Phase-Based Data Placement Scheme for Heterogeneous Memory Systems

Mohammad Laghari
Computer Science and Engineering
Koç University, Istanbul, Turkey
mlaghari16@ku.edu.tr

Najeeb Ahmad
Computer Science and Engineering
Koç University, Istanbul, Turkey
nahmad16@ku.edu.tr

Didem Unat
Computer Science and Engineering
Koç University, Istanbul, Turkey
dunat@ku.edu.tr

***Abstract*—Heterogeneous memory systems are equipped with two or more types of memories, which work in tandem to complement the capabilities of each other. The multiple memories can vary in latency, bandwidth and capacity characteristics across systems and they come in various configurations that can be managed by the programmer. This introduces an added programming complexity for the programmer. In this paper, we present a dynamic phase-based data placement scheme to assist the programmer in making decisions about program object allocations. We devise a cost model to assess the benefit of having an object in one type of memory over the other and apply the cost model at every application phase to capture the dynamic behaviour of an application. Our cost model takes into account the reference counts of objects and incurred transfer overhead when making a suggestion. In addition, objects can be transferred across memories asynchronously between phases to mask some of the transfer overhead. We test our cost model with a diverse set of applications from NAS Parallel and Rodinia benchmarks and perform experiments on Intel KNL, which is equipped with a high bandwidth memory (MCDRAM) and a high capacity memory (DDR). Our dynamic phase-based data placement performs better than initial placement and achieves comparable or better performance than cache mode of MCDRAM.**

***Index Terms*—high bandwidth memory, object placement, MCDRAM, DRAM, KNL**

I. Introduction

In recent years, we have observed a rise in the number of systems with diverse types of memories to counter the engineering limits of DDR memory technologies [1]. For instance, a typical DDR4 can only transfer data at a rate of 88GB/s to the CPU [12] and at this rate a compute unit cannot be utilized at its full capacity, leading to wasted clock cycles and low flops rate. As a result, heterogeneous memory systems equipped with multiple memory types each with distinct characteristics have emerged to overcome the bandwidth limitations. Some of the high-bandwidth memory (HBM) technologies are high bandwidth memory standard by JEDEC [11], hybrid memory cube (HMC) by Micron [16], or a technology like WideIO [10]. Intel Knights Landing (KNL) chip comes with an HBM called Multi-Channel DRAM (MCDRAM), which boasts 450GB/s memory bandwidth as compared to its slower DRAM (88GB/s). The increase in bandwidth, however, comes at the cost of higher access latency and low capacity. To compensate for these shortcomings, HBMs are typically augmented with high capacity memories which generally have a lower access latency.

Having multiple memories introduces the need for data management. In this regard, programmers have the option to explore various configurations. These configurations can be broadly categorized as 1) *hardware-managed*, therefore transparent to the programmer, or 2) *software-managed* through OS or application code. In hardware-based strategies, HBM is considered as a last level cache and hardware handles the data admission and eviction. In software-based management, heterogeneous memory systems allow programmers to allocate application data on either memory depending on application characteristics, potentially improving the overall application performance. For HBM management through software, previous work focuses on 1) *OS-based approaches* and 2) *Application-driven allocations*. Even though OS-based approaches do not require any modifications at the application and free the programmer from concerns about object allocations, they require changes in the OS and operate on the page granularity rather than data objects.

In application-driven allocations, objects are explicitly partitioned between high bandwidth memory and high capacity memory by the programmer [3] or frameworks assisting the programmer [13]. Placement of objects can be performed statically at the beginning of the program, or dynamically as the program executes based on the phases of the application. In our prior work [13], we proposed an initial object placement algorithm and observed a considerable improvement in execution time. Our placement algorithm is based on 0-1 Knapsack and takes into account the object sizes and their memory reference counts to suggest an initial allocation scheme for the entire application. This approach, however, fails to capture the object activity at any particular time as the program executes. In addition, the decision of which objects to place where depends on numerous factors. For example, HBM in Intel KNL is favorable to bandwidth-bound applications and can cause performance degradation to latency-bound applications [12] [13]. Other factors such as whether application performs strided accesses or has write-intensive workload require more complex decision-making for applications with large memory footprint. In this paper, we propose a new dynamic object allocation scheme which performs on the fly object admission and eviction, to and from the HBM.

We identify the attributes which can affect the choice for objects being in a particular memory during program execution. These attributes can be hardware-related features such as bandwidth of each memory type, transfer bandwidth between memories, or software-related such as memory access pattern, object reference count, object size etc. We incorporate these attributes into a cost model, which estimates the benefit of having an object in one memory over the other. Using this cost model for each application phase, we apply Knapsack algorithm and transparently move the objects between two memories, dynamically adapting the application behavior. We demonstrate our framework on high bandwidth memory available in Intel KNL architecture with several applications.

The remainder of the paper is organized as follows. Section II discusses the architecture of heterogeneous memory systems. Section III describes the cost model we devise. Section IV presents the working of our tool. Section V evaluates the cost model and tool on several applications. Section VI and VII discuss the related work and concludes the paper.

