MOCA: Memory Object Classification and Allocation in Heterogeneous Memory Systems

Aditya Narayan*, Tiansheng Zhang*, Shaizeen Aga†, Satish Narayanasamy† and Ayse K. Coskun*
*Boston University, Boston, MA 02215, USA; Email: {adityan, tszhang, acoskun}@bu.edu
†University of Michigan, Ann Arbor, MI 48109, USA; Email: {shaizeen, nsatish}@umich.edu

***Abstract*—In the era of abundant-data computing, main memory's latency and power significantly impact overall system performance and power. Today's computing systems are typically composed of homogeneous memory modules, which are optimized to provide either low latency, high bandwidth, or low power. Such memory modules do not cater to a wide range of applications with diverse memory access behavior. Thus, heterogeneous memory systems, which include several memory modules with distinct performance and power characteristics, are becoming promising alternatives. In such a system, allocating applications to their best-fitting memory modules improves system performance and energy efficiency. However, such an approach still leaves the full potential of heterogeneous memory systems under-utilized because not only applications, but also the memory objects within that application differ in their memory access behavior.**

This paper proposes a novel page allocation approach to utilize heterogeneous memory systems at the memory object level. We design a *memory object classification and allocation framework (MOCA)* to characterize memory objects and then allocate them to their best-fit memory module to improve performance and energy efficiency. We experiment with heterogeneous memory systems that are composed of a Reduced Latency DRAM (RLDRAM) for latency-sensitive objects, a 2.5D-stacked High Bandwidth Memory (HBM) for bandwidth-sensitive objects, and a Low Power DDR (LPDDR) for non-memory-intensive objects. The MOCA framework includes detailed application profiling, a classification mechanism, and an allocation policy to place memory objects. Compared to a system with homogeneous memory modules, we demonstrate that heterogeneous memory systems with MOCA improve memory system energy efficiency by up to 63%. Compared to a heterogeneous memory system with only application-level page allocation, MOCA achieves a 26% memory performance and a 33% energy efficiency improvement for multi-program workloads.

***Index Terms*—heterogeneous memory, energy efficiency, memory management**

I. Introduction

The *memory wall*, prevalent over the last three decades, has been widening the gap between memory access and computation speeds [1]. With the recent advent of memory-intensive applications, main memory performance and power play an even more significant role in overall system energy efficiency in modern servers and high-end supercomputers. Increasing the number of processing cores on a single chip adds more pressure on the memory system, thereby limiting system performance improvement. Recent studies show that main memory consumes more than 25% of the total system power for servers in data centers [2]. Thus, improving memory performance and energy efficiency is crucial for achieving higher system efficiency.

Traditionally, the main memory (e.g., DRAM) of a computing system is composed of a set of homogeneous memory modules. The key attributes that determine the efficiency of a memory system are latency, bandwidth, and power. An ideal memory system should provide the highest data bandwidth at the lowest latency with minimum power consumption. However, there is no such perfect memory system as a memory module with high performance generally has a high power density. Hence, memory modules come in different flavors, optimized to either improve system performance or reduce power consumption.

Due to diverse memory access behavior across workloads, a memory system with homogeneous memory modules is often not sufficient to meet all workloads' memory requirements. Motivated by performance-power tradeoffs among various memory modules for diverse workloads, heterogeneous memory systems have been proposed to improve performance and energy efficiency for a variety of systems, from embedded systems to modern servers (e.g., [3]–[9]). A heterogeneous memory system is composed of several memory modules with different performance and power characteristics. Such a system can be realized through multiple memory controllers, which can synchronize the accesses [10]. There are already commercial products that utilize heterogeneous memory. For example, Intel's Knights Landing processor has an on-chip high-bandwidth memory (HBM) together with an off-chip DDR4 [5]. In GPUs, scratchpad memory is widely used with off-chip DRAM in order to accelerate memory accesses without losing capacity. As another example, AMD Radeon R9 Fury X, in addition to its off-chip DDR3, also consists of an interposer with an HBM stack for higher bandwidth [11].

