine and one on the previous-generation Xeon Phi (Knights Corner). Each of these setups exhibits different hardware characteristics in terms of latency and bandwidth of different memory levels and number of hardware threads. This variability helps us explore the impact of each characteristic on the usefulness of data migration. We describe here each platform and its performance.

A. NUMA System

One straightforward way to test a platform with several levels of memory that can be managed from an application



(a) STREAM Add reported performance over a 40 MiB array, between 1 and 24 hardware threads on the main socket.



(b) Memory access latency of a random walk across a linked list of increasing size.

Figure 1: Hardware characteristics of our NUMA platform.

is to use a NUMA machine. Such a system exhibits several memory nodes that are at a different *distance* from the various cores available.

Although we do not expect a future deep-memory system to exhibit the same latency/bandwidth ratios between memory nodes, this provides us with an easily accessible first approximation platform.

Our NUMA system of choice is a dual-socket node using Intel Xeon E5–2670 v3 "Haswell" processors (each with 12 cores running at 2.3 GHz). Two memory nodes are present on this machine, each 64 GiB in size. Each socket also contains a 30 MiB shared L3 cache and 256 KiB L2 caches private to each core. This system is running a vanilla 4.1.3 Linux kernel. For the remainder of this paper, we will always use the same execution context: we run the benchmarks on CPU socket 0 (main socket) and use the memory node 0 (closest to this socket) as the *fast* node. The memory node 1 (closer to the second CPU socket) is used as the *slow* memory. All measurements reported in the remainder of this paper are the average of 10 runs (including STREAM which already reports the best of 10 internal measurements). Unless visible on the figure, confidence intervals at 95% are too small to show. All the benchmarks were compiled with GCC 4.8.4 with optimizations (`-O2`), except for the codes where the exact memory behavior matters (latency benchmark, memory loader), which were compiled without optimizations (`-O0`).

We use the following benchmarks to characterize this system. First, for memory bandwidth, we run the STREAM benchmark [6]. Our goal is to measure maximum available bandwidth from the main socket to each memory node. We also vary the number of threads used, in order to understand the minimum thread count needed to maximize the bandwidth usage. Figure 1a presents the results.

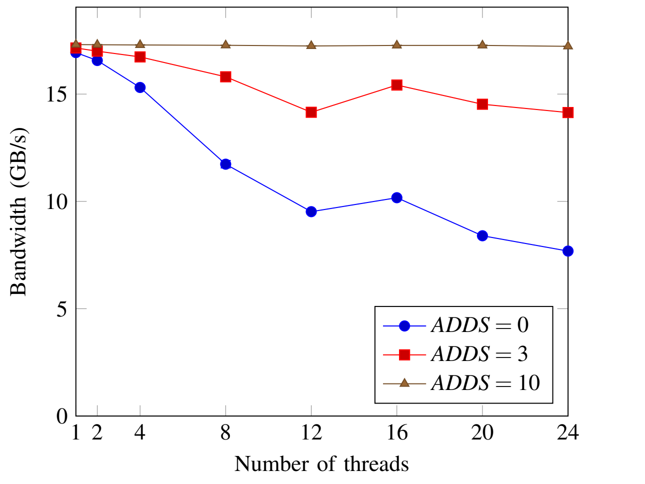
We then measure the access latency of each memory node from the main socket. To do so, we use a typical random-access benchmark [7], [8], which works by making a large number of accesses to a single, contiguous memory region containing a shuffled linked list. By adjusting the size of the memory region, we can control which level of the is exercised. To prevent the hardware prefetch mechanism from interfering, we set the size of individual elements to be twice the size of a physical cache line. Figure 1b details those measurements.

As we can see, this system exhibits both bandwidth and latency differences between the two memory nodes when accessed from the same socket. When saturated, the bandwidth of the fast node is 2.5 times better than that of the slow node. We note that the benchmark requires multiple (eight) threads to saturate the bandwidth. Thus, migrating data between the nodes will also require multiple threads in order to go as fast as possible. We also note that the slow node takes 1.5 times longer to access than does the fast node. In that case, codes whose performance is impacted by memory latency will likely benefit from data migration, too.