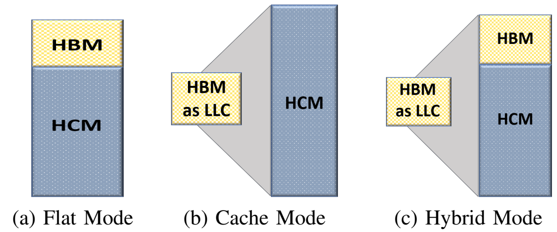
II. System Architecture

We assume that a heterogeneous memory system is equipped with multiple memories which are controlled by the same processing unit. These distinct memories differ from one another in their characteristics and together they complement one another. An HBM can act as a cache for the high capacity memory (HCM). Whereas the high capacity memory, typically larger in size, can act as a data bank for HBM. In our work we aim to capture all systems which are set in the aforementioned memory configuration. It is important to note that the HBM is not always advantageous; can lead to higher power consumption or longer memory latencies.

Figure 1 shows the possible configurations in which a typical heterogeneous memory system can operate. HBM can either **a)** be set as a separately addressable memory or **b)** act as a hardware-managed last level cache. In addition to these two settings, HBM **c)** can be configured as a hybrid of both such that part of it acts as a cache while the rest of it can be explicitly managed through software. One of the such heterogeneous systems available today is Intel KNL chip, which will be used in our benchmarks and experiments. Another example can be non-volatile memory-based heterogeneous memory system, where DRAM is used as a software or hardware-managed cache for NVRAM [26], [14], [24].

III. Cost Model

We devise a cost model, which captures the attributes of a system and takes into account the application level details to produce a score for each object to be placed on a desired memory. This cost model forms the objective function of the 0-1 Knapsack algorithm which produces an allocation scheme for an application being run on an heterogeneous memory system. The knapsack is considered as HBM and is associated with a maximum weight that is the total capacity of HBM. The data objects are considered to be the individual items, which can be carried by the knapsack. Each item has a weight and a value, which in our case are the size of the object and its score, assigned by the cost model. The objective is to place the most valuable objects in HBM without overloading it.



(a) Flat Mode (b) Cache Mode (c) Hybrid Mode

Fig. 1: Memory modes of a heterogeneous system. a) HBM and HCM are two separately addressable memories b) HBM is managed by hardware. Automatic caching is performed. c) HBM is partially available as a separately addressable memory and partly as a hardware managed last level cache (LLC).

A. Cost Model for Initial Placement

The initial placement approach allocates the objects at the start of an application and the placement does not change throughout the execution. Objects are accessed from the respective memories they are allocated to. Since this scheme is static, the cost model does not consider the lifecycle of an object. This trade off is compensated by the zero overhead of object transfers across memories, yielding in a performance benefit for certain application workloads.

The initial placement variant of the cost model only considers the overall load and store accesses to memory of an object. A memory can prioritize writes over reads or vice versa and this feature could vary from one memory architecture to another. Therefore we add α and β coefficients for the writes and reads, respectively for a setup where loads or stores are to be favored over the other. For example, on Intel KNL write-sensitive objects should be prioritized to be placed on MCDRAM [13]. The following equation calculates a score for each object m which forms the basis of our cost model's objective function in initial placement. The Knapsack algorithm suggests a candidate object list to the programmer based on these scores.

$$Score_m = \alpha * Writes_m + \beta * Reads_m \quad (1)$$

B. Cost Model for Phase-Based Placement

Phase-based cost model improves the initial placement by taking the lifecycle of an object into account during the program execution. Applications can have objects which are heavily accessed in a particular phase of their runtime. Having such objects in HBM throughout the program execution cannot yield full potential of heterogeneous memory systems. Since the different memories can be controlled through software by the programmer, it is more advantageous to evict unused objects and allocate objects which are going to be used in the next phases. There are several ways to identify the application phases. One of the easiest way is to divide the application into phases based on loops. Since a loop accesses the same objects

in a repeating fashion, the benefit of bringing those objects into HBM can compensate the transfer overhead.

We also identify if certain objects in a phase tend to be latency-bound or bandwidth-bound by analyzing whether accesses are indirect or streaming. The former being a candidate for latency-bound object while the latter can be categorized as a bandwidth-bound object. We use this information in our cost model to better decide which object would yield a higher performance benefit in a particular kind of memory. For instance, having a latency-bound object in MCDRAM of Intel KNL can yield a degraded application performance. Phase-based cost model uses the following attributes about the application and the underlying machine: **a)** Remaining load and store count of an object, **b)** Transfer overhead caused by data movement, **c)** Read and write bandwidths of different memories, **d)** Strided access behavior and latency-boundness.

