Object Placement for High Bandwidth Memory Augmented with High Capacity Memory

Mohammad Laghari

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

Didem Unat

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

***Abstract*—High bandwidth memory (HBM) is a new emerging technology that aims to improve the performance of bandwidth-limited applications. Even though it provides high bandwidth, it must be augmented with DRAM to meet the memory capacity requirement of many applications. Due to this limitation, objects in an application should be optimally placed on the heterogeneous memory subsystems. In this study, we propose an object placement algorithm that places program objects to fast or slow memories in case the capacity of fast memory is insufficient to hold all the objects to increase the overall application performance. Our algorithm uses the reference counts and type of references (read or write) to make an initial placement of data. In addition, we perform various memory bandwidth benchmarks to be used in our placement algorithm on Intel Knights Landing (KNL) architecture. Not surprisingly high bandwidth memory sustains higher read bandwidth than write bandwidth, however, placing write-intensive data on HBM results in better overall performance because write-intensive data is punished by the DRAM speed more severely compared to read-intensive data. Moreover, our benchmarks demonstrate that if a basic block makes references to both types of memories, it performs worse than if it makes references to only one type of memory in some cases. We test our proposed placement algorithm with 6 applications under various system configurations. By allocating objects according to our placement scheme, we are able to achieve a speedup of up to 2x.**

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

I. Introduction

Memory bandwidth in multicore systems is not scaling fast enough to feed data to increasing number of cores, making the performance of many applications bound by memory bandwidth [31], [23], [30]. In efforts to overcome the bandwidth limitation of many systems, high bandwidth memory (HBM) has been recently developed [14]. The high bandwidth memory often needs a secondary high capacity memory (DDR) to meet the memory requirements of an application. This memory configuration is expected to continue until the cost of 3D technology reduces to the same price level per bit of capacity as the high capacity lower-bandwidth DDR memories [2].

In such a heterogeneous memory setting, the performance of an application relies on how different memories are utilized by the programmer because each memory differs in bandwidth from one another. An optimal or near-optimal object placement strategy can allow the application to scale efficiently on the underlying system and reach the full potential of any system's performance capability. We propose an object placement tool that suggests a placement policy to the programmer for each object. The proposed algorithm finds a placement based on the size of each object, its reference count and the space availability in the fast memory. Our tool first performs profiling of the application to obtain reference count of the objects in an application. Due to its higher bandwidth, keeping highly referenced objects in the fast memory would result in better performance of the application and would likely increase its sustained bandwidth. In cases where fitting the entire data in fast memory is not possible due to its space limitation our tool only suggests those objects that maximize memory accesses from fast memory based on the individual read-write bandwidths of both fast and slow memories.

To test our tool, we use Intel Knights Landing (KNL) processor and its MCDRAM as the fast memory, while DDR being the slow memory and refer to them likewise in the rest of the paper. In order to quantify the read and write bandwidths of HBM in KNL, we perform the STREAM benchmark [19] under various scenarios. The copy and triad kernels of STREAM provide us with a better understanding of the capabilities of both the fast and slow memories. The results from KNL benchmarks suggest that reads are faster than writes on both memory types. We devise a modified triad benchmark to assess the effect of mixed accesses, that is, the effect of accessing objects from different memories while executing a single program statement. We observe that in some cases the observed bandwidth of referencing to two types of memories in a basic block is worse than the bandwidth of referencing to a single type of memory. In summary, we make the following contributions in this work.

- We devise a set of benchmarks derived from well-known STREAM benchmark to conduct in-depth analysis of MCDRAM on Intel KNL architecture. These benchmarks allow us to measure how the fast and slow memories on KNL perform under various conditions.
- We propose a placement algorithm that suggests possible placements for objects in a program on the fast or slow memories. The algorithm works for any heterogeneous memory architecture, however, in this paper, we test our

algorithm on Intel KNL only.
- To evaluate the performance of the placement algorithm, we use a diverse set of applications and we observe a speedup of up to $2x$.

The remainder of the paper is organized as follows. Section II discusses related work. Section III presents the STREAM benchmark results and motivates our work. In Section IV we describe our placement algorithms and Section V presents the methodology. Results are presented in Section VI and conclusions are in Section VII.

II. Related Work

Heterogeneous memory systems that combine different types of memories have been proposed as a solution to mitigate the scalability problems of memory bandwidth-limited applications [20], [5]. However such systems expose programming complexities because the programmer has to decide placement of objects on different types of memories for best performance. There have been various works which propose solutions to make programming such systems easier for the programmer. Some of these techniques are operating system (OS)-based [34], [17] or hardware-based [22], [21], [33], where placement decision relies on pages rather than objects. Managing placement through the aforementioned approaches relieves the programmer of any additional changes to the program. However because the granularity of placement is based on pages, multiple objects that reside on the same page can be placed on the same type of memory even though these objects have very different reference traces. Our tool provides a placement policy based on individual objects, enabling a finer-grained management of placement.

