Data Placement on Heterogeneous Memory Architectures

by

Mohammad Laghari

A Dissertation Submitted to the
Graduate School of Sciences and Engineering
in Partial Fulfillment of the Requirements for
the Degree of

Master of Science

in

Computer Science and Engineering

KOÇ UNIVERSITY

August 6th, 2018

Data Placement on Heterogeneous Memory Architectures

Koç University

Graduate School of Sciences and Engineering

This is to certify that I have examined this copy of a master's thesis by

Mohammad Laghari

and have found that it is complete and satisfactory in all respects,

and that any and all revisions required by the final

examining committee have been made.

Committee Members:

____

Asst. Prof. Didem Unat

____

Prof. Yücel Yemez

____

Asst. Prof. Ayşe Yılmazer

Date: ____________________________

I dedicate this work to my parents

ABSTRACT

Memory bandwidth has long been the limiting scaling factor for high-performance applications. To overcome this limitation, various heterogeneous memory systems have emerged. A heterogeneous memory system is equipped with multiple memories each with distinct characteristics. Among other characteristics, one common characteristic across these systems includes a memory with a significantly higher bandwidth than the others. This particular memory is known as the high bandwidth memory in general. Some of such high bandwidth memory (HBM) technologies include the hybrid memory cube (HMC) by Micron and high bandwidth memory standard by JEDEC. Intel's latest Xeon Phi processor, namely Intel Knights Landing (KNL), is equipped with an HBM known as the multi-channel DRAM, or MCDRAM, along with a DDR. The MCDRAM boasts up to 450 GB/s memory bandwidth as compared to its slower counterpart DDR which boasts only up to 88 GB/s. Unfortunately, as the bandwidth increases, the access latency for MCDRAM increases. Due to technology limitations and the high price per byte rate, HBM is offered in a small capacity as compared to traditional DDR in heterogeneous memory systems. Therefore, to overcome the smaller capacity of HBM, a heterogeneous memory system is also equipped with a higher capacity DDR. In such systems, the programmer is offered a choice to perform explicit allocations to each memory or let hardware handle data caching to the HBM. An intelligent object allocation scheme can yield a performance boost of the application. On the contrary, if an allocation is made without considering memory and application characteristics the overall performance of an application can drastically degrade. The object allocation choice coupled with the choice of deciding a system configuration can overburden the programmer resulting in increased programming effort and time consumption.

This thesis presents an object allocation scheme which is based on a system and application-specific cost model, a combinatorics optimization algorithm commonly known as the 0/1 Knapsack and a tool which combines these two components in practice. The cost model considers various characteristics of application data such as the object sizes, memory access counts, type of access, etc. In addition to application characteristics, the cost model also considers memory bandwidth under various conditions, including data streaming bandwidth and data copy bandwidth. These characteristics make our cost model rich allowing it to suggest an intelligent object allocation scheme. Using the aforementioned characteristics, the cost model determines a score for each object. These scores are used as values for each object in the 0/1 Knapsack algorithm to determine objects to be allocated on HBM where Knapsack size is HBM size. The tool uses the cost model to make an intelligent decision for object placement. The tool comes in two flavors: **1) static placement** where object placement is decided at the beginning of application execution and **2) dynamic placement** where objects are evicted and admitted to high bandwidth memory on the run based on application phases thus incurring the object movement cost. In the latter variant, the cost model considers the movement cost of objects from one memory to another while deciding on objects to be placed on the HBM. The tool is also capable of conducting object transfers asynchronously. The asynchronous transfers allow the tool to hide the transfer cost between phases.

We evaluate our allocation scheme using a diverse set of applications from NAS Parallel and Rodinia benchmark suites. The included applications have varying workloads and memory access patterns which exhibit the characteristics of real world applications. During the evaluation, Intels Knights Landing and its high bandwidth memory, namely MCDRAM, was used. The object placement suggested by the tool yields a speedup of up to 2.5x. We observe that latency-sensitive applications fail to benefit from high bandwidth memory allocation. This is because of the higher access latency of HBM. We also compared the results with the automatic hardware caching of Intel KNL. In hardware mode, the application is executed on the system without any changes and the caching is done by hardware automatically. We observe that our allocation scheme yields result better than that of hardware caching majority of the time.

ÖZETÇE

Bellek bant genilii, yksek performansl uygulamalar iin performans artrmada snrlayc bir faktr olmutur. Bu snrlamann stesinden gelmek iin eitli heterojen bellek sistemleri ortaya kmtr. Heterojen bir bellek sistemi her biri farkl zelliklere sahip olan birden fazla bellekten oluur. Bu bellek sistemleri eitlilik gsterseler de ortak bir zellie sahiptirler. Bu zellik sistemdeki dier belleklere kyasla daha yksek bant geniliine sahip bir bellein ierilmesidir. Bu zel bellek ise genelde yksek bant genilii bellei (HBM) olarak bilinir. HBM teknolojilerinden bazlar, Micron'un hibrid bellek kpn (HMC) ve JEDEC'in yksek bant genilii bellek standardn ierir. Intel'in en yeni Xeon Phi ilemcisi Intel Knights Landing (KNL), DDR ile birlikte ok kanall DRAM veya MCDRAM olarak da bilinen bir HBM ile donatlmtr. DDRde bant genilii 88 GB/s iken, MCDRAMde bu rakam 450 GB/sdir. Ancak, bant genilii arttka, MCDRAM iin gecikme sresi artar. Buna ek olarak, teknolojik kstlardan ve bayt bana yksek fiyattan dolay HBM, heterojen bellek sistemlerinde geleneksel DDR'ye kyasla kk bir kapasitede sunulmaktadr. HBM'nin bu kapasite kstnn stesinden gelmek iin, heterojen bellek sistemleri daha yksek kapasiteli bir DDR ile donatlmtr. Bu tr sistemlerde, programc her bellee zel ayrma yapabilir veya donanm HBM'yi nbellek olarak kullanabilir. Akll bir nesne yerletirme emas, uygulamalarda performans art salayabilir. Bunun aksine, bellek ve uygulamalarn zelliklerini dikkate almadan yerletirme yaplrsa, uygulamalarn genel performans byk lde debilir. Nesne yerletirme seimi ve buna ek olarak sistem konfigrasyonuna karar verme seimi, programlayc iin fazladan sorumluluk oluturur. Bu sorumluluk da artan programlama abas ve zaman tketimine neden olur.