Firstly, the phase-based cost model discards the memory access information of objects from the previous phases for the current phase. This helps the cost model to decide whether an object is losing its advantage over its lifetime during execution. If the number of accesses for a particular object decreases, it becomes less advantageous to keep that object in HBM. Secondly, the eviction and admission of objects from and to HBM introduces an overhead. This overhead is due to synchronous transfer of objects between memories. To overlap the transfers with execution of the previous phase, we spare some threads for the transfer mechanism. Note that this means slightly reduced thread count for the phase computation. The third attribute relates to the different read and write bandwidth performance of different types of memories. The fourth attribute considers indirect or strided access behavior of an application. Our cost model prioritizes the objects which are accessed in a contiguous fashion to be placed on HBM in Intel KNL. Note that this behavior can vary between systems. In such cases, our cost model can make decisions accordingly.

The four attributes discussed above are translated into a final cost model, which is used as the objective function in the 0-1 Knapsack algorithm. An object can be accessed from either HBM or HCM in a particular phase. Based on the cost model, we transfer objects to the appropriate memory. An object is favorable to reside on one type of memory if its access time is reduced when it is placed in that memory. We calculate the access time of an object based on its score, element type and the stream bandwidth of the particular memory its going to be accessed from. We compute the access time metric only to estimate the benefit of having the object on one particular memory. By no means, this metric is the time required to load or store that object. For an object m, the access time is:

$$AccessTime_m = \frac{Score_m * ElementType_m}{Bandwidth_{stream}} \quad (2)$$

We calculate the score in the same fashion as for the initial placement strategy. However, there is one notable change in score calculation. For initial placement we use the global read and write references to calculate the score. In phase-based placement, we use the remaining reads and writes for that object in the application. This ensures that the score of the object decreases if the object has fewer memory accesses in the later part of the application. The score function for phase-based placement translates to:

$$Score_m = \alpha * RemainingWrites_m + \beta * RemainingReads_m \quad (3)$$

We observe in our experiments that memory access bandwidth is reduced in the presence of strided or indirect accesses as expected. This access pattern can change the access time of an object and in turn can influence the decision of placing an object in HBM. To incorporate this behavior, we use the $Bandwidth_{strided}$ instead of $Bandwidth_{stream}$.

Through $Access\ Time_m$, the algorithm decides whether the object is used fairly enough to be kept in HBM. Since objects are moved between phases, it introduces the added overhead of transfer between memories. In worst case, every phase can have a different working set, resulting in too much transfer overhead. To take this overhead into account, we calculate the transfer cost incurred by moving an object. Following equation shows the transfer cost calculation performed to determine the overhead of transferring an object from one memory to another. The copy bandwidth for each memory type can vary. The transfer cost takes into account this differing feature of the memory as well.

$$TransferCost_m = \frac{ObjectSize_m}{Bandwidth_{copy}} \quad (4)$$

We use access time and transfer cost of an object m to build our objective function ($ObjFunc$), which is in turn used in 0-1 Knapsack algorithm to decide which objects should reside on HBM. We add the eviction cost of object n in case an object needs to be evicted from HBM to HCM to open up space for object m. This overhead is very similar to transfer cost except that it uses the copy bandwidth from HBM to HCM, which can be different than the one from HCM to HBM [13].

$$ObjFunc_m = AccessTime_m - TransferCost_m - EvictionCost_n \quad (5)$$

The 0-1 Knapsack algorithm uses this objective function for the phase objects at each phase and tries to maximize the function. As an output, the algorithm lists the objects that are the best candidates to reside on HBM for a particular phase.

IV. Implementation

We present our work in the form of a tool which performs object allocation and asynchronous transfers of data between different types of memories. The working of our tool is two-tiered. In the first tier, we profile the application using two different sampling based profilers. In the next tier, we insert API calls to the tool in the application so that the tool can run the object selection algorithm and perform object transfers between the memories. In the following sections we explain these steps.

A. First Tier: Profiling

Our cost model requires object-level information to be collected on the application to devise an allocation strategy. In particular, it requires the **a)** program-level load and store

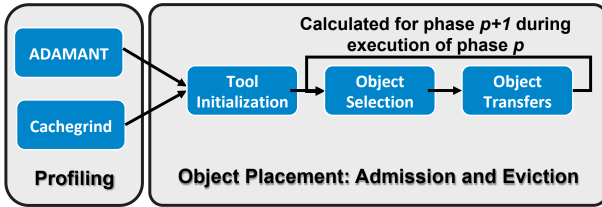


Fig. 2: Interaction of ADAMANT and Cachegrind with our tool. ADAMANT provides the object reference counts whereas Cachegrind determines code blocks used as phases.

counts, **b)** phase-level load and store counts, and **c)** the size of each object. We assess various tools which use static code analysis or binary instrumentation techniques to intercept object level events [4] [8] [22]. Collecting object-level information through static code analysis [23] is very fast and incurs minimal overhead but it has limited capabilities as it cannot capture indirect memory accesses or conditional executions. Binary instrumentation techniques are known to be accurate but slow, incurring large overheads. To strike a balance between accuracy and speed, we leverage ADAMANT [7] and Cachegrind [15] to gather object-level information along with the phase information.