To leverage the benefits of such a system, we need a systematic memory management scheme for heterogeneous memory systems. A heterogeneity-aware page allocation policy should ideally map a page to a memory module that is best suited to that page's access characteristics. Prior work proposes allocating memory pages based on the aggregate memory access behavior of each application [3]. Another approach allocates critical words in a cache line to a latency-optimized memory module [4]. However, such existing application-level policies do not harness the heterogeneity in memory behavior that exists within an application.

In this work, we observe that *an application's memory objects, particularly those allocated dynamically in the heap space, exhibit vastly diverse memory access behavior, and* *such behavior often significantly differs from the application's aggregate memory access behavior.*. To exploit this observation, we propose a framework that enables allocating memory objects of an application to different memory modules based on their access behavior. We refer to our framework as *memory object classification and allocation (MOCA)*. MOCA includes a profiler that names, profiles, and classifies memory objects within an application based on their memory access behavior as well as an allocation scheme at the OS level.

We focus particularly on heap objects as these often substantially contribute to the performance or energy bottlenecks of a memory-intensive application. We *name* a heap object based on the program site (where the object is instantiated) and the calling context where it is allocated. Our profiler then classifies each named object based on its memory access behavior observed over a representative set of profiled executions. Then, the classification is stored as part of the application binary. At runtime, heap's virtual memory is partitioned into as many types as the number of memory modules in the system (three in our study), and MOCA's memory allocator allocates a heap object to one of those partitions based on its type. The OS is responsible for allocating pages from a particular virtual memory heap partition to physical pages in memory device with the desired characteristics.

Our specific contributions are listed as follows:

- As part of MOCA, we develop a technique to uniquely name each heap object instantiated during application execution and profile its memory accesses. MOCA then classifies memory objects based on the profiling results.
- Again, in MOCA, we develop a runtime page allocation mechanism that allocates each memory object to its best-fitting memory module based on the memory object classification results.
- We provide a detailed implementation of MOCA and demonstrate the benefits of object-level page allocation in a heterogeneous memory system consisting of low power DDR (LPDDR), reduced latency DRAM (RLDRAM), and HBM modules. MOCA improves memory system energy efficiency by up to 63% for multi-program workloads over a baseline homogeneous DDR3 system. On average, it increases memory performance by 26% and improves energy efficiency by 33% compared to an application-level page allocation policy [3] for multi-program workloads with diverse memory objects.

The rest of the paper starts with the motivation for heterogeneous memory systems and object-level page allocation in Sec. II. Section III describes our proposed framework, MOCA. Section IV details MOCA implementation in a real-system and Sec. V and VI present our simulation framework and results. We review related work in Sec. VII and conclude in Sec. VIII.

II. Motivation

A. Heterogeneous Memory Systems

Memory vendors provide memory modules with various performance and power characteristics, targeting a wide range of system requirements. For example, RLDRAM is a memory type optimized for low access latency, which makes it ideal for switch and router applications [14]. It is implemented similar to SRAM, where the entire address is provided in the same clock cycle, thereby reducing the access latency significantly. However, the static and dynamic power consumption of RLDRAM is 4-5x higher than a DDR3/DDR4 module, and the bandwidth is lower. On the other hand, LPDDR reduces power consumption substantially, but has higher access latency and lower bandwidth; thus, it is attractive for mobile platforms. HBM is an innovative memory technology which stacks multiple DRAM layers vertically, where layers are connected by through-silicon-vias. An HBM boasts of more channels per device, smaller page sizes per bank, wider activation windows and a dual command line for simultaneous read and write [15]. These features distinguish HBMs to provide performance and power improvements in case of bandwidth-sensitive workloads. However, there is no single memory module that can provide the lowest latency, highest bandwidth, and lowest power consumption at the same time. Thus, we argue that homogeneous memory systems are often not sufficient in an era of diverse computation- and memory-intensive workloads.