Bu tez, sistem ve uygulamaya zel maliyet modeline dayanan bir nesne yerletirme emasn, genellikle 0/1 Knapsack olarak bilinen bir kombinatorik optimizasyon algoritmasn ve bu ikisini pratikte birletiren bir arac sunmaktadr. Maliyet modeli nesne

boyutlar, bellek eriim saylar, eriim tr, ve bunun gibi eitli uygulama zelliklerini gz nnde bulundurmaktadr. Uygulama karakteristiine ek olarak, maliyet modeli, veri ak bant genilii ve veri kopyalama bant genilii de dahil olmak zere eitli alardan bellek bant geniliini de dikkate almaktadr. Bu zellikler, akll bir nesne yerletirme emas nermeye yarayan maliyet modelimizin ieriini zenginletirmektedir. Belirtilen zellikleri kullanarak, maliyet modeli her nesne iin bir skor belirler. Bu skorlar, her bir nesnenin Knapsack algoritmasndaki boyutunu ifade eder. Knapsack boyutu olarak ise HBMnin boyutu kullanlr. Ara, nesne yerletirmede akll bir karar vermek iin maliyet modelini kullanr ve iki farkl yerletirme yapabilir: **1) nesnelerin yerleiminin** en bata yapld statik yerletirme ve **2) nesnelerin transferine** neden olan, uygulamann fazlarna dayal olarak HBMye aktarma veya HBMden karma yaplan dinamik yerletirme. Dinamik yerletirmede, maliyet modeli, HBM'ye yerletirilecek nesnelere karar verirken, nesnelerin bir bellekten dier bellee olan transferinin maliyetini gz nnde bulundurur. Ayrca ara, nesneleri ezamansz transfer etme yeteneine sahiptir. Ezamansz transferler, aracn transfer maliyetini uygulamann fazlar arasnda gizlemesine olanak salar.

Yerletirme emamz, NAS Paralel ve Rodinia karlatrmal deerlendirme paketlerinden alnan bir dizi uygulama zerinde test ettik. Kullanlan uygulamalar, pratikte kullanlan uygulamalarnn zelliklerini sergileyen, deiken i yklerine ve bellek eriim modellerine sahiptirler. Deerlendirme iin Intel'in Knights Landing ilemcisini ve yksek bant genilii bellei olan MCDRAMyi kullanldk. Gelitirdiimiz ara tarafndan nerilen nesne emas, 2,5 kata kadar bir hzlanma salad. Gecikme sresine duyarl uygulamalarn yksek bant geniliine sahip bellek yerleiminden yararlanamadklarn da gzlemledik. Bunun nedeni, HBM'nin daha yksek gecikme sresine neden olmasdr. Ayrca, sonular Intel KNL'nin otomatik donanm nbellei ile de karlatrdk. Donanm modunda, uygulama, herhangi bir deiiklik yaplmadan yrtld ve donanm tarafndan otomatik olarak nbellee alma ilemi yapld. Yerletirme emamzn, ou zaman otomatik donanm nbelleinden daha iyi sonu verdiini gzlemledik.

ACKNOWLEDGMENTS

First and foremost, I would like to thank my advisor Dr. Didem Unat for guiding me through the research which I have conducted during my Masters. Her trust in me to conduct research has been the primary driving force throughout my scholarship here at Koç. I would also thank TUBITAK for bestowing me with a generous funding to complete my studies.

I would also acknowledge the countless OpenSource projects which I have used in my research. These projects were the tools with which I crafted my thesis.

Lastly, I would thank my friends and family for sending out good thoughts and staying by my side.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

NOMENCLATURE

- KNL: Knights Landing
- HBM: High Bandwidth Memory
- HCM: High Capacity Memory
- OS: Operating System
- NPB: NAS Parallel Benchmarks
- NAS: NASA Advanced Supercomputing
- NUMA: Non-Uniform Memory Access
- HM: Heterogeneous Memory

Chapter 1

INTRODUCTION

1.1 Motivation

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 [Ang et al., 2014]. For instance, a typical DDR4 can only transfer data at a rate of 88GB/s to the CPU [Jeffers et al., 2016a] 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 [JEDEC, 2013], hybrid memory cube (HMC) by Micron [Pawlowski, 2011], or a technology like WideIO [JEDEC, 2011]. 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.

1.2 Heterogeneous Memory Systems

The computational capacity of a system is not completely utilized due to lacking bandwidth performance of the memory system. This results in lower than expected performance figures for various applications. This shortcoming can result in drastically degraded performance for applications with a large memory footprint. To overcome this, Heterogeneous Memory (HM) systems introduce multiple memories in a single system. HM systems try to overcome the shortcoming by providing a higher bandwidth memory which acts as a cache for the slower memory. HM systems can have a high bandwidth but lower capacity memory module acting as a cache for the traditional DDR. It can also have a much higher capacity memory module for which DDR can act as a cache. These modules can form a variety of memory configurations which are often presented to the programmer to choose from. The augmented memory can be used as **1)** a separately addressable memory module, **2)** a hardware managed memory module, or **3)** a combination of the two modes described above.

1.3 Hardware and Software Approaches

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 [Cantalupo et al.,] or frameworks assisting the programmer [Laghari and Unat, 2017]. 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 [Laghari and Unat, 2017], we proposed an initial object placement algorithm and observed a considerable improvement in execution time of an application. 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 [Jeffers et al., 2016a] [Laghari and Unat, 2017]. 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. This thesis proposes 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.

1.4 Initial Placement and Dynamic Placement

The tool described in this thesis offers two variants of object placement. The two variants mainly differ from one another by the amount of object and application level information they take while deciding which object should reside in a particular memory. **1) Initial Placement:** where objects are allocated to a particular memory before the application execution has begun. The allocations are based on the overall object activity during the entire run of the application. Since the objects are not moved during application execution from the memory they are allocated to in the beginning, this strategy does not incur any object movement cost. **2) Dynamic Placement:** where objects have the freedom of moving across memories. The application is divided into phases. Phases can be identified in a variety of ways. The reason to divide an application into phases is to keep those objects which are accessed together in a repeating fashion. One easy way to determine phases in an is to divide the application on loops. These loops can be for or $while$ loops. Since the objects in these phases will be accessed together, the benefit of keeping them together in HBM can compensate for the object movement across memories. The tool is capable of performing these transfers asynchronously. When a phase X is under execution, the tool can calculate the object placement strategy for phase $X + 1$ and perform the necessary object transfers before the phase $X + 1$ starts its execution.

1.5 Thesis Organization

The rest of this thesis is organized as follows. Chapter 2 gives a background about the underlying system architecture. It talks about the high bandwidth memory technologies and gives a brief introduction of the Intel's Knights Landing processor and its Multi-Channel DRAM. Chapter 3 talks about the related work in the domain of systems consisting of multiple memories. Chapter 4 describes the benchmarks conducted on one of the HM system, namely Intel KNL. Chapter 5 and 6 describe the initial and dynamic placement algorithms and their working, respectively. They also contain the evaluation and findings of both the placement strategies. Lastly, chapter 7 states the conclusion.

Chapter 2

BACKGROUND

2.1 High Bandwidth Memory Technologies

High bandwidth memory (HBM), as the name suggests, is a type of memory which is capable of providing a higher bandwidth to the system. Unlike traditional memory technologies, this technology allows the memory module to generate a greater sustained memory bandwidth increasing the throughput of the application being executed resulting in higher performance. The actual bandwidth can vary across particular technologies and the architecture of the memory module. The HBM equipped in Intel Knights Landing processor can yield up to 450 GB/s. Whereas its counterpart, the DDR4 is only limited to 88 GB/s. A high bandwidth memory is capable of achieving such high bandwidth figures because of its architectural design. A HBM module consists of DRAM dies stacked on top of each other. These dies are stacked atop each other and are connected with each other using a special material commonly known as "through-silicon-vias", or TSVs, and microbumps. Due to their vertical architecture, the HBM memory module does not occupy a large area therefore it is easily placed on-package. The memory module is connected to the CPU or GPU using a special interconnect, known as the interposer.