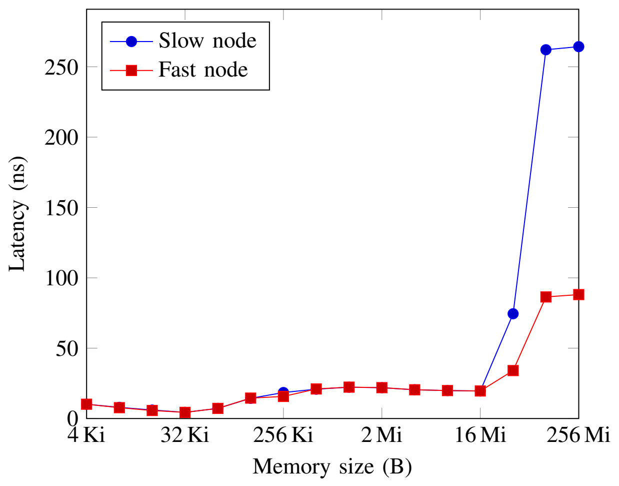
B. Degraded NUMA System

As indicated earlier, on-package memory is expected to feature on the order of five times more bandwidth than standard memory nodes do. Our NUMA system, however, exhibits a difference half as large.

Although we cannot modify this system to improve the bandwidth of the fast memory node, we can further degrade the bandwidth of the slow one. Doing so should create a setup with a latency/bandwidth ratio closer to that of deep-memory systems. Since we are using only one socket for our benchmark runs, we can use the second socket to run a *loader* workload using some of the memory bandwidth of the slow node. By controlling this loader's usage of bandwidth, we can create a setup where the fast node exhibits much better bandwidth than the slow node does.



(a) Degraded NUMA: STREAM bandwidth measured with 24 threads while running our stencil loader, for various additional instructions on each step of the stencil.



(b) Memory access latency of a random walk across a linked list of increasing size. For our chosen degraded parameters.

Figure 2: Hardware characteristics of our degraded NUMA platform.

To perform this memory load, we use a program similar to the STREAM benchmark used in the preceding section. Our loader performs a 3-point stencil across a memory region, splitting its work in equal parts for each thread. By controlling the number of threads and the size of the memory region accessed, we can control how much memory this benchmark is accessing. We also insert a configurable number of additional instructions between each step of the stencil, in order to decrease the bandwidth needed by each thread.

To demonstrate this control of the memory bandwidth of the slow node, we run concurrently our loader (on the second socket) and STREAM (24 threads on the main socket). Varying either the loader's thread count or the additional instructions (ADDS) per step, we show in Figure 2a the different bandwidths we obtain.

As we can see, this loader has an impact on the bandwidth measured by STREAM. The more instructions added per step, the lower our loader uses the memory bandwidth. Similarly, increasing the amount of threads increases the bandwidth usage of our loader. Because we are interested here only in decreasing the bandwidth of the slow node statically, we choose the parameters resulting in the worst bandwidth. Specifically, we use 24 threads with $ADDS = 0$ in the remainder of this paper for experiments running on the degraded platform. These parameters result in a 55% reduction in measured bandwidth for the slow node and a 5.6 performance ratio between the nodes, which is close to the platform we attempt to approximate. The latency impact of our loader was also measured and is displayed in Figure 2b. As we can see, the fast node is not impacted, but the slow node is.

C. *Xeon Phi*

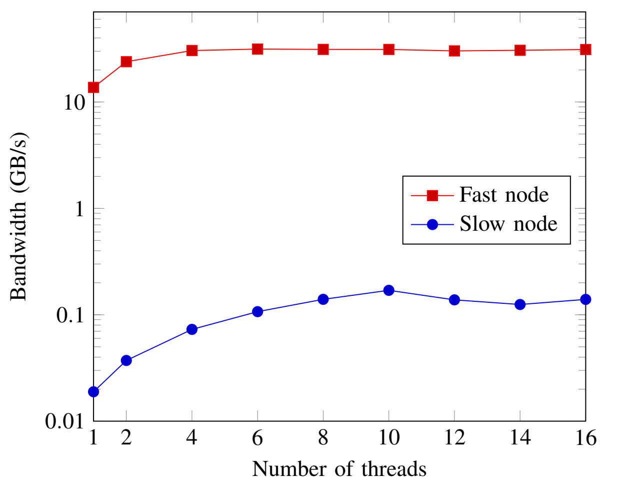
While the two previous platforms already exhibit interesting features to compare the impact of hardware characteristics on data migration, we also wanted to test more unusual setups. For this purpose, we used a system with the current generation Xeon Phi 5110P (Knights Corner) coprocessor board, which is connected to the host machine via PCI Express. The host processor is a dual-socket Intel Xeon CPU E5-2670 v2 (10 CPU cores per socket, two hardware threads per core) with 32 GiB memory connected to each socket. The Xeon Phi board is equipped with 8 GiB of RAM, which is also visible on the host machine as device memory. To mimic the far memory in a platform with deep memory hierarchy, we map the Xeon Phi's physical memory directly into user space on the host. We point out that this architecture is notably different from the purely NUMA-based environments, because memory accesses to the Xeon Phi bypass the cache hierarchy of the host CPU.[1]