1) ADAMANT: is a sampling based address tracing tool that provides object-level load and store information at the program level without any significant overhead. ADAMANT can intercept with memory allocation calls and distinguish between statically and dynamically allocated objects. In order to store the object access information, our tool maintains a hashmap of objects and their load and store counts. For static objects, the mechanism of recording loads and store is straightforward – by matching the object names in the application code. For dynamic allocations, we compare the memory start addresses given by ADAMANT to the memory addresses we obtain from the application code. This allows us to keep track of each object and its remaining memory accesses throughout the execution.

2) Cachegrind: is one of the tools in the Valgrind instrumentation framework [15], which we use for extracting phase information from an application. Cachegrind simulates the application behavior with the memory and caches in a computer. It provides us with last level cache misses and can distinguish misses as load or store misses. We interpret LLC misses as accesses to memory and combine these statistics with those found by ADAMANT for a better estimate of an object's memory traffic. In addition, Cachegrind allows us to virtually divide the application into phases. The phases form the boundaries at which we run our object placement algorithm to select objects for the upcoming phase. Cachegrind gives memory access information at the granularity of a statement from the program. We sum the memory accesses made by the statements inside a code block and determine if that particular code block qualifies as a phase, incurring significant amount of memory traffic. Since a phase contains multiple objects, our algorithm can make a selection of objects to be kept on HBM.

ADAMANT and Cachegrind are two separate tools, therefore they require separate profiling runs to gather application information at a target platform. However, once the information is collected, there is no need to perform profiling again.

B. Second Tier: Object Placement

After we have gathered object-level statistics and phase information of an application, we insert API calls to our tool into the application. Our API can perform allocation and deallocation of objects seamlessly as the programmer would not need to evict and admit objects during the runtime. At every phase our tool takes care of all object level asynchronous migrations from one memory to another if our object selection algorithm decides to do so.

1) Initialization: At the beginning of the target application, we initialize our tool with the previously profiled data. This includes the output from ADAMANT and Cachegrind, namely the object level statistics and the phase information. After initialization, our tool stores all the objects-level information, which includes each object's size and reference count to memory. To reduce the cost of allocation and deallocation for every object to be placed in HBM, we divide the HBM into buffers. Each buffer is configured such that it can host the biggest object in the application. However, if one of the objects is significantly larger than the rest, we allocate a separate buffer exclusively for that object. Along with configurable buffer size, during initialization, we can also tune the number of buffers allocated. For each buffer, we maintain a modified bit to keep track whether the object stored in the buffer is modified or not. When the object selected for eviction to accommodate another object is not modified, it can simply be overwritten, incurring no eviction cost.

2) Object Selection: We currently require the programmer to add an API call before every phase of the application which is determined using Cachegrind. The purpose of this API call is to run the object placement algorithm at each of these calls, which is equipped with the objective function discussed in the cost model. Since our tool has acquired the phase-level information previously, it knows which objects are being accessed in a particular phase. Based on this information, and the output from the placement algorithm, our tool transparently transfers objects to and from the HBM. As mentioned earlier, the task of moving objects from one memory to another comes with an overhead. Our placement algorithm, with the help of cost model, does not move objects which have a movement cost greater than the benefit gained from keeping objects in HBM. Lastly, during the execution of the last phase, the tool does not perform any transfers.

3) Object Eviction: After our placement algorithm suggests objects to be placed on HBM, the tool checks for the object currently residing on HBM. If either of the buffers in HBM contain objects from the selection, our tool does not modify those buffers. For the remaining buffers which does not contain matching objects, we need to evict these objects and add new objects to HBM, or we can simply overwrite the objects currently residing on HBM. The modified bit for each buffer

determines whether the object requires transfer or not. If so, the tool transfers the content from that buffer back to HCM prior to transferring the new object in that particular buffer.

4) Asynchronous Transfers: We make use of parallelism during the transfer and object placement selection process. The runtime uses two threads to run the object placement algorithm and perform required transfers to and from both memory types. This job is carried asynchronously during application execution. Therefore, by the time the program reaches the next phase, the objects used in that particular phase are in the appropriate memories. The asynchronous transfers minimize the execution of the application as compared to the scenario when these calculations and transfers are conducted synchronously. Asynchrony is not achieved in all cases. If the application is executing phase p, our tool will calculate the placement strategy for phase $p+1$. However, in some cases, our tool will suggest objects to be removed from HBM for phase $p+1$ which are currently being used in phase p from HBM. In such cases the tool will wait for phase p to complete its execution and then proceed with the transfer of object to or from HBM.

V. Evaluation

In this section, we evaluate the performance of our phase-based object placement tool against various placement policies on Intel Knights Landing (KNL). The Intel KNL machine is equipped with a high bandwidth memory which is known as Multi-Channel DRAM, or MCDRAM for short. Intel KNL has three memory modes. A *Flat Mode* where the HBM can be accessed as a separately addressable memory, a *Cache Mode* where the HBM acts as a hardware managed last level cache and a *Hybrid Mode* in which the HBM can act as a combination of the two aforementioned modes. We compare our tool under various memory configurations supported in KNL with 64 cores. Table I summarizes these configurations and the labeling convention that we will use in results.