Several companies have pioneered in HBM technology and are offering their own flavor of high bandwidth memory integrated in their products. The HBM revolution started from AMD and SK Hynix. One of the first device using HBM technology was AMD's Fiji GPUs. Later JEDEC adopted HBM as the industry standard. Intel introduced Knights Landing processor which is equipped with Intel's proprietary HBM technology, namely Multi-Channel DRAM, or MCDRAM. Hybrid Memory Cube (HMC), developed by Micron Technologies, is a competitor of HBM

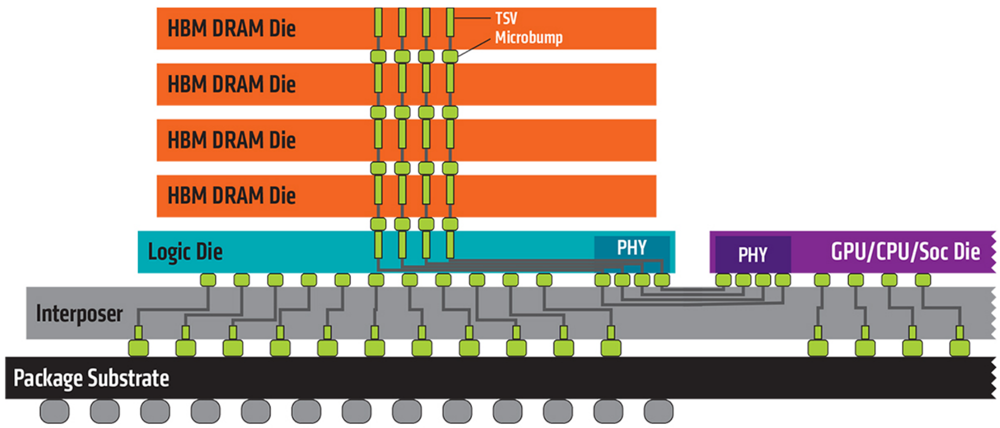

Figure 2.1: Model of a HBM technology developed by AMD. Image taken from https://www.amd.com/en/technologies/hbm

technology. Similar to HBM, HMC is uses a stacked DRAM die structure connecting them through TSVs. However, unlike the standard dies used in HBM, HMC uses DRAM dies with a higher number of memory banks. Both memories, however, aim to achieve a higher overall bandwidth.

Figure 2.1 shows AMD's HBM technology. The figure clearly shows the individual components of HBM technology. The memory module is placed on-package alongside the CPU of GPU.

2.2 Intel Knights Landing

In June 2016, Intel launched its latest Xeon Phi proessor, codenamed Intel Knights Landing (KNL). Unlike the older versions, this processor is self-bootable and binary compatible. Intel KNL is equipped with 72 cores arranged within a 2-dimensional mesh networks using 36 tiles. Each tile consists of 2 cores and 2 VPUs for each core. Each tile also contains a shared L2 cache of 1 MB. On the same package lie the 8

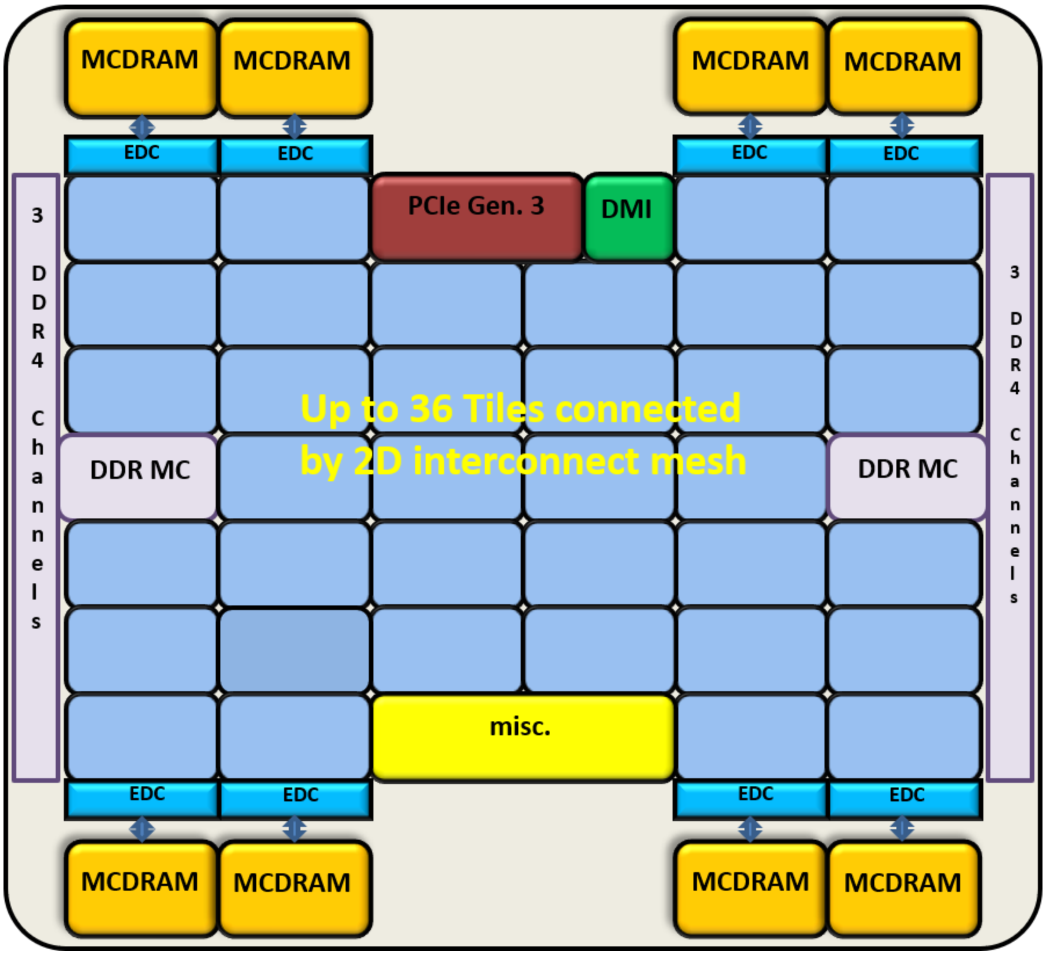

Figure 2.2: Tile arrangement and placement of MCDRAM modules on-package on an Intel KNL chip

modules of MCDRAM of size 2 GB each. Intel Knights Landing comes which an on-package HBM with a capacity of 16 GB which is capable of boasting 450 GB/s of memory bandwidth along with an off-package DDR that acts as a high capacity memory. Figure 2.2 shows the 2-dimensional mesh network of tiles in an Intel KNL chip. The chip also contains 6 channels to DDR, 3 on each side. Figure 2.3 shows the structure of a tile. Intel KNL mesh network can be configured to a variety of cluster modes. **1) All-to-All** clustering mode is the most general mode. There is no affinity between the tiles and memory in this mode. **2) Quadrant** clustering mode divides