As before, we measure bandwidth and latency for accessing both the host's and Xeon Phi's memory. All measurements are performed on socket 0, and for the host memory we use NUMA node 0 (we stress that all the Phi experiments run on the host; we only use the Xeon Phi card for its memory). Figures 3a and 3b present the results. We can see that the DRAM of this system behaves like our NUMA system, with similar bandwidth and latency. The Xeon Phi, being accessed through the PCIe bus, has a much higher latency and less bandwidth (respectively 14 times and over 200 times) than does the DRAM. While not the best approximation for Knights Landing, it could be useful for approximating the performance of NVRAM.

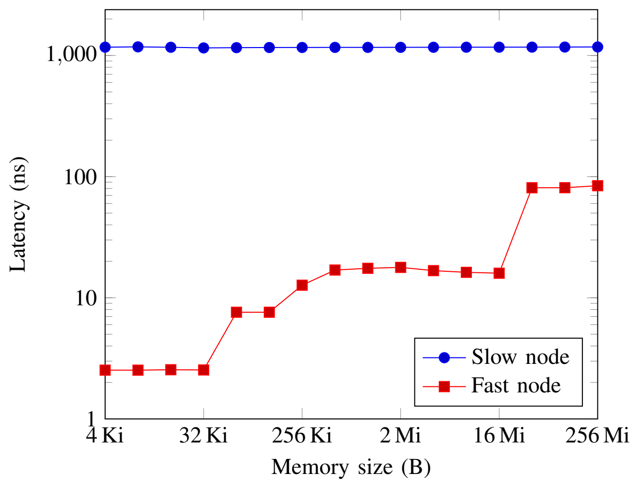
IV. Data Migration

Now that we have a collection of platforms to evaluate data migration, we can look at the various migration mechanisms currently available to user applications.

[1]The Xeon Phi is also equipped with a number of DMA engines, which provide significantly higher bandwidth; but we deliberately use direct memory mappings so that we can access it with regular `load/store` instructions.



(a) STREAM Add reported performance between 1 and 16 hardware threads on the main socket.



(b) Memory access latency of a random walk across a linked list of increasing size.

Figure 3: Hardware characteristics of our Xeon Phi platform.

Unlike numerous related works (see Section VI) that consider migration from the point of view of the Operating System, our investigation is meant to help devise policies that would provide automatic migration in cooperation with the application's runtimes and using information on the characteristics of the target application (data access patterns, working set size, etc).

We begin by exploring the underlying existing data migration mechanisms that could form the basis of flexible migration policies to be developed later.

A. *Migration Mechanisms*

Chiefly, two sorts of mechanisms are available on modern Linux systems for migrating data between physical locations.

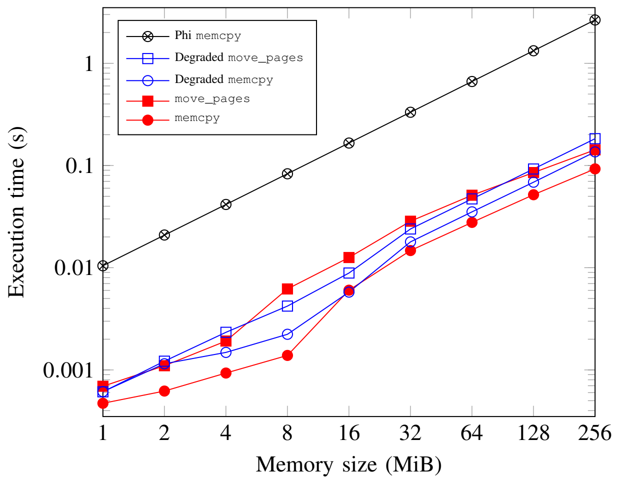
The first mechanism is the `move_pages` system call. It is designed specifically for migration purposes and has been used successfully to perform dynamic placement on NUMA machines [9]. The caller provides a set of 4 KiB pages to move and their desired new location (as a NUMA node). For each page to move, the kernel allocates a new physical page on the destination node, copies the data, updates the associated page table entry, and frees the source page. Since the Linux kernel uses a single thread for this operation, does not take advantage of huge pages, and performs the copying without using vectorized instructions, this mechanism might not be the fastest to migrate large amounts of data [9]. Nevertheless, `move_pages` is the only mechanism available to user applications to transparently alter the physical location of a page and is thus highly desirable in scenarios where changes to the application code are to be minimized.