1) **All-DDR:** All objects are allocated in DDR. MCDRAM is not used.
2) **All-MCDRAM:** All objects are allocated in MCDRAM. DDR is not used.
3) **Hardware Cache:** We make all allocations to DDR and let the hardware cache objects into MCDRAM.
4) **Initial Placement w/o Cache:** Only initial placement is performed based on the program-level object references. HBM is set to 4GB in the algorithm. No hardware caching is enabled.
5) **Dynamic Placement w/o Cache:** uses phase-based object placement and its cost model, and performs asynchronous transfers between HBM and HCM. HBM is set to 4GB. No hardware caching is enabled.
6) **Dynamic Placement w/ Cache:** Similar to the previous configuration, however we augment the allocatable HBM with last level cache by setting aside 4GB of MCDRAM for hardware caching.

Table II shows the applications from Rodinia and NAS Parallel benchmark suites we used for evaluation. The table also shows the memory footprint of each application and the phases which were identified. All our experiments are conducted with 64 OpenMP threads. Thread affinity is set to scatter for all application execution. To rule out any anomalies with the gathered results, we take an average of 5 runs for each experiment. We decided to report speedups instead of execution times because the scale of running time for each application is drastically different. Dynamic placement results use asynchronous transfers if possible unless stated otherwise.

TABLE I: Intel KNL configurations used to evaluate our tool

Label	HBM	DDR	Cache (L3)	Boot Mode
All-DDR	—	96GB	—	Flat
All-MCDRAM	16GB	—	—	Flat
Hardware Cache	—	96GB	16GB	Cache
Initial Placement (w/o Cache)	4GB	96GB	—	Flat
Dynamic Placement (w/o Cache)	4GB	96GB	—	Flat
Dynamic Placement (w/ Cache)	4GB	96GB	4GB	Hybrid

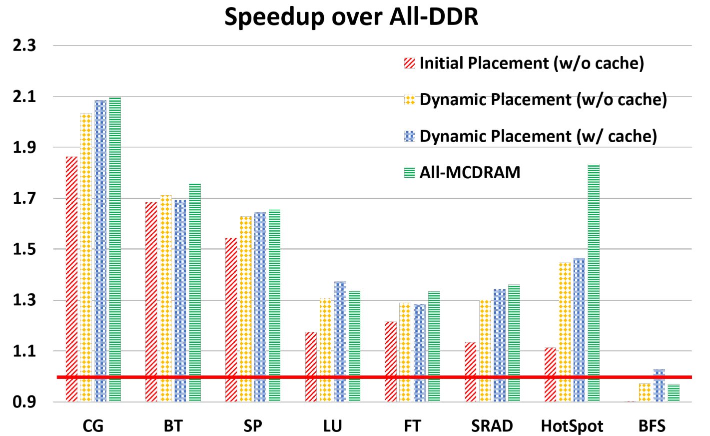


Fig. 3: Speedup achieved by the object placement conducted by our tool against All-DDR mode. Red line shows the baseline. All values below 1 indicate degraded performance.

A. Comparison against All-DDR and All-MCDRAM

Figure 3 shows the speedup achieved when the program is executed with the placement strategy suggested by our tool against the default placement setting i.e. when all the data is housed in the DDR memory, referred as ALL-DDR. In this figure, we also compare our performance against when all the data is allocated in the fast $MCDRAM$, referred as ALL-MCDRAM. We observe and confirm our hypothesis that dynamic placement consistently performs better than the initial placement. This is mainly due to the object eviction and admission protocol conducted by our tool. In most of the cases we observe that our tool performs nearly as good as the best case scenario i.e. when all the objects are allocated in the $MCDRAM$.

TABLE II: Evaluated Applications

Applications	Description	# of Objects	Footprint (GB)	# of Phases
CG [2]	Conjugate Gradient solves unstructured sparse linear systems	13	5.23	17
BT [2]	Block tri-diagonal solver	11	4.49	31
FT [2]	Performs discrete fast Fourier Transform	5	4.50	28
LU [2]	Lower-Upper Gauss-Seidel solver	14	4.53	30
SP [2]	Scalar Penta-diagonal solver	10	7.76	28
SRAD [5]	Diffusion method for ultrasonic and radar imaging applications	7	5.55	4
HotSpot [5]	Thermal simulation tool used for processor temperature estimation	3	6.44	11
BFS [5]	Breadth First Search graph traversal algorithm	6	9.75	5

B. Comparison against Hardware Cache

$MCDRAM$ in Intel KNL can be set as a last level cache. In this mode, the hardware performs caching from DDR to all levels of cache. The application is executed without any changes made to it, which is the easiest mode for the programmer. However, this might not always result in the best performance. Figure 4 shows the speedup achieved by the placement conducted by our tool against hardware caching. We observe that majority of the applications perform well over hardware caching. In all the cases, dynamic placement achieves better performance than automatic hardware caching. For CG, FT and $SRAD$, the initial placement fails to perform better than hardware caching. This is mainly because the initial placement only suggests object allocation based on the global load and store counts and does not dynamically adapt the application behavior. Whereas dynamic placement considers the loads and stores for each phase separately and performs object movement across memories when necessary.