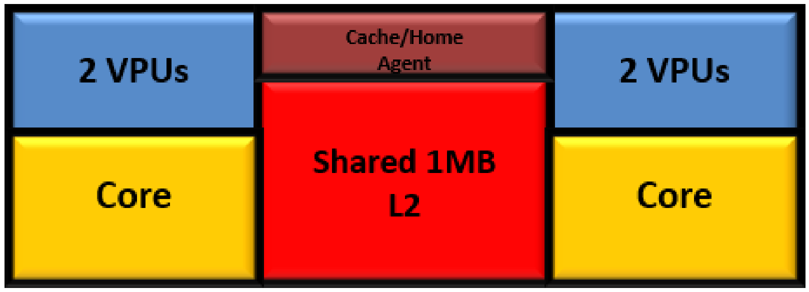

Figure 2.3: Intel KNL tile consisting of 2 cores and a shared L2 cache.

the chip into 4 virtual quadrant. In this mode the memory addresses are hashed to the directory which is in the same quadrant as the memory. The last mode is **3) Sub-NUMA** clustering (SNC) mode where each quadrant is exposed as a separate NUMA domain to the OS. SNC can be configured to either 2 or 4 quadrants. This mode introduces affinity between tile, directory and memory and incurs the lowest latency of all the modes.

2.2.1 Multi-Channel DRAM

As discussed previously, Intel KNL contains a high bandwidth memory, namely Multi-Channel DRAM or the MCDRAM. It comes with a capacity of 16 GB and can reach upto 450 GB/s. The MCDRAM modules are placed atop the chip. It is divided into 8 modules of 2 GB each. MCDRAM can be configured into 3 different memory modes. **1) Flat Mode:** MCDRAM acts as a separately addressable memory. In this mode, the memory allocations are to be managed by the software giving full control to the programmer. However, this mode introduces the effort of making manual allocation to MCDRAM. **1) Cache Mode:** MCDRAM acts as a hardware managed cache. No software changes are required by the programmer. Hardware handles all the caching. This mode is easier to manage as the programmer does not need to change application data allocation. Lastly **1) Hybrid Mode**: MCDRAM can be configured to act both separately addressable memory and hardware-managed cache.

Figure 2.4 shows visualization of each memory mode of Intel KNL.

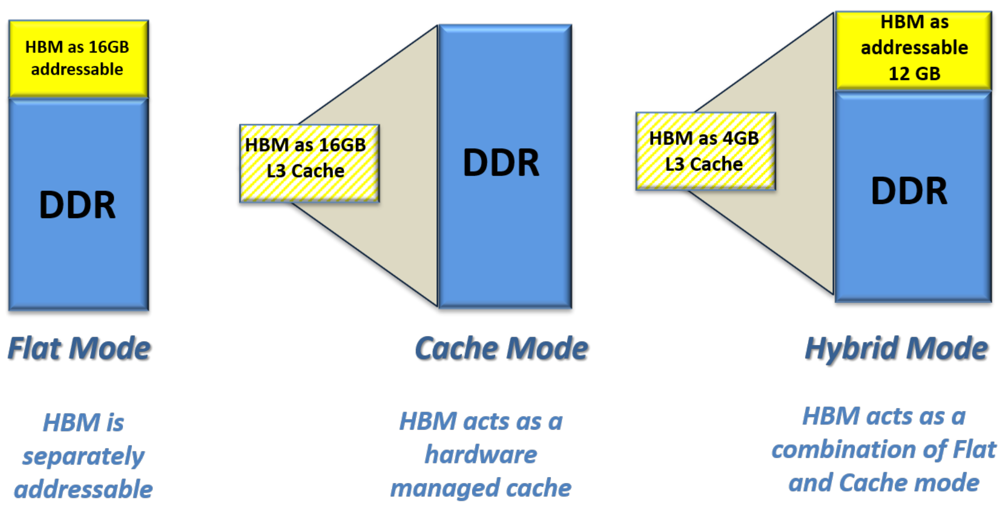

Figure 2.4: Intel KNL memory modes.

Chapter 3

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 [Ang et al., 2014]. 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 [Ramos et al., 2011] [Vega et al., 2011] [Qureshi et al., 2009] [Islam et al., 2016] 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. [Ramos et al., 2011] 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 [Chou et al., 2014], 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. [Servat et al., 2017] 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 [Servat et al., 2013] to gather the application execution profile. After extracting useful information, they use an object selection algorithm based on EVOP [Pea and Balaji, 2014]. 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. [Laghari and Unat, 2017] 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. [Wu et al., 2017] 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.

Chapter 4

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 [Sodani et al., 2016]. 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 modules [McCalpin, 1995]. We have modified the STREAM benchmark to mimic various scenarios. The modified benchmarks and their results are discussed in detail in the following sections.

4.1 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 [Cantalupo et al.,]. 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 [Cantalupo et al.,] 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.

4.2 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 : $A[i] = B[i]$

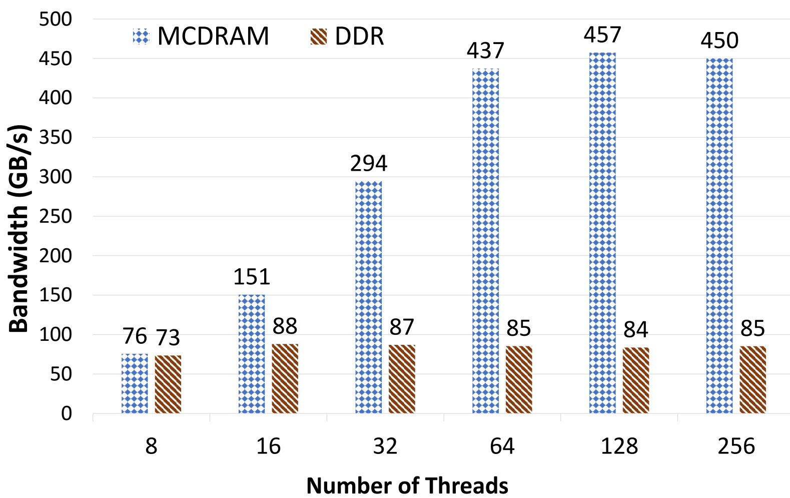

Figure 4.1: Stream Triad Benchmark results for MCDRAM and DDR with $KMP\_AFFINITY = Scatter$.

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 4.1 shows the triad bandwidth achieved using the unmodified STREAM benchmark. MCDRAM observes around 450 GB/s, which matches the published figures by Intel [Jeffers et al., 2016b]. 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.