The second mechanism available is `memcpy`. Being a user-space library call, it can copy only from one virtual memory location to another. When used between memory locations that have been locked to a specific physical node, however, it can still be used to perform data migration. It will be especially quick if the destination is prefaulted; moreover, most compilers and system libraries provide highly optimized versions of this operation, using vectorization for example, so a performance comparison with `move_pages` should be instructive. The major disadvantage of this migration mechanism is the fact that the address of the object changes when it is copied around, potentially requiring changes to the application code.

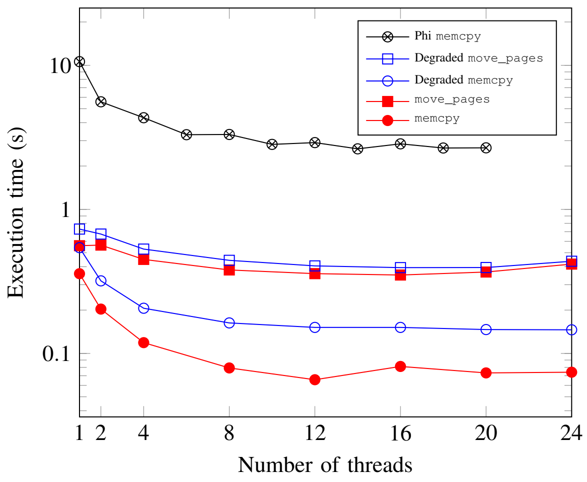
B. *Observations*

We now compare the performance of these two mechanisms to migrate data from the slow node to the fast one, for increasing amounts of data and varying numbers of threads performing the migration. These experiments have two goals: establishing which mechanism is faster for a given data volume and determining the number of threads required for best performance. Our benchmark is simple: starting with a prefaulted, memory-bound source array, perform in parallel either `move_pages` or `memcpy` of the entire array. This benchmark uses OpenMP to parallelize its work, splitting the memory to migrate into equal chunks, one per thread. Both methods are timed on the migration operation, after the source data has been prefaulted and bound to the slow node, although the `memcpy` result includes the cost of unmapping the source buffer and the destination is not prefaulted, in order to make the comparison with `move_pages` fairer. Figure 4a presents the execution time of these operations on one thread for different buffer sizes; Figure 4b focuses on the speedup obtained when using multiple threads for both operations, on all our platforms, for a fixed buffer size of 1 GiB.

The single-threaded results show that for small sizes `move_pages` and `memcpy` have the same performance but that memcpy is faster if the memory region is bigger than a few pages. The results stay within 50% of each other; we attribute the differences to the lack of vectorization in the kernel and different code paths taken in the kernel to update the page



(a) Log-log plot of the performance of serial migration.



(b) Log plot of the performance of parallel migration.

Figure 4: Performance of migration between the slow node and the fast node, from the main socket, on all platforms.

table of the process when performing an explicit page move operation vs reacting to page faults in `memcpy`. Increasing the number of threads used in the process makes the situation clearer. Although both operations perform better when using multiple threads, `memcpy` is consistently significantly faster—over 5 times for NUMA and 3 times for Degraded. Our results using multiple threads are also consistent with the platform measurements: about the same number of threads saturate the bandwidth of the slow node. We also note that `memcpy` is the only available facility to move data between the Xeon Phi memory and the host and that it exhibits a performance over an order of magnitude below that of the other platforms.

These results indicate clearly that `memcpy` is a much better option performance-wise. In fact, the difference in practice could be even greater than shown here. As indicated earlier, our `memcpy` results include the cost of faulting destination pages and releasing the source buffer, in order to make the comparison with `move_pages` fairer. An application could, however, allocate the buffers on the slow and the fast node only once and not release them until the its completion. The cost of page faults would be reduced to a single access for each page, making moving the data in and out of the fast memory node considerably faster. Consequently, in the remainder of this paper we focus exclusively on data migration using `memcpy`.

V. An Exploratory Benchmark

We now want to understand the ideal performance improvement that data migration can bring to an application. For such performance improvement to appear, the application must be able to make progress while the migration is taking place and to perform better when the data used is located on the fast node instead of the slow node.

Given the previous experiments and the expected features of deep-memory systems, we focus on applications with bandwidth-limited performance. Two configurations are of interest. The first involves giving enough threads to an application so that it saturates the fast memory bandwidth. This should leave some hardware threads available to perform data migration. The second configuration is the opposite: dedicating enough threads to data migration to saturate the bandwidth of the slow node and using the remaining threads for the application.