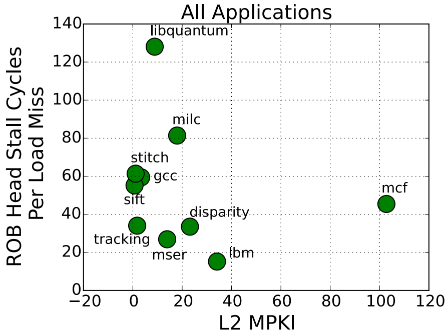


Fig. 1: Memory access behavior of selected applications from SPEC CPU2006 [12] and SDVBS [13] benchmarks. A high L2 MPKI indicates that the application is memory-intensive. A low number of ROB Stall cycles for a memory-intensive application implies high memory-level parallelism.

Another drawback of homogeneous memory systems is their incognizance to the workload running on the system. Workloads differ widely in their memory access behavior as shown in Fig. 1, making homogeneous memory systems sub-optimal in terms of energy efficiency and performance. This figure shows the diverse memory access behavior for a set of applications from SPEC CPU2006 [12] and San Diego Vision Benchmark Suite (SDVBS) [13]. We plot the last-level cache misses per kilo instructions (LLC MPKI), which quantifies memory intensiveness, and the Reorder Buffer (ROB) head stall cycles per load miss, which is a metric for memory level parallelism (MLP) [16]. MLP is the notion of issuing and servicing multiple memory requests in parallel. We provide more details about the chosen metrics in Sec. III.

Phadke *et al.* [3] introduce an application-level allocation policy for heterogeneous memory systems. They profile the memory access behavior of every application as a whole and

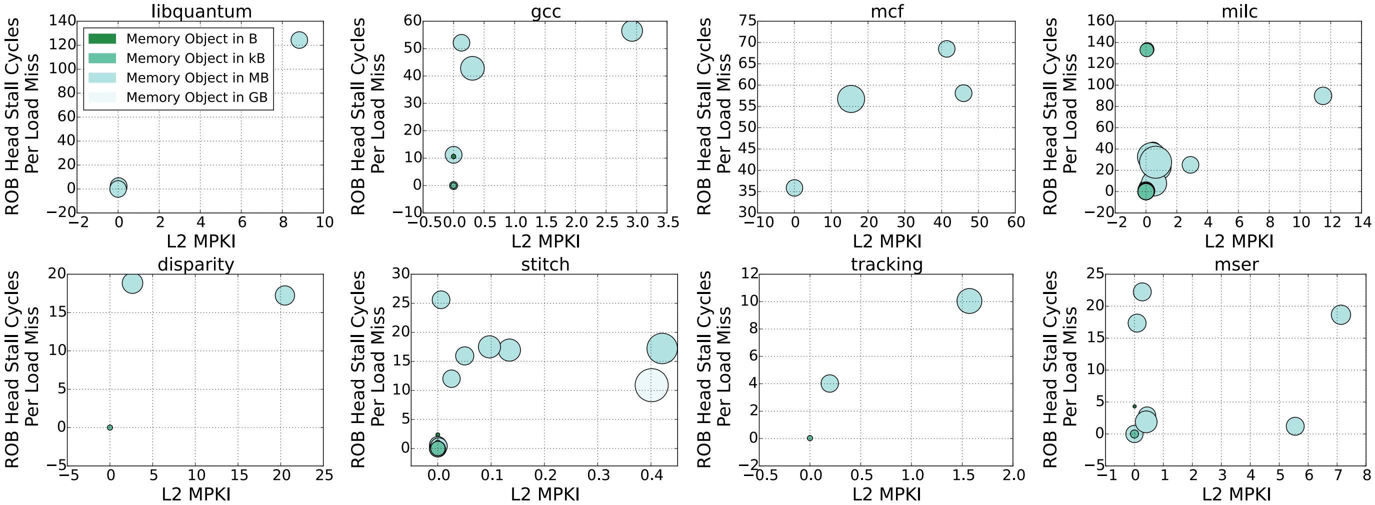


Fig. 2: **Memory access behavior of memory objects for selected SPEC CPU2006 [12] and SDVBS [13] applications. The x-axis plots L2 MPKI and the y-axis plots ROB head stall cycles per load miss. The size of a circle indicates the size of that object.**

allocate the entire application to the best-fit memory module. For example, a computation-intensive workload such as *gcc* has low L2 MPKI. It achieves similar performance using any type of memory module but the overall power consumption can be reduced using LPDDR. On the other hand, a memory-intensive workload (e.g., *mcf* or *milc*) can achieve substantially higher performance using memory modules with low access latency or high bandwidth. Thus, a heterogeneous memory system, which contains different memory modules, is able to cater to a wide range of applications and provide higher energy efficiency [3], [4]. Yet, achieving higher energy efficiency is contingent upon placing an application's data in the right memory module.