4.3 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

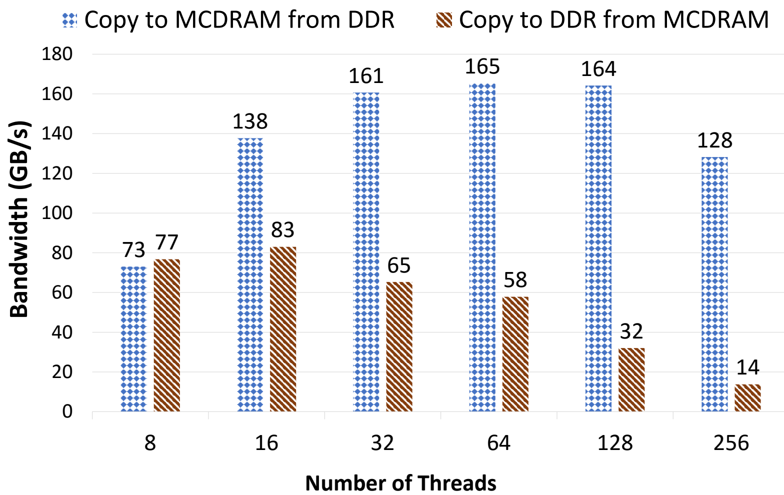

Figure 4.2: Copy operation to and from the two memory types.

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 4.2 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 4.2 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.

4.4 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 better

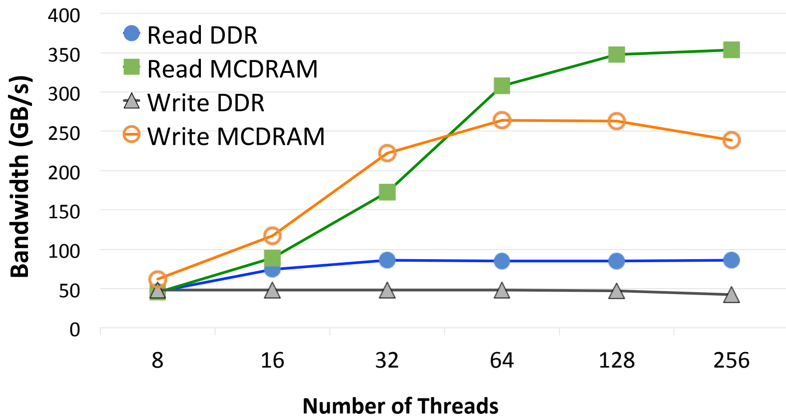

Figure 4.3: Read-Write bandwidth comparison between DDR and MCDRAM.

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.

4.5 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

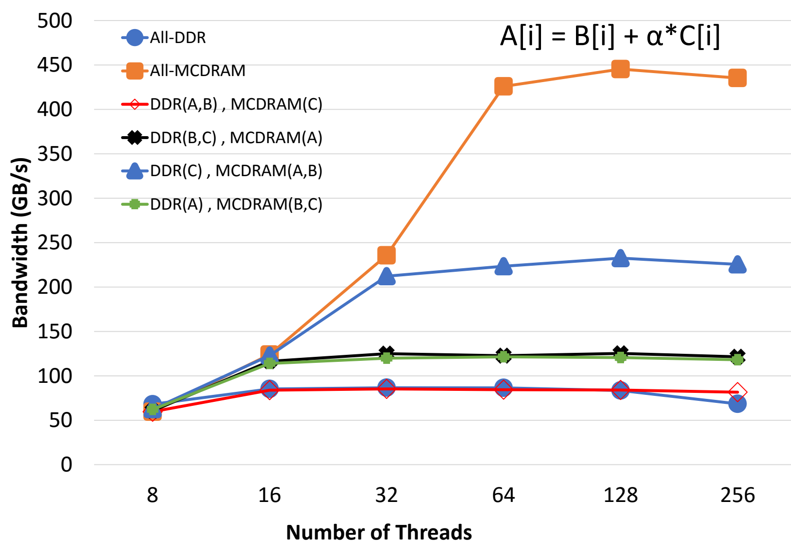

Figure 4.4: Triad bandwidth comparison of objects placed in different memory types.

Figure 4.4 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.

4.6 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.

Chapter 5

INITIAL PLACEMENT ALGORITHM

5.1 Naive Placement 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.

5.2 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 [Sedgewick, 1984]. 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 **??** 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 flavor of the improved algorithm. We refer to this variant as *write sensitive placement*.

5.2.1 Write Agnostic Object 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.

5.2.2 Write Aware Object 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.

5.3 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 5.1.

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 flavors of the placement algorithm as described in Section **??**. **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 5.2.2.

In the evaluations, we use 6 applications, whose descriptions are provided in table 5.2. We evaluate applications based on the speedups achieved, which allows us to compare applications with varying execution times in a single figure.

5.4 Comparison against All-DDR and All-MCDRAM

Figure 5.1 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

Table 5.1: System configuration modes

	HBM	**DDR**	**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

Table 5.2: Evaluated Applications

Applications	**Description**	**# of objects**	**Footprint (GB)**
Triad [McCalpin, 1995]	Triad kernel of STREAM benchmark	3	6.00
Matrix Transpose	Transpose of a matrix is stored in another matrix	2	6.00
CG [Bailey et al., 1991]	Conjugate Gradient solves unstructured sparse linear systems	13	5.23
HotSpot [Huang et al., 2006]	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 [Che et al., 2009a]	Breath first search graph traversal algorithm	6	9.74

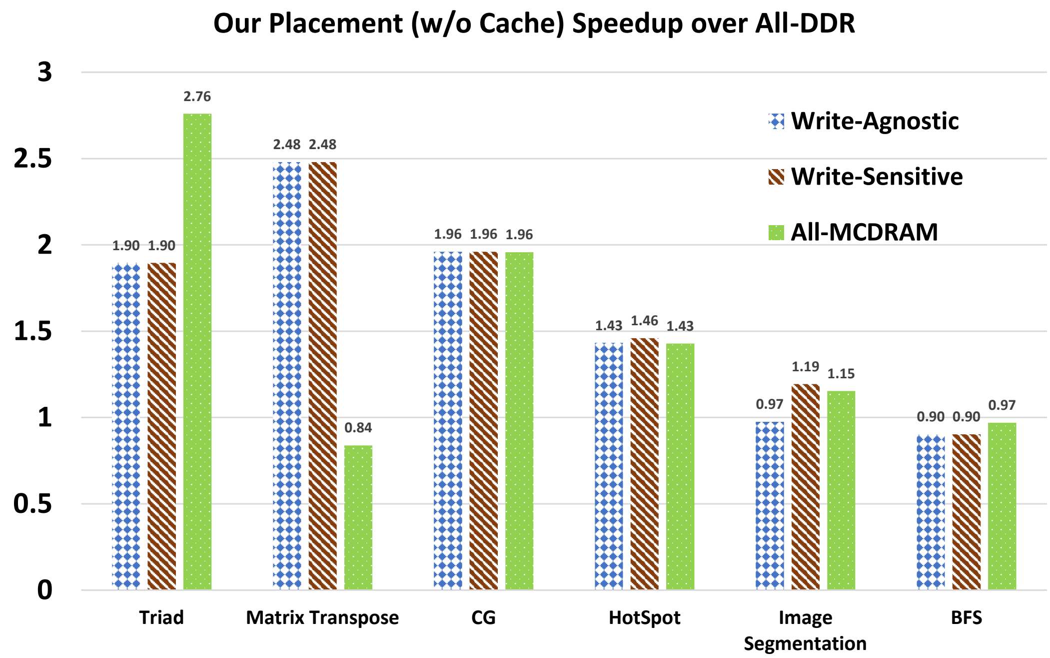

Figure 5.1: Speedup of our placement configuration achieved over placing all objects in the slow memory