Since we want to evaluate these cases while also understanding the exact characteristics of the application that influence the effectiveness of data migration, we use a custom microbenchmark as our application. For simplicity, only two types of threads are used: data migration and worker threads. This benchmark proceeds in phases. In each phase, W worker threads execute a kernel accessing a private memory region located on the fast node. All workers handle the same amount of data and execute the same kernel. In parallel, C migration threads copy the data required by the workers for the next phase from the slow memory to the fast one. Our implementation uses two memory regions on each node type, to allow migration and work to proceed in parallel without interfering. We also make tunable the number of times a kernel is applied during a phase (I) and how much memory each thread is working on. The kernel used by worker threads is similar to STREAM Add: a 3 point stencil over 1D array.

We first study the performance of this microbenchmark without memory migration (i.e., all data stays in slow memory). We use a memory size larger than the last level cache of the platform (128 MiB for NUMA and Degraded, 16 MiB for Phi). When varying the number of threads, this benchmark should behave similarly to the STREAM benchmark. We will use this performance as a baseline for the remainder of our experiments. Figure 5 shows the results. We can see that increasing the number of threads improves the performance of the benchmark until the bandwidth of the slow node is saturated. Increasing the number of iterations of the kernel, however, increases significantly the run time of the benchmark but does not change its overall behavior. Note that we ran those experiments only up to the number of cores on the system for the Xeon Phi platform.

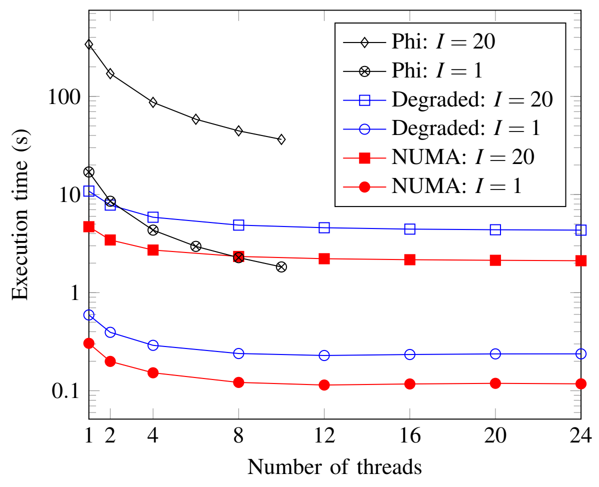


Figure 5: Impact of the level of parallelism on the performance of our microbenchmark, for various kernel iteration counts I, on all platforms.

Next, we study the performance of this microbenchmark when it is given enough cores for the worker threads to saturate the bandwidth of the fast node (e.g., 8 for NUMA and Degraded, 6 for Phi—refer to Figures 1a and 3a). The other 4 cores are used for data migration. We then vary the computation time of each phase. Our goal is to study the effects of data migration depending on the ability of the application to prefetch its data while the workers are still busy with the previous phase. Our microbenchmark is written so that if data migration takes too long, the benchmark will stall, waiting for the data required for the next phase to be ready.

Figure 6a displays results from this experiment, for a number of kernel iterations (I) between 1 and 20 on our platforms, using one thread per core on the main socket. We keep the memory size from the last experiment, to ensure that the workers are still impacted by the available bandwidth. We measure the execution time and report it in the form of the speedup of the benchmark compared with two baselines: when using the same number of worker threads W_b in the baseline (8 for NUMA and Degraded, 6 for Phi) and when using the number of worker threads in the baseline equal to the sum of worker and migration threads (12 for NUMA and Degraded, 10 for Phi). We can see that migration can significantly improve the performance of our benchmark when compared with either baseline. Thus, a better performance can be obtained by removing worker threads from the application and using them to transfer data around. As long as workers are kept busy, the benchmark will perform in a way comparable to running directly from the fast memory node. Starting from a low number of iterations, the worker threads are given more work in each phase, leaving time for the migration to complete. The resulting performance increase ends for NUMA and Degraded when the workers no longer stall waiting for the migration. Results from Phi show the same trend but do not level off even for 20 kernel iterations since the migration cost is much higher.

We now study the performance of this microbenchmark when enough cores are instead dedicated to data migration to saturate the bandwidth of the slow node (8 for NUMA and Degraded, 6 for Phi). Again, we vary the number of iterations of the kernel for each worker. Figure 6b presents the results obtained between 1 and 20 iterations; we use the same two baselines for the speedup as in the previous experiment. The difference in the number of worker threads between the baselines is greater than before, leading to a wider gap in the plot between each pair of experiments. Prioritizing the number of migration threads leads to a slightly worse speedup than before for NUMA and Degraded because it does not leave enough worker threads to process the deluge of input data. The same is not the case with the Phi runs, however, where increasing the number of migration threads reduces the memory bandwidth bottleneck.