Hassan et al. [12] shows that the software-based object placement approach for NVRAM systems is better than current state-of-the-art hardware or OS approaches. They argue that placement at the granularity of objects is better than placement at the granularity of pages. Their work also concludes that such an approach is more energy efficient than placement managed by hardware or OS. We adopt the object-based approach and apply it to the KNL architecture to improve the observed bandwidth by the application. Another work that uses object-based placement but for NVRAM and DRAM system is from Dulloor et al. [11]. The authors propose a tool, X-Mem, which provides an API that works on top of the default malloc function. Initially, the programmer has to explicitly tag data structures that will increase the application performance if kept in the fast memory. After tagging all such data structures, the program does profiling using Intel Pin [18]. Access counts are then used in a function, which calculates the benefit of keeping each data structure in the fast memory. Then the X-Mem allocator API uses jemalloc to allocate objects.

Similarly, in [29], authors propose an access pattern-aware solution for object placement for hybrid DRAM and NVRAM memories, which uses a static code analysis tool, ExaSAT[30], to gather application information at compile time. In that work, dynamic and static energy consumption of NVRAM plays an important role in making decisions about object placement. While our approach uses runtime information, sustained bandwidth of an application is the main objective for placement.

Some of the early work for data placement for heterogeneous memory systems use simulators rather than real architectures [8], [16], [32]. Steinke et al. [28] performs placement of programs on a scratchpad memory in embedded devices. In their approach, they place a method and its variables in a program on the fast memory, based on the number of instructions executed by each member function or variable access. The authors associate the instruction counts with their corresponding energy consumption and use Knapsack algorithm [24] to statically place objects on fast memory.

Closest work to ours is from Shen et al. [25], which studies the impact of heterogeneous systems on KeyStone II architecture. They develop a profiling tool called DataPlacer built on top of the Intel's Pin tool [18] that provides insights to programmers while porting code to systems with multiple types of memories. DataPlacer and their benchmark suite are designed for heterogeneous processor systems where there are two types of processors, each having its independent memory resource. Whereas our work focuses on systems with heterogeneous memories managed and accessed by a single type of compute resource.

III. Memory Bandwidth Benchmarks

To construct the placement algorithm for heterogeneous memory systems, we first study the capabilities of a system equipped with an HBM augmented with DDR. One of the variants of HBM recently introduced by Intel is MCDRAM, or *Multi-Channel DRAM* in the KNL processor [27]. MCDRAM is a configurable memory module, which can be set to either of the three modes on boot. These modes represent their accessibility by the programmer and their ease of use. The modes also define the granularity level at which an application can be configured to leverage maximum advantage from it. These modes are 1) *Cache Mode*, 2) *Flat Mode*, and 3) *Hybrid Mode*. In *cache mode*, MCDRAM acts as the last level cache to DRAM. The memory management in this mode is done by the hardware requiring no changes in the software. The downside of this mode is its added latency on cache misses. The second configurable mode is the *flat mode*, in which MCDRAM acts as a separately addressable memory module. This allows the programmer to control object placement to the level of granularity that they deem fit in order to maximize the application performance. Lastly, the *hybrid mode* is a combination of the two aforementioned modes, where part of the MCDRAM acts as the cache and the rest acts as a separately addressable memory. In this work, we are interested in the *flat mode* of MCDRAM since it allows the programmer to decide which objects to place on what kind of memory explicitly.

In this section we conduct bandwidth benchmarks to verify the capability of HBM in particular to what extent can it benefit an application. The STREAM benchmark is the *de facto* standard for performing bandwidth analysis of memory

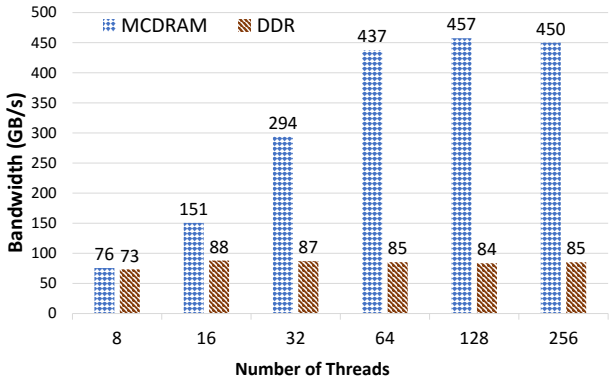

Fig. 1: Stream Triad Benchmark results for MCDRAM and DDR with $KMP\_AFFINITY = Scatter$.