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 [Asanovi et al., 2006]. 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 difference between the amount of loads and stores for majority of their objects. The three ob-

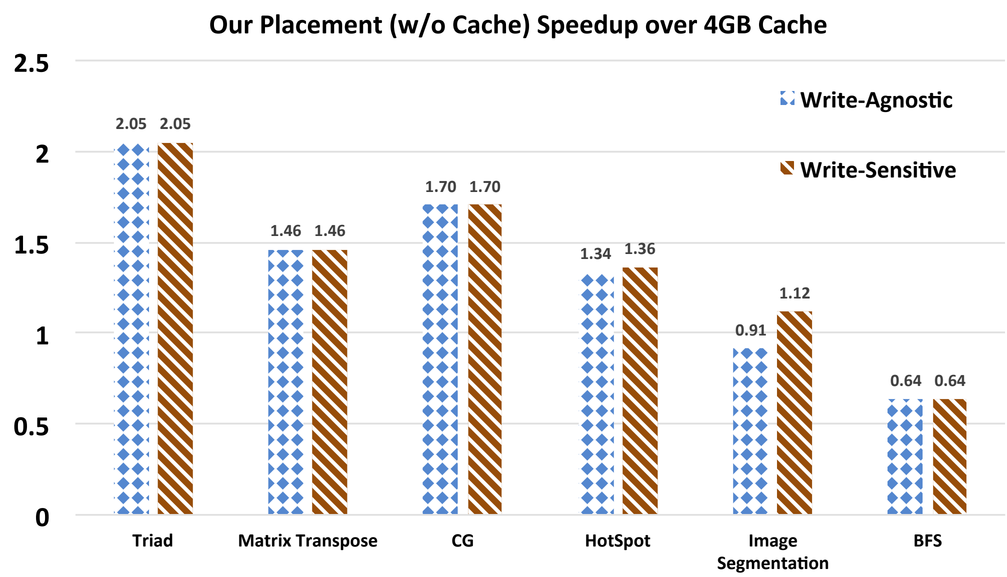

Figure 5.2: Speedup achieved over placing all objects in the slow memory coupled with 4GB MCDRAM as a last level cache.

jects 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.

5.5 Comparison against MCDRAM as a last level cache

4GB Cache

Figure 5.2 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.

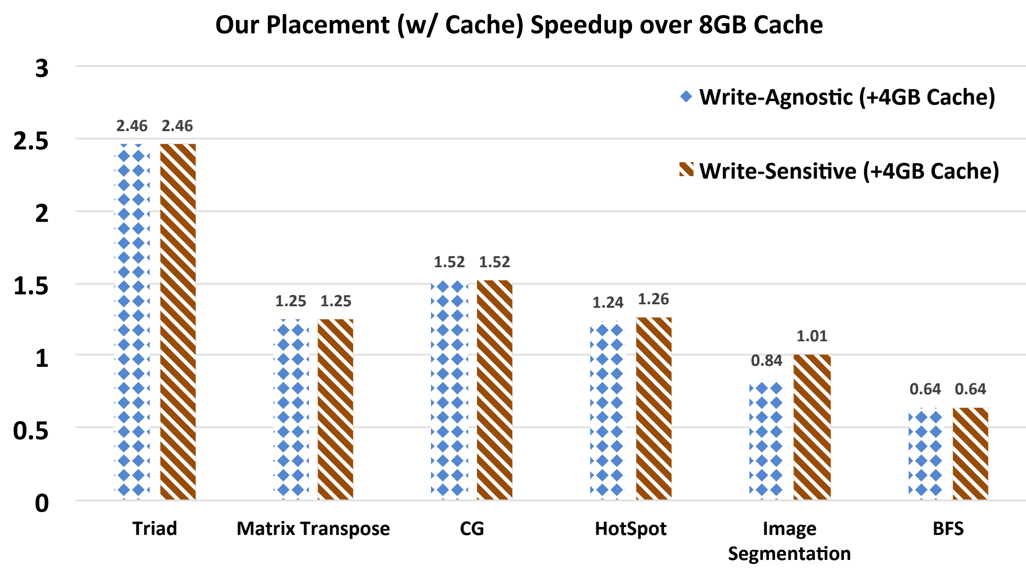

Figure 5.3: Speedup achieved over placing all objects in slow memory using 8GB MCDRAM as a last level cache. Our placement is coupled with a 4GB L3 MCDRAM cache, an addressable 4GB of MCDRAM and DDR.

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.

8GB Cache

Figure 5.3 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.

Algorithm 1 Object Placement

procedure PLACEOBJECTS(Objects, M, inFast, Access, S, α)
M[0:n-1][0:W-1] $\leftarrow$ 0
inFast[0:n-1][0:W-1] $\leftarrow$ false
Access[0:n-1] $\leftarrow$ Access counts for objects
Reads[0:n-1] $\leftarrow$ getReadAccesses(Access)
Writes[0:n-1] $\leftarrow$ getWriteAccesses(Access)
S[0:n-1] $\leftarrow$ Object Sizes
Candidates $\leftarrow \{\}$
for $i = 1...n$ **do**
for $j = 0...W$ **do**
$M_{i,j} \leftarrow M_{i-1,j}$
if $S_{i-1} \leq j$ **then**
$M_{i,j} \leftarrow \max(M_{i-1,j}, \text{Reads}_{i-1} + \alpha * \text{Writes}_{i-1} + M_{i-1,j-S_{i-1}})$
if $M_{i,j} > M_{i-1,j}$ **then**
inFast$_{i,j} \leftarrow$ true
while $n > 0$ **do**
if inFast$_{n,W}$ ==true **then**
Candidates.push(Objects$_{n-1}$.getName())
$W = W - S_n$
$n = n - 1$
return Candidates

Chapter 6

DYNAMIC PLACEMENT

6.1 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.

6.1.1 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 [Laghari and Unat, 2017]. 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.

$$Score_m = \alpha * Writes_m + \beta * Reads_m \tag{6.1}$$

6.1.2 Cost Model for Phase-Based Dynamic Placement

Phase-based cost model improves the initial placement by taking the life cycle 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 to 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 the 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:

$$AccessTime_m = \frac{Score_m * ElementType_m}{Bandwidth_{stream}} \tag{6.2}$$

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:

$$Score_m = \alpha * RemainingWrites_m + \beta * RemainingReads_m \tag{6.3}$$

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.

$$TransferCost_m = \frac{ObjectSize_m}{Bandwidth_{copy}} \tag{6.4}$$

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 [Laghari and Unat, 2017].

$$ObjFunc_m = AccessTime_m - TransferCost_m - EvictionCost_n \tag{6.5}$$

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.