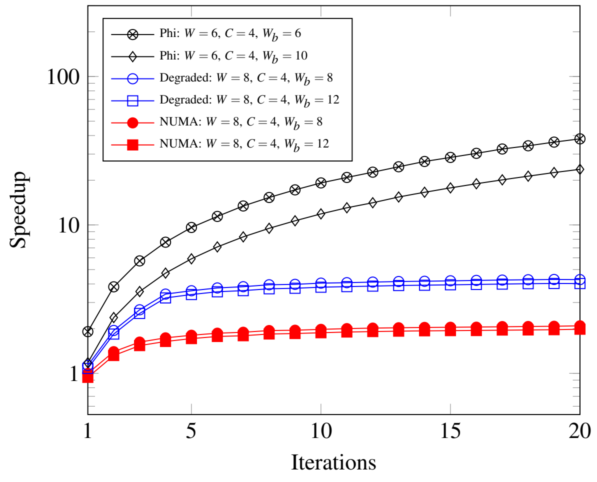
Overall, these experiments demonstrate that application-driven data migration has the potential to significantly improve the performance on future deep-memory platforms. Perhaps unsurprisingly, the speedup obtained (around 2 times for NUMA, 4 times for Degraded, and up to 80 times for Phi) is closely related to the bandwidth gap between fast and slow memory for each platform, as measured in Section III. Still, the ability to close this gap on NUMA and Degraded strongly indicates the viability of this approach, even if it is done using a dedicated microbenchmark.

VI. Related Work

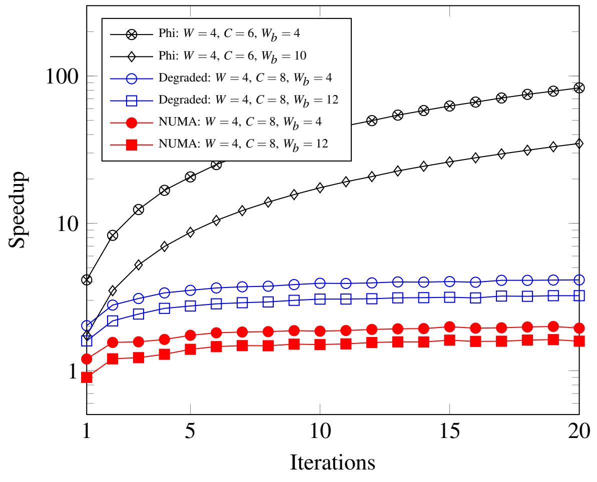
The general issue of migrating hot data to faster but limited-size memory is an old problem. The constraints are diversified, however, and they depend mostly on how the computing device interacts with each kind of memory.

Increasingly, flash-based block devices are used not as an extension to hard drives [10]–[12], but as a memory level that sits between DRAM and the hard drive. SSDs have been used as DRAM extensions for metagenomics [13], graph traversal [14], stencil computation [15], and big data computing [16]. More elaborate approaches have merged NAND flash and PCM into a unified, single-memory unit that requires the migration of hot data from NAND to PCM to be addressed by the CPU [17]. The extension of the memory hierarchy with flash-based devices calls for migration mechanisms that are different from the ones considered by our current work, as neither `move_pages` nor `memcpy` can be used with a block device.

In high-performance computing, the most commonly available deep-memory architectures to date are made of GPUs accessing their own on-package GDDR as faster memory and the host DRAM as slower memory [18]. While recent GPUs can seamlessly address the host DRAM, the preferred approach remains the asynchronous prefetching of the data because of PCIe bandwidth constraints and the need for pinning host memory for GPU access [19]–[21]. With the



(a) Enough worker threads to saturate the fast node.



(b) Enough migration threads to saturate the slow node.

Figure 6: Impact of the number of kernel iterations on the speedup obtained from data migration.

Intel Xeon Phi code-named Knights Corner, the host can use the coprocessor memory as a slower memory via the SCIF interface [22]. Other authors used the host memory as the slower kind for computations on Knights Corner [23]. Both GPUs and Knights Corner operate as accelerators with respect to their host DRAM. In comparison, our work explores regular processors.