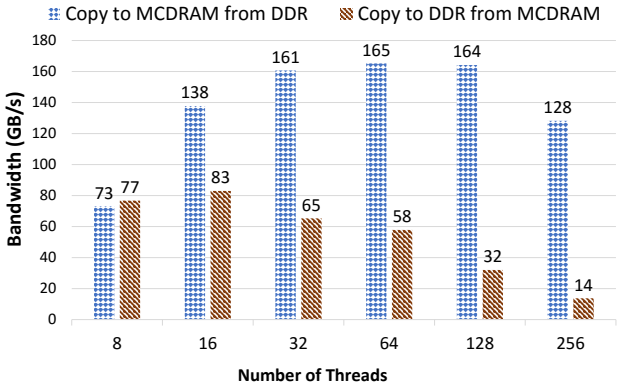

Fig. 2: Copy operation to and from the two memory types.

modules [19]. We have modified the STREAM benchmark to mimic various scenarios. The modified benchmarks and their results are discussed in detail in the following sections.

A. Experiment Setup

We perform our experiments on KNL processor equipped with 68 cores. MCDRAM in KNL acts as a fast memory and DRAM acts as a high capacity slow memory for our experiments. The system is configured to Sub-NUMA 4 clustering (SNC-4) mode with MCDRAM set to flat mode. Cores are distributed in tiles. Each tile consists of 2 cores, a shared L2 cache of 1MB and 4 Vector Processing Units, 2 for each core. In SNC-4, two of the clusters have 16 cores each while the other two have 18 cores each. Each cluster in SNC-4 mode stores data associated with its cores on the nearest MCDRAM non-uniform memory access (NUMA) node. This results in a lower latency for memory accesses. Flat mode allows us to manage the MCDRAM through the Memkind library [6]. Unless stated otherwise we set the affinity of threads to *scatter* instead of *compact* by using the flag `KMP_AFFINITY=scatter`. This allows the application to fully utilize the multiple channels accessing the memory modules removing any chances of congestion when fewer number of threads than cores are running. We use the `qopt-streaming-stores` flag and set it to `always` to bypass the cache since there is no data reuse in the stream benchmark. The *Memkind* library [6] developed by Intel provides an interface to allocate objects manually to available memory types. We use `interleaved` memory allocation provided by the library such that memory addresses are allocated to all memory banks in turn.

B. STREAM Benchmark

In this section we describe the results obtained from unmodified STREAM benchmark on KNL. We focus on the triad and copy kernels of STREAM:

1) COPY : $A[i] = B[i]$
2) TRIAD : $A[i] = alpha * B[i] + C[i]$

In our experiments, data was explicitly allocated to the desired type of memory using the Memkind library. With explicit allocation we assert that the data is only placed on the memory of our choice and this allows us to verify the bandwidth difference between on-package MCDRAM and the DDR.

Figure 1 shows the triad bandwidth achieved using the unmodified STREAM benchmark. MCDRAM observes around 450 GB/s, which matches the published figures by Intel [15]. In Sub-NUMA cluster mode by varying the number of threads, we experience that the peak bandwidth is achieved earlier, starting from only 64 threads and onwards, if the thread affinity is set to scatter. This means that the threads are distributed evenly across the tiles on all four clusters on the chip, which translates to better usage of the eight access channels of MCDRAM.

C. Copy Bandwidth Between Two Memories

Next we measure the sustained bandwidth of copying data from one memory to another. We modified the stream benchmark to copy objects from MCDRAM to DDR and vice versa by allocating source array to one memory and destination array to another. Similarly, these objects are allocated in an interleaved fashion. Figure 2 demonstrates the results of the copy operation. The experiments show that the copy bandwidth from MCDRAM to DDR is lower than the copy bandwidth from DDR to MCDRAM, which made us investigate the read and write bandwidths separately for the two types of memories. Figure 2 also shows that as the number of threads increase, the copy bandwidth to DDR decreases dramatically to only 14 GB/s using all the available threads.

D. Read vs Write Bandwidths

In this section we investigate the read and write bandwidths of MCDRAM and DDR. The experiments show MCDRAM bandwidth of 350 GB/s and 270 GB/s for read and write, respectively. It seems that up to 32 cores, the benchmark is not bandwidth-limited on MCDRAM. In general write operations have slightly less overhead than reads in terms of data movement. The higher cost of write at the memory controller or in the memory shows up only when the application becomes bandwidth-limited. We see this trend on MCDRAM up to 32 cores, when the write performance is

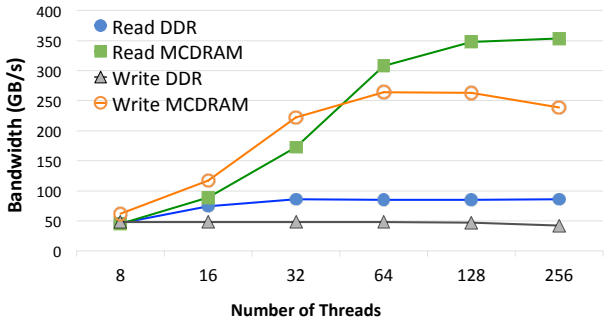

Fig. 3: Read-Write bandwidth comparison between DDR and MCDRAM.