6.2 Methodology

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 on 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.

6.2.1 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 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 [Chan et al., 2013] [Dulloor et al., 2016] [Shende and Malony, 2006]. Collecting object-level information through static code analysis [Unat et al., 2015] 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

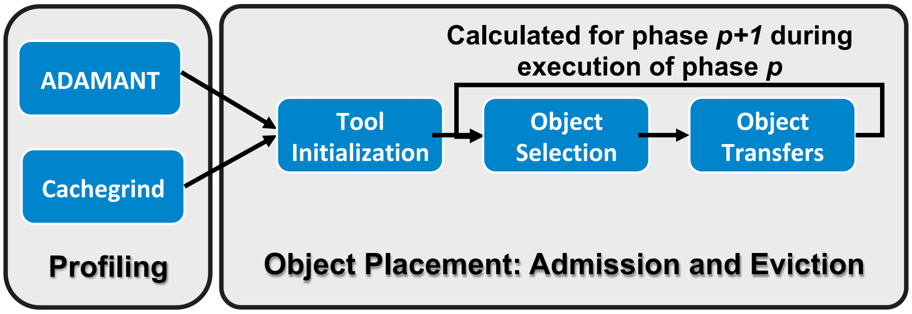

Figure 6.1: Interaction of ADAMANT and Cachegrind with our tool. ADAMANT provides the object reference counts whereas Cachegrind determines code blocks used as phases.

ADAMANT [Cicotti and Carrington, 2016] and Cachegrind [Nethercote and Seward, 2007] to gather object-level information along with the phase information.

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.

Cachegrind

is one of the tools in the Valgrind instrumentation framework [Nethercote and Seward, 2007], 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.

6.2.2 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.

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. 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.

Object Selection

We currently require the programmer to add an API call before every phase of the application which is determined using Cachegrind in the first tier. 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 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.

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.

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.

6.3 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. Table 6.1 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 6.2 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 6.1: 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

6.3.1 Comparison against All-DDR and All-MCDRAM

Figure 6.2 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 $MCDRAM$.

6.3.2 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 6.3 shows the speedup achieved by the placement conducted by our tool against hardware caching. We

Table 6.2: Evaluated Applications

Applications	**Description**	**# of Objects**	**Footprint (GB)**	**# of Phases**
CG [Bailey et al., 1991]	Conjugate Gradient solves unstructured sparse linear systems	13	5.23	17
BT [Bailey et al., 1991]	Block tri-diagonal solver	11	4.49	31
FT [Bailey et al., 1991]	Performs discrete fast Fourier Transform	5	4.50	28
LU [Bailey et al., 1991]	Lower-Upper Gauss-Seidel solver	14	4.53	30
SP [Bailey et al., 1991]	Scalar Penta-diagonal solver	10	7.76	28
SRAD [Che et al., 2009b]	Diffusion method for ultrasonic and radar imaging applications	7	5.55	4
HotSpot [Che et al., 2009b]	Thermal simulation tool used for processor temperature estimation	3	6.44	11
BFS [Che et al., 2009b]	Breadth First Search graph traversal algorithm	6	9.75	5

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 *Image Segmentation*, 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.

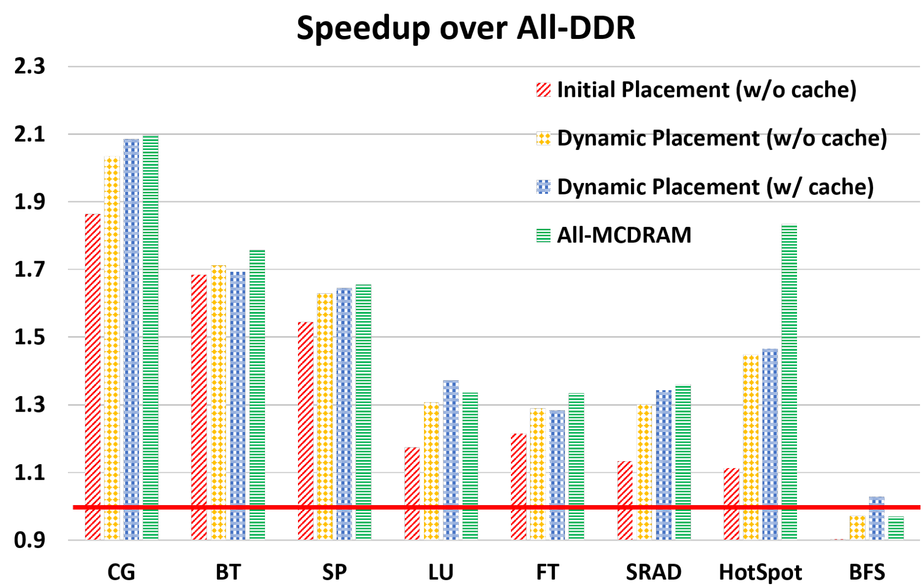

Figure 6.2: Speedup achieved by the object placement conducted by our tool against All-DDR mode. Red line shows the baseline. All values below 1 indicate degraded performance.

6.3.3 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 6.4 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 computations. 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 6.3 shows the number of phases in which a transfer occurs and how many

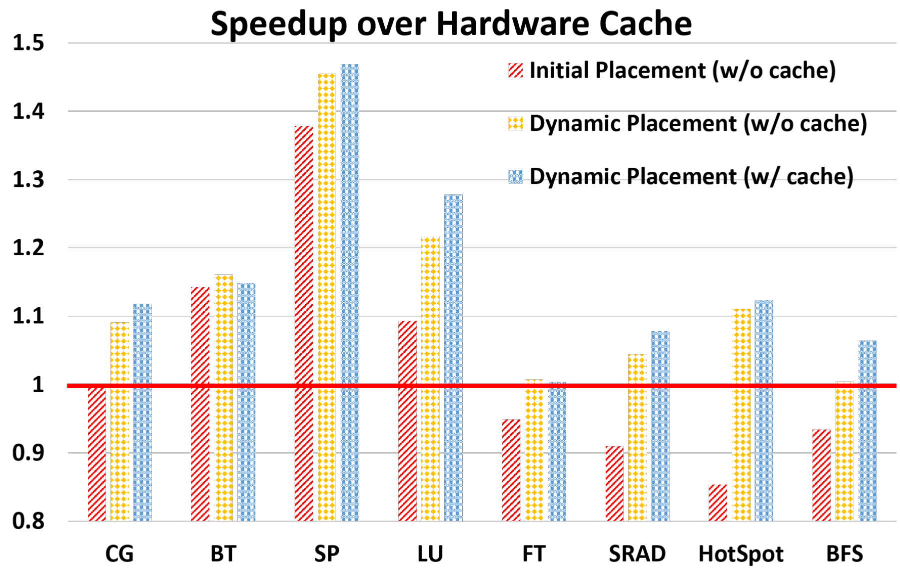

Figure 6.3: Speedup achieved by our tool against MCDRAM acting as LLC. Red line shows the baseline. All values below 1 indicate degraded performance.

objects are 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 total 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 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.