An important difference exists between all the aforementioned hierarchical memory systems, along with their associated needs for migration, and the MCDRAM-DRAM usage modes in the upcoming Knights Landing. With the latter, program data is natively addressable from both memories and in a symmetric way with respect to the interface. A two-level natively addressable deep-memory architecture has been emulated [24] to behave like the Knights Landing in that respect. Others [25] have studied a hierarchical memory that can also be used as cache or addressed as a faster DRAM on chip, just like the Knights Landing.

Page migration using NUMA nodes has been explored. Researchers have considered applications that exhibit a repetitive pattern of memory access and perform the migrations based on previous iterations of similar computations [26]. With programs for which no assumption can be made, a sampling-based approach is used where pages with excessive remote references are migrated to nodes closer to the accessing cores. Others [9] used both `move_pages` and `memcpy` in their approach to memory migration. A new `madvise` flag has also been implemented to change the access rights of pages meant to be migrated, forcing the kernel to trigger a migration upon fault. Where the focus of the previous work was on regular NUMA architectures, we use some of the same mechanisms to explore emerging deep memory hierarchies, which feature different performance trade-offs and, critically, significant capacity differences between memory hierarchies, requiring different migration policies.

VII. Conclusion

Performance of HPC applications on future computing platforms with deep memory hierarchies will depend on the ability to choose carefully which data structures must be located where and when. In this context, we studied here the possibility for applications to dynamically migrate data between the levels of such deep-memory architectures. Because these architectures are not yet available, we used a variety of approximation platforms, including NUMA systems and accelerators, with different latency and bandwith characteristics, to study the usefulness of user-level data migration.

Using state-of-the-art methods, we measured the characteristics of those platforms, as well as the performance of user-level data migration mechanisms available on Linux systems. We found that with some additional software-induced degradation, a regular NUMA system can reasonably approximate the bandwidth of a deep-memory architecture. Our comparison of the performance of `move_pages` and `memcpy` indicates that the latter, while more challenging to use for applications, shows performance improvement by an integer factor in multithreaded scenarios.

Given these measurements, we then designed a set of experiments to evaluate the best performance improvements an application can expect from data migration. The resulting experiments show that a significant speedup—approaching the measured difference in the memory bandwidth—can be obtained by reducing the number of worker threads in the application and using them for data migration instead. To obtain such improvements, the workload needs to meet a number of requirements in terms of the memory access pattern and working set size characteristics and must be insensitive to an occasional stall. The ratio of worker threads to migration threads needs to be chosen carefully to maximize the migration bandwidth while ensuring that the application is not starved of compute resources.

Overall, we believe this study provides important information about the relation between the number of threads, the amount of work in the application, and the characteristics of the hardware platform. This should prove valuable to efficiently use the soon-to-be-released Knights Landing architecture, for example.

In terms of future work, we are interested in extending this study to benchmarks mimicking exascale applications. We will also use hardware performance counters and tracing to obtain a deeper understanding of how application kernel characteristics impact the performance of data migration schemes. We are eagerly awaiting deep-memory architectures to become available so that we can continue our work on real hardware. We want to investigate whether the performance gap we have identified between the kernel and user-level data migration mechanisms can be narrowed down.

The early experiments we presented required performing data migration explicitly in the application. It will be highly valuable to design facilities to automate, or at the east help with, the data migration process. In particular, we are interested in facilities for applications or compilers to annotate data structures with memory usage information and the use of this information in automatic data placement and migration policies at the runtime level.

Acknowledgments

Results presented in this paper were obtained using the Chameleon testbed supported by the National Science Foundation. This material was based upon work supported by the U.S. Department of Energy, Office of Science, Advanced Scientific Computer Research, under Contract DE-AC02-06CH11357.

References

[1] J. Dongarra, P. Beckman *et al.*, "The international exascale software project roadmap," *Int. J. High Perform. Comput. Appl.*, vol. 25, no. 1, pp. 3–60, Feb. 2011. [Online]. Available: http://dx.doi.org/10.1177/1094342010391989

[2] E. Gardner, "What public disclosures has Intel made about Knights Landing?" https://software.intel.com/en-us/articles/what-disclosures-has-intel-made-about-knights-landing.

[3] A. Sodani, "Knights Landing (KNL): 2nd generation Intel® Xeon Phi processor," in *Symposium on High Performance Chips (HotChips)*, Aug. 2015.

[4] I. Buck, "Nvidia's next-gen Pascal GPU architecture to provide 10x speedup for deep learning apps," https://blogs.nvidia.com/blog/2015/03/17/pascal/.

[5] AMD, "High bandwidth memory: Reinventing memory technology," http://www.amd.com/en-us/innovations/software-technologies/hbm.