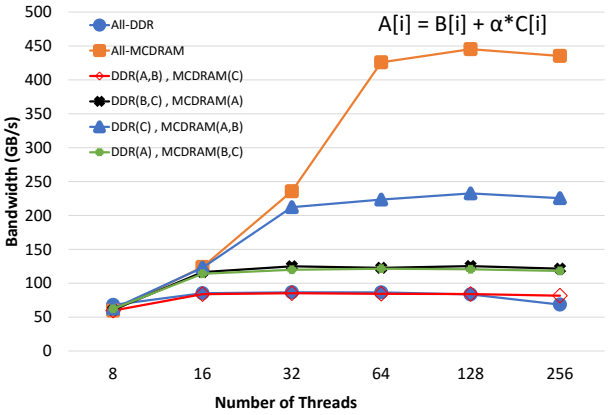

Fig. 4: Triad bandwidth comparison of objects placed in different memory types.

better than read. Bandwidth on DDR is more stable with varying number of threads, 88 GB/s and 50 GB/s for reads and writes, respectively.

E. Mixed Triad Benchmark

In this benchmark, we measure the sustained bandwidth when an operation makes references to both types of memories to either load or store its operands. Measuring the bandwidth for different configurations is important because in an application objects referenced in a loop or basic block may come from different memory types thus observed bandwidth can be lowered than that of if all objects are referenced from a single type of memory. The triad operation uses three data objects A, B and C and a scalar quantity α. We modify the stream triad and place the objects as follows:

1) A and B in MCDRAM, C in DDR
2) A in MCDRAM, B and C in DDR
3) B and C in MCDRAM, A in DDR
4) C in MCDRAM, A and B in DDR

Figure 4 shows very interesting results. We observe that the best performance is achieved in the first configuration where the MCDRAM contains two of the objects and one of them is the object associated with the write operation. This is because the MCDRAM can handle both reads and writes at a higher bandwidth than DDR. Among these four configurations, the lowest bandwidth is observed in the third case, where MCDRAM has both of the read-intensive objects, while DDR has the write-intensive object. This shows that the write operation to DDR becomes the bottleneck causing the entire operation to be limited to only approximately 80 GB/s at 64 threads, in which use of MCDRAM provides no performance benefit. We compare these four cases with two additional cases where all the data is either kept in the MCDRAM or DDR. As expected the best bandwidth is achieved when all the objects are in MCDRAM. However, the performance of all objects in DDR is better than configurations 3) and 4) when more than 64 threads are used.

F. Discussion of Memory Benchmarks

We propose the following placement scheme for Intel KNL architecture. If the size of data is less than the remaining space in MCDRAM, we suggest allocating all objects to the MCDRAM. If the size of data exceeds that of the remaining space in MCDRAM (fast memory), then we suggest allocating write intensive data to MCDRAM while the read intensive data to DDR (slow memory). The read bandwidth of MCDRAM is higher than its write bandwidth, same is the case for DDR. However, if we perform object placement according to higher bandwidth criteria, the write bandwidth of DDR would lead to the overall bandwidth of the system to become a bottleneck causing the overall sustained bandwidth to be lower than the case when write intensive data is allocated in MCDRAM. A higher bandwidth policy might heavily penalize the application. This finding constitutes the basis of our placement algorithm, which will be discussed in the next section.

IV. Placement Algorithm

This section discusses the proposed object placement algorithm. We discuss the naive placement algorithm and present an optimized version of it. Then we improve that algorithm by differentiating read references from write references for each object.

A. Naive Algorithm

The naive version of the algorithm picks objects greedily according to their frequency of accesses along with their sizes. It takes three inputs 1) the sizes of each object, 2) the reference count of each object, and 3) the fast memory size. First, it computes all the possible sets of objects in a program. Then by iterating over all the sets generated in the previous step, it calculates the total size requirement and the total reference count for each subset. In the meantime the algorithm checks if the total object size of the subset exceeds the fast memory capacity, if so, it excludes that particular subset from consideration of potential placement on the fast memory. It repeats this procedure until all the subsets are consumed and returns the subset with the highest overall reference count while having enough size to be accommodated in fast memory. The runtime complexity of this algorithm grows exponentially therefore we improve this algorithm, which will be discussed next.

B. Improved Placement Algorithm

To improve the time complexity of the naive placement algorithm, we resort to a dynamic programming scheme, which is largely known as the Knapsack algorithm [24]. This algorithm brings down the complexity of our approach from exponential to pseudo-exponential time, allowing us to generate mappings for a relatively large input. In the placement problem, the knapsack is considered as the fast memory. It is associated with a maximum weight that it can carry, which in our case is the total capacity of the fast memory. The data objects to be placed on fast memory are represented as the individual items, which can be carried by the knapsack. These individual items have the properties of weight and a value, which in our case are the sizes of each object and its reference count, respectively. High reference count refers to a high value. The objective is to maximize the most valuable objects in fast memory without overloading it. Unlike the naive algorithm, this algorithm does not generate all the possible combinations of the input. Instead it computes the best possible solution (object mapping on fast memory) for each value of the size from one to maximum fast memory size recursively.