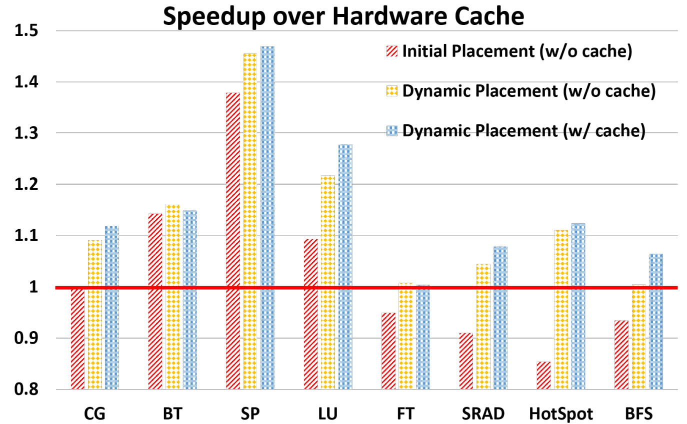


Fig. 4: Speedup achieved by our tool against MCDRAM acting as LLC. Red line shows the baseline. All values below 1 indicate degraded performance.

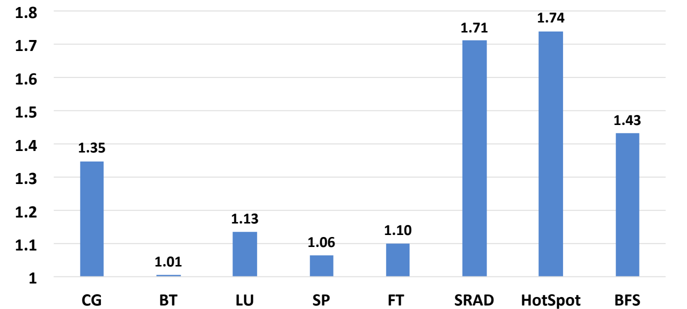


Fig. 5: Speedup achieved by asynchronous transfers over synchronous transfers using dynamic placement strategy.

C. Analyzing Transfers between Phases

In this section, we study the benefit of asynchronous transfers and analyze some statistics about object movement between phases. Our tool hides the overhead of object selection and object transfer for phase $p + 1$ by overlapping this calculation and data movement while phase p is under execution. Figure 5 shows the speedup achieved over the placement when transfers are performed synchronously. We perform the same experiments for each application by conducting the object selection and movements synchronously i.e. when object selection and object transfers are done in a serial fashion without sparing any threads for these operations. The speedup achieved confirms that asynchronous transfers have a considerable advantage over synchronous transfers even though we steal two threads from the main computation to perform object selection and transfer asynchronously.

Table III shows the number of phases in which a transfer occurs and the number of objects moved in total. It also shows the percentage of phases that data movement is required. For example, for CG, out of 17 phases, six phases require data movement between two memories and a total of 13 objects are moved. Results prove the benefit of dynamic placement since it adapts the application behavior as the working set of application changes from phase to phase. In our experiments, we also observe that a higher transfers/phase percentage value has a direct relation to increased performance using asynchronous transfers. This might be because it gives more opportunities to hide the transfer latency since object selection algorithm runs at every phase regardless. For the analysis of asynchronous transfers, we spared 2 threads which yielded the best performance on Intel KNL. This number can vary across different machines.

D. Latency-Sensitive Applications

While some applications benefit from the high bandwidth characteristics of an HBM, others might get degraded perfor-

TABLE III: Some statistics about object movement

Applications	Phases	Transfers	Objs Moved	Transfers/Phase
CG	17	6	13	35.3%
BT	31	3	10	9.67%
LU	28	4	12	13.3%
SP	30	3	8	11.1%
FT	28	10	12	35.7%
SRAD	4	1	3	25.0%
HotSpot	11	6	9	54.5%
BFS	5	3	5	60.0%

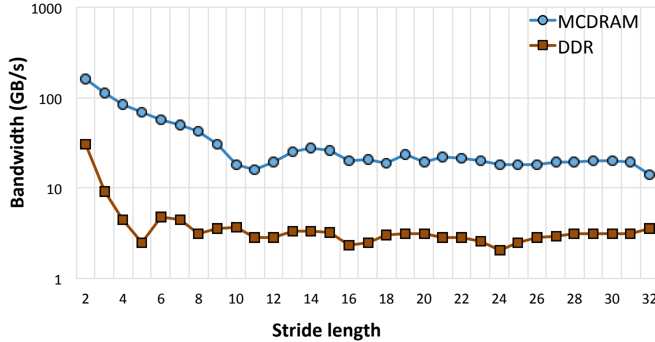


Fig. 6: Bandwidth degradation comparison of MCDRAM and DDR in Intel KNL as the stride length is increased.

mance because of higher latency of HBM compared to DRAM. Applications which include indirect or strided access can be characterized as latency-sensitive applications. Figure 6 shows the bandwidth degradation of MCDRAM and DDR in KNL. As the stride size increases, the bandwidth of both memories degrades drastically. Due to high access latency of HBM in Intel KNL, MCDRAM loses its advantage over DDR for some objects as in the BFS application.