[6] J. D. McCalpin, "Memory bandwidth and machine balance in current high performance computers," *IEEE Computer Society Technical Committee on Computer Architecture Newsletter*, pp. 19–25, Dec. 1995.

[7] U. Drepper, "What every programmer should know about memory," http://people.redhat.com/drepper/cpumemory.pdf, 2007.

[8] S. Perarnau, M. Tchiboukdjian, and G. Huard, "Controlling cache utilization of HPC applications," in *International Conference on Supercomputing (ICS)*, 2011.

[9] B. Goglin and N. Furmento, "Enabling high-performance memory migration for multithreaded applications on Linux," in *IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, May 2009, pp. 1–9.

[10] S. Ko, S. Jun, Y. Ryu, O. Kwon, and K. Koh, "A new Linux swap system for flash memory storage devices," in *International Conference on Computational Sciences and Its Applications (ICCSA)*, Jun. 2008, pp. 151–156.

[11] M. Lin, S. Chen, G. Lv, and Z. Zhou, "Optimised Linux swap system for flash memory," *Electronics Letters*, vol. 47, no. 11, pp. 641–642, May 2011.

[12] K. Liu, X. Zhang, K. Davis, and S. Jiang, "Synergistic coupling of SSD and hard disk for QoS-aware virtual memory," in *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Apr. 2013, pp. 24–33.

[13] B. Van Essen, H. Hsieh, S. Ames, and M. Gokhale, "DI-MMAP: A high performance memory-map runtime for data-intensive applications," in *Proc. Int. Workshop on Data-Intensive Scalable Computing Systems (DISCS 2012)*, Nov. 2012, pp. 731–735.

[14] B. Van Essen, H. Hsieh, S. Ames, R. Pearce, and M. Gokhale, "DI-MMAP—A scalable memory-map runtime for out-of-core data-intensive applications," *Cluster Computing*, vol. 18, no. 1, pp. 15–28, 2015.

[15] H. Midorikawa and H. Tan, "Locality-aware stencil computations using flash SSDs as main memory extension," in *15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, May 2015, pp. 1163–1168.

[16] J. Lee, S. Park, M. Ryu, and S. Kang, "Performance evaluation of the SSD-based swap system for big data processing," in *13th IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, Sep. 2014, pp. 673–680.

[17] K. Park, S. K. Yoon, and S. D. Kim, "Selective data buffering module for unified hybrid storage system," in *14th IEEE/ACIS International Conference on Computer and Information Science (ICIS)*, Jun. 2015, pp. 173–178.

[18] K. Nakano, "The hierarchical memory machine model for GPUs," in *IPDPSW*, May 2013, pp. 591–600.

[19] C. Statz, M. Mutze, S. Hegler, and D. Plettemeier, "Hybrid CPU-GPU computation of adjoint derivatives in time domain," in *Computational Electromagnetics Workshop (CEM)*, Aug. 2013, pp. 32–33.

[20] N. Garcia and R. C. Olmos, "GPU-accelerated poincaré; map method for harmonic-oriented analyses of power systems," in *IEEE Power and Energy Society General Meeting (PES)*, Jul. 2013, pp. 1–5.

[21] P. Alfaro, P. Igounet, and P. Ezzatti, "A study on the implementation of tridiagonal systems solvers using a GPU," in *30th International Conference of the Chilean Computer Science Society (SCCC)*, Nov. 2011, pp. 219–227.

[22] S. Potluri, D. Bureddy, K. Hamidouche, A. Venkatesh, K. Kandalla, H. Subramoni, and D. K. Panda, "MVAPICH-PRISM: A proxy-based communication framework using InfiniBand and SCIF for Intel MIC clusters," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, Nov. 2013, pp. 1–11.

[23] B. Gerofi, A. Shimada, A. Hori, and Y. Ishikawa, "Partially separated page tables for efficient operating system assisted hierarchical memory management on heterogeneous architectures," in *13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, May 2013, pp. 360–368.

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

[25] X. Dong, Y. Xie, N. Muralimanohar, and N. P. Jouppi, "Simple but effective heterogeneous main memory with on-chip memory controller support," in *SC*, Nov. 2010, pp. 1–11.

[26] D. S. Nikolopoulos, T. S. Papatheodorou, C. D. Polychronopoulos, J. Labarta, and E. Ayguade, "User-level dynamic page migration for multiprogrammed shared-memory multiprocessors," in *International Conference on Parallel Processing (ICPP)*, Aug. 2000, pp. 95–103.