We present two flavors of the improved placement algorithm based on Knapsack. The first one takes into account the total number of references made by an object. In this case, an object's fate is decided only by its overall reference count without differentiating references as read or write. We refer to this flavor as *write-agnostic placement*. As discussed in Section III the read and write bandwidths differ for both fast and slow memories. Therefore, for a program with objects having widely varying read and write counts, we propose the second flavour of the improved algorithm. We refer to this variant as *write sensitive placement*.

1) Write-Agnostic Placement: Algorithm 1 shows the pseudo code of this algorithm based on the dynamic programming implementation of Knapsack. First, the algorithm initializes four data structures, namely *M, inFast, access*, and *size*. *M* is the grid in which we evaluate whether to include an object, *inFast* is a boolean auxiliary grid where we store the decision of inclusion of each object, *access* stores the reference counts of each object and *S* stores the size in kilobytes of each object. The variables *n* indicates the number of objects and *W* is the capacity of fast memory. In line 8, the algorithm iterates over the objects considering each sub-solution at a time. At each iteration of the first for loop, the algorithm considers the object if its size is less than or equal to the current size being considered on the fast memory. If the object can fit inside the fast memory, the algorithm checks (line:12-15) if keeping the object in fast memory is better than the current objects in fast memory. If the object qualifies to be in the fast memory, a value of *true* is set for that object in the boolean grid. When all the objects in the list are considered, the algorithm terminates. After this, another loop iterates over the boolean grid to select the selected items. These items are finally suggested to the programmer to place on the fast memory.

Algorithm 1 Object Placement

```
1:  procedure PLACEOBJECTS(Objects, M, inFast, Access, S, α)
2:      M[0:n-1][0:W-1] ← 0
3:      inFast[0:n-1][0:W-1] ← false
4:      Access[0:n-1] ← Access counts for objects
5:      Reads[0:n-1] ← getReadAccesses(Access)
6:      Writes[0:n-1] ← getWriteAccesses(Access)
7:      S[0:n-1] ← Object Sizes
8:      Candidates ← {}
9:      for i = 1...n do
10:         for j = 0...W do
11:             M_{i,j} ← M_{i-1,j}
12:             if S_{i-1} ≤ j then
13:                 M_{i,j} ← max(M_{i-1,j}, Reads_{i-1} + α∗ Writes_{i-1} + M_{i-1,j-S_{i-1}})
14:                 if M_{i,j} > M_{i-1,j} then
15:                     inFast_{i,j} ← true
16:                 end if
17:             end if
18:         end for
19:     end for
20:     while n > 0 do
21:         if inFast_{n,W} ==true then
22:             Candidates.push(Objects_{n-1}.getName())
23:             W = W − S_n
24:         end if
25:         n = n − 1
26:     end while
27:     return Candidates
28: end procedure
```

2) Write-Sensitive Placement: To address both reads and writes separately, we allocate two separate data structures, one for each type of reference counts for all objects, namely *Reads* and *Writes* in Algorithm 1. The working of the algorithm is the same as described in the previous section. However, for this case, when an object is being considered to be placed in the fast memory, its write access count is multiplied by a coefficient, α to weight writes more than reads. The value of α can vary between different heterogeneous memory systems. For KNL, we observe that read bandwidth is roughly 1.5 times the write bandwidth, therefore we use the coefficient as 1.5 in our experiments. The coefficient for another system could be determined by measuring the ratio between its read and write bandwidths. By using a coefficient, we penalize the reads.

V. Methodology

For characterizing the application behaviour, there are various tools based on static code analysis, binary instrumentation or profiling [7], [26], [1]. However many of these tools provide overall behaviour of the application but do not offer detailed object specific information. To gather object level information, we leverage the ADAMANT tool [10] developed at the San Diego Supercomputer Center. ADAMANT combines both

hardware counters and simulations to collect performance data. It captures the data objects that are statically, dynamically or automatically allocated and maintains an object database and their associated events. The tool distinguishes between stores and loads which allows us to make decisions based on different behavior of an object during its life cycle.

The placement tool that we have developed has two phases of execution. In the first phase, we acquire the object-level statistics using ADAMANT. These statistics include distinguished reads and writes reference along with the size of the object, which are fed into the placement algorithm. The algorithm then suggests which objects to be placed on fast memory. Currently, we provide the candidate object information to the programmer and it is programmer's responsibility to allocate these objects on the suggested memory module. Our future work will focus on easing the allocation part of the placement and will remove the need of programmer's intervention.