Our cost model distinguishes between such objects by changing their access bandwidths. As discussed previously, if an application contains objects which exhibit indirect accesses, we change its access bandwidth to $Bandwidth_{strided}$. For HBM, the $Bandwidth_{strided}$ is lower than the $Bandwidth_{stream}$. Changing bandwidths for objects according to their access patterns can yield different and better placement strategies. We changed the bandwidth for BFS to monitor this effect. Three objects in two phases in BFS were allocated $Bandwidth_{strided}$ for calculating placement strategy. For the remaining objects we used stream bandwidth. Overall, we only observe 1% improvement but we expect the benefit is higher for an application which has a lot more objects with mixed access patterns. Future work will investigate this further.

VI. Related Work

It is expected that in near future computers will have a combination of different memory pools, each complementing the effect of the other [1]. In such a case, the demand for seamless object placement is likely to increase. For a programmer, it is ideal that these different memory pools are managed without any extra effort. In lieu of this, several works have been proposed to minimize the effort of the programmer for deciding object placement strategies.

Hardware centric approaches [19][25][18][9] usually leverage the built-in hardware components in a system. Performance counters and registers are a most widely used to extract hardware-level statistics. Hardware centric approaches can either work at the granularity of a whole application or at specific user-annotated points within the application. As the granularity increases, the overhead of gathering information from these components increases. To mitigate the overhead, sampling techniques are adapted. Ramos et al. [19] focus on monitoring the memory controller to analyze the memory access pattern of the application. After querying this information, they perform page migration from one memory to another. In [6], the authors present a scheme in which line swaps are made between the two memories through hardware. This approach is similar to a hardware cache that hosts recently accessed data. Like a cache, the granularity is a cache line.

Software-based approaches implement transfer of object at the software stack. These approaches are highly customizable and can cater to a variety of applications. These strategies can be either **1)** static or **2)** dynamic. In the former, once an object is allocated to a memory, it is not evicted from it. Whereas, in the latter, the runtime manages the locality of objects across different memory types, i.e. an object can be transferred between memories during application execution as in our proposed approach.

Servet et al. [21] use a two pass approach for object placement. Their decision for a two pass approach is based on the tools they use to gather object-level statistics. In their first pass they use Extrae [20] to gather the application execution profile. After extracting useful information, they use an object selection algorithm based on EVOP [17]. Later they override the malloc function call to allocate objects to the appropriate memory based on the object placement strategy listed by their selection scheme. However, their work does not consider the memory usage by an object during the application execution. Once allocated, objects are not evicted and reside in the same memory where they are allocated.

Laghari et al. [13] proposes an initial placement approach in which the objects are allocated to a particular memory based on their memory access pattern. The basis function in their allocation scheme considers loads and stores of a particular object, separately. In addition to this, their tool can prioritize the type of memory access based on the system used. In their work, Intel KNL yields a better application performance if write-intensive objects are allocated to the fast memory. Therefore, they prioritize stores over loads. Their approach, however, only performs initial placement.

Wu et al. [26] perform phase-based dynamic object allocation for NVRAM-based main memory systems. Their tool, Unimem, divides the application into phases where the phases are code blocks between two MPI calls. Based on the objects residing in those code blocks and their access pattern, a cost model decides which objects to place on the main memory.

Their cost model keeps track of objects usage, memory access type and the overhead incurred by the transfer of objects across different memory types. Unlike HBM-based main memory systems, in their approach it is assumed that placing objects on DRAM is always advantageous over NVRAM. Our cost model is flexible and more general, which can be applied to other heterogeneous memory systems. Unimem's evaluation is purely simulation based, while our work is conducted on real hardware, Intel KNL.

VII. Conclusion

Heterogeneous memory systems, which are equipped with multiple memories each with different characteristics, allow programmers to perform object placement explicitly on different memories. Such systems, however, introduce the burden of deciding which objects to place on which kind of memory. To assist the programmer, we present a phase-based dynamic data placement scheme which takes into account the object's life cycle and its activity in an application to suggest a placement strategy. We develop a runtime tool which can distinguish between the different characteristics of each memory and can perform object allocation in a way which will maximize overall application performance. Our tool divides the application into phases and performs asynchronous object eviction and admission into different memories. We tested our placement algorithm on various applications on Intel KNL which is equipped with MCDRAM and DDR memory, and observe a speedup of up to $2x$. Our future work will facilitate the placement further by fully automating the entire process.

VIII. Acknowledgments

Authors from Koç University are supported by the Turkish Science and Technology Research Centre Grant No: 215E185.