6.3.4 Latency-Sensitive Applications

While some applications benefit from the high bandwidth characteristics of an HBM, others might get degraded performance because of higher latency of HBM compared to DRAM. Applications which include indirect or strided access can be characterized as latency-sensitive applications. Figure 6.5 shows the bandwidth degradation of MCDRAM and DDR in KNL. As the stride size increases, the bandwidth of both

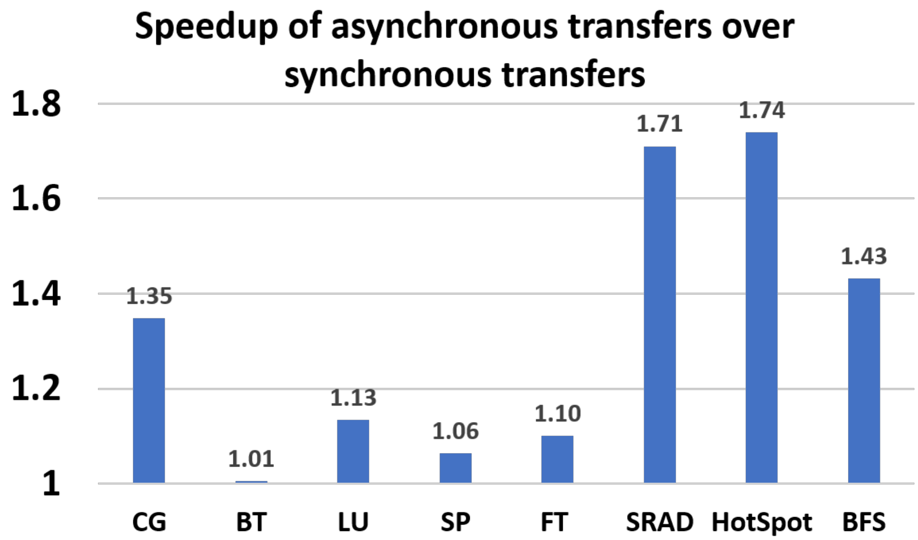

Figure 6.4: Speedup achieved by asynchronous transfers over synchronous transfers using dynamic placement strategy.

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 improvement but we expect the benefit is higher for an application which has a lot more objects with mixed access patterns.

Table 6.3: 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%

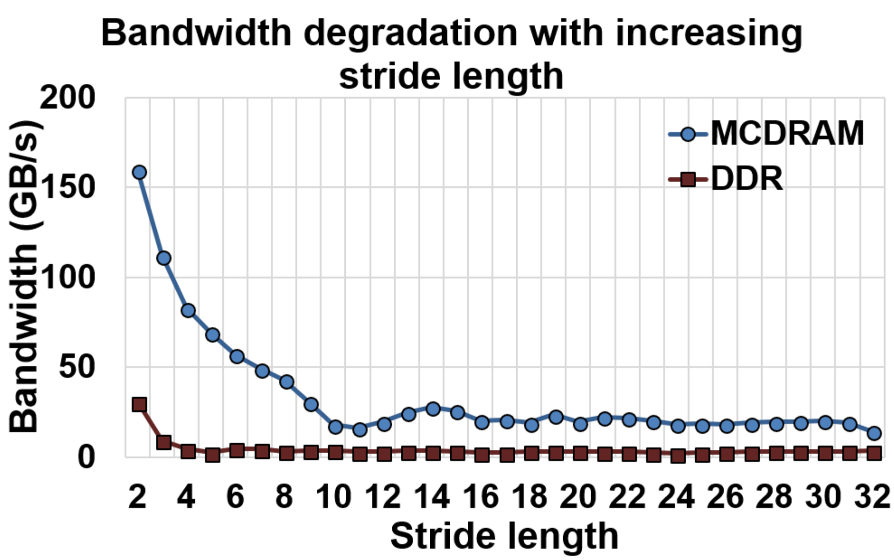

Figure 6.5: Bandwidth degradation comparison of MCDRAM and DDR in Intel KNL as the stride length is increased.

Chapter 7

CONCLUSION

In this work we explore the benefits of heterogeneous memory systems. Such systems are equipped with multiple memories both having varying characteristics. An HM system typically has a high bandwidth memory which, as its name suggests, has a considerably higher bandwidth than the other memory. The HBM in an HM system can have differing latency characteristics. The HBM is usually low in capacity therefore a secondary higher capacity memory is required to complement the HBM. The high capacity memory has a lower bandwidth than the HBM. Multiple memories introduce the need for efficient and intelligent object allocation to reach the maximum potential of the HM system. To address this challenge, this thesis proposes two schemes **1) Initial Placement** which considers object memory references and its size in the application and based on these statistics, it suggests an object allocation strategy to the programmer. Objects, in this scheme, are allocated statically prior to application execution and are not changed throughout application execution. The second scheme is **2) Dynamic Placement** which considers object memory accesses, sizes of objects, and phases in an application. Based on a cost model, it suggests a placement scheme of objects in HBM. Moreover, data transfers between the phases are performed asynchronously to hide the transfer overhead. 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.

BIBLIOGRAPHY

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

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

[Bailey et al., 1991] Bailey, D. H., Barszcz, E., Barton, J. T., Browning, D. S., Carter, R. L., Dagum, L., Fatoohi, R. A., Frederickson, P. O., Lasinski, T. A., Schreiber, R. S., Simon, H. D., Venkatakrishnan, V., and Weeratunga, S. K. (1991). 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. ACM.

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

[Chan et al., 2013] Chan, C., Unat, D., Lijewski, M., Zhang, W., Bell, J., and Shalf, J. (2013). Software design space exploration for exascale combustion co-design.

In Kunkel, J. M., Ludwig, T., and Meuer, H. W., editors, *Supercomputing*, pages 196–212, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Che et al., 2009a] Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J. W., Lee, S.-H., and Skadron, K. (2009a). 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. IEEE Computer Society.

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

[Chou et al., 2014] Chou, C., Jaleel, A., and Qureshi, M. K. (2014). 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. IEEE Computer Society.

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

[Dulloor et al., 2016] Dulloor, S. R., Roy, A., Zhao, Z., Sundaram, N., Satish, N., Sankaran, R., Jackson, J., and Schwan, K. (2016). 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. ACM.

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

[Islam et al., 2016] Islam, M., Banerjee, S., Meswani, M., and Kavi, K. (2016). 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. ACM.

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

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

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

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

[Laghari and Unat, 2017] Laghari, M. and Unat, D. (2017). 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.

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

[Nethercote and Seward, 2007] Nethercote, N. and Seward, J. (2007). 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. ACM.

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

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

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

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

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

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

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

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

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

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

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

[Wu et al., 2017] Wu, K., Huang, Y., and Li, D. (2017). 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. ACM.