VI. Evaluation

To evaluate the proposed placement algorithm, we perform experiments on the Intel KNL architecture using 64 threads with thread affinity set to *scatter*. We compare the performance of the placement algorithm against various system configurations summarized in Table I.

1) **All-DDR:** All objects are allocated in the slow memory (DDR).
2) **All-MCDRAM:** All objects are allocated in the fast memory (MCDRAM).
3) **4GB Cache:** We make all allocations to the DDR in this mode and let the hardware cache objects into MCDRAM. We also fix the size of MCDRAM acting as last level cache to 4GB for a fair comparison with our placement algorithm where there is no cache. In this configuration, no explicit allocations are made to the MCDRAM.
4) **8GB Cache:** This configuration is the same as the previous one except that the MCDRAM size acting as last level cache is 8GB. This configuration allows us to compare our placement algorithm, where we set the cache size to 4GB and fast memory size to 4GB.
5) **Our Placement w/o Cache:** We allocate objects on the fast memory based on the suggestions provided by our placement algorithm. The fast memory size is set to 4GB in the algorithm. No hardware caching is enabled.
6) **Our Placement w/ Cache:** This is similar to the previous configuration, however we augment the allocatable fast memory used by our placement algorithm with last level cache by setting aside 4GB of MCDRAM for hardware caching.

We have two flavours of the placement algorithm as described in Section IV. **Write-Agnostic** allocates objects on the fast memory by considering load and store accesses cumulatively. **Write-Sensitive** favors stores over loads during calculating the suggested placement. For our experiments, we have set the coefficient to 1.5 as discussed in Section IV-B2.

TABLE I: System configuration modes

	Fast Memory	Slow Memory	Cache (L3)	Boot Mode
All-DDR	—	384GB	—	Flat
All-MCDRAM	16GB	—	—	Flat
4GB Cache	—	384GB	4GB	Hybrid
8GB Cache	—	384GB	8GB	Hybrid
Our Placement (w/o Cache)	4GB	384GB	—	Flat
Our Placement (w/ Cache)	4GB	384GB	4GB	Hybrid

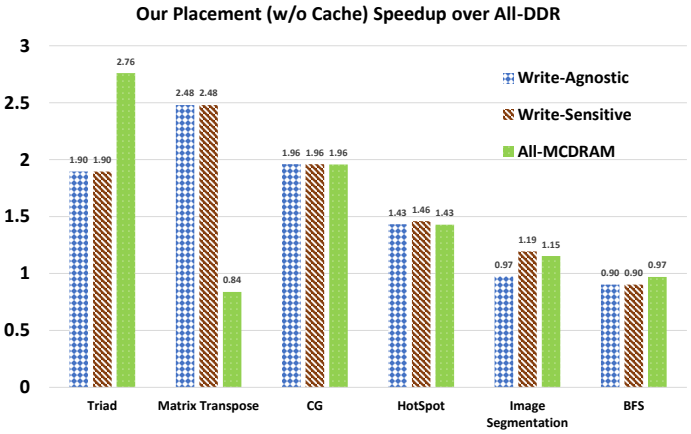

Fig. 5: Speedup of our placement configuration achieved over placing all objects in the slow memory

In the evaluations, we use 6 applications, whose descriptions are provided in table II. We evaluate applications based on the speedups achieved, which allows us to compare applications with varying execution times in a single figure.

A. Comparison against All-DDR and All-MCDRAM

Figure 5 illustrates the speedup achieved by the proposed placement without L3 cache over the All-DDR configuration, i.e. when all the data of an application is placed on the slow memory. The figure also shows the case when all the data is allocated on the fast memory. Both versions of our placement algorithm outperform the All-DDR for most of the applications. However, we observe a performance degradation in *BFS*. This is mainly due to the nature of graph traversal applications. Such applications are limited by memory latency [3]. The memory latency of MCDRAM is higher than that of DDR. Therefore, placing more data on MCDRAM coupled with indirect access to objects increases the latency and degrades the performance of application.

For *Triad*, *Matrix Transpose* and *CG*, both versions of our placement algorithms suggest the same placement. This is due to the access pattern of the application. Write-Sensitive algorithm suggests a better placement over Write-Agnostic for applications where there is a significant difference in read and write references of the objects. The suggestions by Write-Sensitive version of our proposed algorithm for *HotSpot* and *Image Segmentation* yields higher performance because of the

TABLE II: Evaluated Applications

Applications	Description	# of objects	Footprint (GB)
Triad [19]	Triad kernel of STREAM benchmark	3	6.00
Matrix Transpose	Transpose of a matrix is stored in another matrix	2	6.00
CG [4]	Conjugate Gradient solves unstructured sparse linear systems	13	5.23
HotSpot [13]	Approximates processor temperature and power by solving PDEs	3	6.00
Image Segmentation	Divides and recolors an image into segments to identify boundaries	7	5.55
BFS [9]	Breath first search graph traversal algorithm	6	9.74