References

[1] J. A. Ang, R. F. Barrett, R. E. Benner, D. Burke, C. Chan, J. Cook, D. Donofrio, S. D. Hammond, K. S. Hemmert, S. M. Kelly, H. Le, V. J. Leung, D. R. Resnick, A. F. Rodrigues, J. Shalf, D. Stark, D. Unat, and N. J. Wright. Abstract machine models and proxy architectures for exascale computing. In *2014 Hardware-Software Co-Design for High Performance Computing*, pages 25–32, Nov 2014.

[2] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The nas parallel benchmarks—summary and preliminary results. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, Supercomputing '91, pages 158–165, New York, NY, USA, 1991. ACM.

[3] C. Cantalupo, V. Venkatesan, J. Hammond, K. Czurlyo, and S. D. Hammond. memkind: An extensible heap memory manager for heterogeneous memory platforms and mixed memory policies.

[4] C. Chan, D. Unat, M. Lijewski, W. Zhang, J. Bell, and J. Shalf. Software design space exploration for exascale combustion co-design. In J. M. Kunkel, T. Ludwig, and H. W. Meuer, editors, *Supercomputing*, pages 196–212, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[5] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*, pages 44–54, Oct 2009.

[6] C. Chou, A. Jaleel, and M. K. Qureshi. Cameo: A two-level memory organization with capacity of main memory and flexibility of hardware-managed cache. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-47, pages 1–12, Washington, DC, USA, 2014. IEEE Computer Society.

[7] P. Cicotti and L. Carrington. Adamant: Tools to capture, analyze, and manage data movement. *Procedia Computer Science*, 80:450 – 460, 2016.

[8] S. R. Dulloor, A. Roy, Z. Zhao, N. Sundaram, N. Satish, R. Sankaran, J. Jackson, and K. Schwan. Data tiering in heterogeneous memory systems. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, pages 15:1–15:16, New York, NY, USA, 2016. ACM.

[9] M. Islam, S. Banerjee, M. Meswani, and K. Kavi. Prefetching as a potentially effective technique for hybrid memory optimization. In *Proceedings of the Second International Symposium on Memory Systems*, MEMSYS '16, pages 220–231, New York, NY, USA, 2016. ACM.

[10] JEDEC. Wide I/O Single Data Rate. JESC 229, JEDEC, December 2011. http://www.jedec.org/standards-documents/docs/jesd229.

[11] JEDEC. High Bandwidth Memory (HBM) DRAM. JESC 235, JEDEC, October 2013. http://www.jedec.org/standards-documents/docs/jesd235.

[12] J. Jeffers, J. Reinders, and A. Sodani. *Intel Xeon Phi Processor High Performance Programming: Knights Landing Edition 2Nd Edition*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition, 2016.

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

[14] D. Li, J. S. Vetter, G. Marin, C. McCurdy, C. Cira, Z. Liu, and W. Yu. Identifying opportunities for byte-addressable non-volatile memory in extreme-scale scientific applications. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, pages 945–956, May 2012.

[15] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 89–100, New York, NY, USA, 2007. ACM.

[16] J. T. Pawlowski. Hybrid memory cube (hmc). In *2011 IEEE Hot Chips 23 Symposium (HCS)*, pages 1–24, Aug 2011.

[17] A. J. Peña and P. Balaji. Toward the efficient use of multiple explicitly managed memory subsystems. In *2014 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 123–131, Sept 2014.

[18] M. K. Qureshi, V. Srinivasan, and J. A. Rivers. Scalable high performance main memory system using phase-change memory technology. *SIGARCH Comput. Archit. News*, 37(3):24–33, June 2009.

[19] L. E. Ramos, E. Gorbatov, and R. Bianchini. Page placement in hybrid memory systems. In *Proceedings of the International Conference on Supercomputing*, ICS '11, pages 85–95, New York, NY, USA, 2011. ACM.

[20] H. Servat, G. Llort, K. Huck, J. GiméNez, and J. Labarta. Framework for a productive performance optimization. *Parallel Comput.*, 39(8):336–353, Aug. 2013.

[21] H. Servat, A. J. Peña, G. Llort, E. Mercadal, H. C. Hoppe, and J. Labarta. Automating the application data placement in hybrid memory systems. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 126–136, Sept 2017.

[22] S. S. Shende and A. D. Malony. The tau parallel performance system. *The International Journal of High Performance Computing Applications*, 20(2):287–311, 2006.

[23] D. Unat, C. Chan, W. Zhang, S. Williams, J. Bachan, J. Bell, and J. Shalf. Exasat: An exascale co-design tool for performance modeling. *The International Journal of High Performance Computing Applications*, 29(2):209–232, 2015.

[24] D. Unat Erten. Access pattern-aware data placement for hybrid dram/nvm memories. *Turkish Journal of Electrical Engineering and Computer Sciences*, 29(2):209–232, 2017.

[25] A. Vega, F. Cabarcas, A. Ramírez, and M. Valero. Breaking the bandwidth wall in chip multiprocessors. In *2011 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation*, pages 255–262, July 2011.

[26] K. Wu, Y. Huang, and D. Li. Unimem: Runtime data managementon non-volatile memory-based heterogeneous main memory. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '17, pages 58:1–58:14, New York, NY, USA, 2017. ACM.