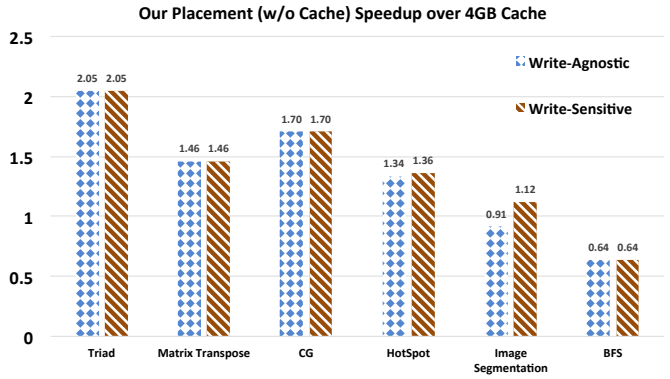

Fig. 6: Speedup achieved over placing all objects in the slow memory coupled with 4GB MCDRAM as a last level cache.

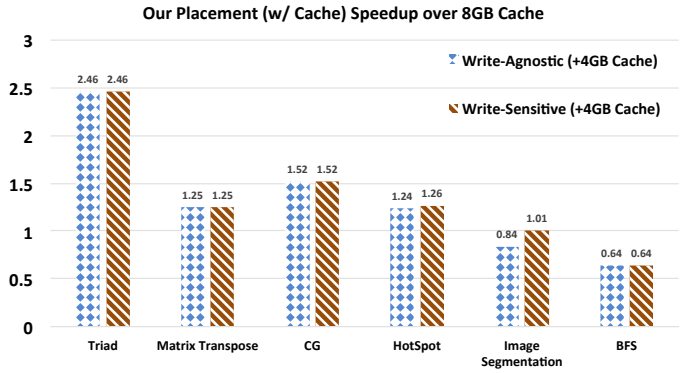

Fig. 7: Speedup achieved over placing all objects in slow memory using 8GB MCDRAM as a last level cache. Our placement is coupled with a 4GB L3 MCDRAM cache, an addressable 4GB of MCDRAM and DDR.

difference between the amount of loads and stores for majority of their objects. The three objects in *HotSpot* have a difference of more than 80% between its load and store counts. While *Image Segmentation* has a difference of 70% between load and store counts for more than half of its objects.

B. Comparison against MCDRAM as a last level cache

1) 4GB Cache: Figure 6 shows the speedup achieved by our placement algorithm against the implicit placement done by hardware caching. We evaluate the applications in this mode because it performs hardware caching at runtime without requiring any changes to the source code, therefore no intervention from the programmer is required. We find that all applications except *BFS* perform better with the placement suggested by our algorithm. This shows that our placement algorithm can beat the hardware caching and suggest a more intelligent placement of objects. With *Matrix Transpose*, our placement scheme suggests to place the object being written in to on the fast memory. Depending on how the transpose loop is written, this may lead to non-contiguous accesses to the corresponding object in fast memory. Due to the higher latency of MCDRAM, the array that is accessed non-contiguously should not be placed in fast memory. Therefore, we implemented the matrix transpose in a way that the elements of write array are referenced contiguously for all cases. This allows us to leverage the high bandwidth characteristic of MCDRAM without getting penalized by its higher memory latency trait.

2) 8GB Cache: Figure 7 shows the speedup achieved by our algorithm coupled with a 4GB of L3 MCDRAM cache over a configuration with 8GB of L3 MCDRAM cache and slow memory. According to our settings, with this configuration all the data can be cached in the fast memory (acting as L3 cache of 8GB). With this comparison, we show that our placement coupled with a 4GB L3 cache can yield a better performance.

VII. Conclusion

In this work we explore the benefits of heterogeneous memory systems. Such systems are equipped with a fast memory with higher bandwidth but lower capacity, and a slow memory with lower bandwidth but higher capacity. Due to their different characteristics, allocating objects in a particular memory can largely affect the performance of an application. To address this challenge, we propose a placement algorithm that takes into account the knowledge about memory accesses and sizes of objects in an application, and suggests a placement scheme of objects in the fast memory. We tested our placement algorithm on various applications on Intel KNL which is equipped with a fast (MCDRAM) and a slow (DDR) memory, and observe a speedup of up to $2x$. We also observe that both memories have different bandwidth rates for load and store operations. We use this to suggest an informed placement scheme to the programmer. Our future work will facilitate the placement further by fully automating the process.

VIII. Acknowledgments

Dr. Unat is supported by TUBITAK with project number 116C066. Authors from Koç University are supported by TUBITAK Grant No: 215E185. Authors would like to thank

Dr. Pietro Cicotti from San Diego Supercomputer Center for his input in project.

REFERENCES

[1] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent. Hpctoolkit: tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience*, 22(6):685–701, 2010.

[2] 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 *Proceedings of the 1st International Workshop on Hardware-Software Co-Design for High Performance Computing*, Co-HPC '14, pages 25–32, Piscataway, NJ, USA, 2014. IEEE Press.

[3] K. Asanović, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[4] 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.

[5] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel. Scratchpad memory: a design alternative for cache on-chip memory in embedded systems. In *Proceedings of the Tenth International Symposium on Hardware/Software Codesign. CODES 2002 (IEEE Cat. No.02TH8627)*, pages 73–78, May 2002.

[6] 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. Technical report, Sandia National Laboratories (SNL-NM), Albuquerque, NM (United States), 2015.

[7] C. Chan, D. Unat, M. Lijewski, W. Zhang, J. Bell, and J. Shalf. *Software Design Space Exploration for Exascale Combustion Co-design*, pages 196–212. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[8] N. Chatterjee, M. Shevgoor, R. Balasubramonian, A. Davis, Z. Fang, R. Illikkal, and R. Iyer. Leveraging heterogeneity in dram main memories to accelerate critical word access. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, pages 13–24, Washington, DC, USA, 2012. IEEE Computer Society.

[9] 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 *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, IISWC '09, pages 44–54, Washington, DC, USA, 2009. IEEE Computer Society.

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

[11] 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.

[12] A. Hassan, H. Vandierendonck, and D. S. Nikolopoulos. Software-managed energy-efficient hybrid dram/nvm main memory. In *Proceedings of the 12th ACM International Conference on Computing Frontiers*, CF '15, pages 23:1–23:8, New York, NY, USA, 2015. ACM.

[13] W. Huang, S. Ghosh, S. Velusamy, K. Sankaranarayanan, K. Skadron, and M. R. Stan. Hotspot: Acompact thermal modeling methodology for early-stage vlsi design. *IEEE Trans. Very Large Scale Integr. Syst.*, 14(5):501–513, May 2006.

[14] JEDEC. High Bandwidth Memory (HBM) DRAM. JESC 235, JEDEC, October 2013.

[15] 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.

[16] 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.

[17] F. X. Lin and X. Liu. Memif: Towards programming heterogeneous memory asynchronously. *SIGARCH Comput. Archit. News*, 44(2):369–383, Mar. 2016.

[18] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM.

[19] J. D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, 1995.

[20] M. Pavlovic, Y. Etsion, and A. Ramirez. On the memory system requirements of future scientific applications: Four case-studies. In *2011 IEEE International Symposium on Workload Characterization (IISWC)*, pages 159–170, Nov 2011.

[21] M. K. Qureshi, V. Srinivasan, and J. A. Rivers. Scalable high performance main memory system using phase-change memory technology. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 24–33, New York, NY, USA, 2009. ACM.

[22] 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.

[23] B. M. Rogers, A. Krishna, G. B. Bell, K. Vu, X. Jiang, and Y. Solihin. Scaling the bandwidth wall: Challenges in and avenues for cmp scaling. *SIGARCH Comput. Archit. News*, 37(3):371–382, June 2009.

[24] R. Sedgewick. *Algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1984.

[25] D. Shen, X. Liu, and F. X. Lin. Characterizing emerging heterogeneous memory. In *Proceedings of the 2016 ACM SIGPLAN International Symposium on Memory Management*, ISMM 2016, pages 13–23, New York, NY, USA, 2016. ACM.

[26] S. S. Shende and A. D. Malony. The tau parallel performance system. *Int. J. High Perform. Comput. Appl.*, 20(2):287–311, May 2006.

[27] A. Sodani, R. Gramunt, J. Corbal, H. S. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agarwal, and Y. C. Liu. Knights landing: Second-generation intel xeon phi product. *IEEE Micro*, 36(2):34–46, Mar 2016.

[28] S. Steinke, L. Wehmeyer, B. Lee, and P. Marwedel. Assigning program and data objects to scratchpad for energy reduction. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '02, pages 409–, Washington, DC, USA, 2002. IEEE Computer Society.

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

[30] 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.

[31] 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.

[32] W. Wei, D. Jiang, S. A. McKee, J. Xiong, and M. Chen. Exploiting program semantics to place data in hybrid memory. In *Proceedings of the 2015 International Conference on Parallel Architecture and Compilation (PACT)*, PACT '15, pages 163–173, Washington, DC, USA, 2015. IEEE Computer Society.

[33] Z. Xue and D. B. Thomas. Sysalloc: A hardware manager for dynamic memory allocation in heterogeneous systems. In *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–7, Sept 2015.

[34] W. Zhang and T. Li. Exploring phase change memory and 3d die-stacking for power/thermal friendly, fast and durable memory architectures. In *2009 18th International Conference on Parallel Architectures and Compilation Techniques*, pages 101–112, Sept 2009.