B. Exploiting Heterogeneity at a Finer Granularity

Operating at a coarser granularity, *application-level* profiling and page allocation often do not fully utilize the performance and energy benefits of heterogeneous memory systems. This is because not all memory objects within an application display similar memory access behavior. Many applications are composed of a number of memory objects with significant variations in their memory accesses. Objects within an application are often accessed at different frequencies, have different memory sizes, or exhibit different levels of MLP. We claim that it is beneficial to profile applications and allocate pages at the granularity of memory objects.

Figure 2 shows the distribution of memory objects within selected applications from the SPEC CPU2006 and SDVBS benchmark suites. Similar to application-level results, we use L2 MPKI and ROB stall cycles per load miss to determine whether a memory object is latency-sensitive, bandwidth-sensitive, or neither. We see a wide distribution across both of these metrics for objects within the same application. Memory-intensive applications such as *milc* and *mser* have only a few memory objects with high L2 MPKI. In contrast to an application-level allocation, which would place all the objects into an RLDRAM module for these applications, a finer-level allocation could place the objects with low L2 MPKI into an LPDDR module, thereby reducing the overall power consumption.

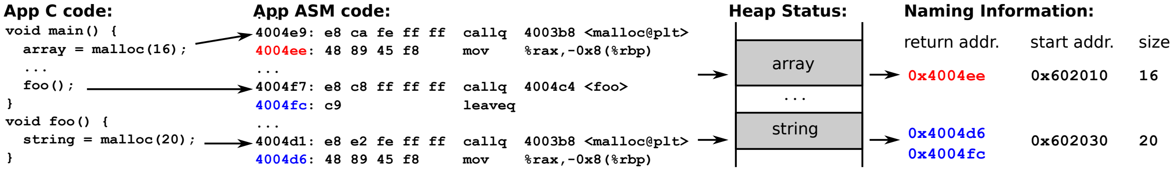
In order to tap into this heterogeneity in memory access behavior of objects within an application, we design an object-level page allocation policy. This policy, which we describe next, places each object into the best-fitting memory module in a given heterogeneous memory system.

III. MOCA: Memory Object Classification and Allocation

To leverage the advantages of heterogeneous memory systems for performance and energy efficiency improvement, we propose the MOCA framework. MOCA consists of an offline profiling stage that uniquely names every memory object allocated in the heap memory space[1] through a consistent naming convention. It then collects statistics of metrics that characterize the memory access behavior of each named memory object within every application. Using this information, MOCA classifies memory objects based on their sensitivity to memory access latency and memory bandwidth. These steps of profiling and classification are conducted offline. Then, during runtime execution of an application, MOCA runs a method to allocate each memory object to the best-fitting memory module, based on the memory object classification. Figure 4 shows the flow of MOCA.

Our work targets applications that run repeatedly on a given system. Many such applications exist in mobile systems, PCs, data centers, or supercomputers. We use representative training inputs during profiling and test with other reference inputs. Such profiling-based approaches work well for applications with fairly similar behavior across different input sets (e.g., [17], [18]).

[1] An application's memory use mainly includes stack, heap, and code. Stack and code segments often have low LLC MPKI (i.e., they tend to utilize the caches well) and have minimal impact on performance when placed on different memory modules. Therefore, our work focuses on the heap usage.


App C code:

```
void main() {
  array = malloc(16);
  ...
  foo();
}
void foo() {
  string = malloc(20);
}
```

App ASM code:

```
4004e9: e8 ca fe ff ff   callq  4003b8 <malloc@plt>
4004ee: 48 89 45 f8      mov    %rax,-0x8(%rbp)
...
4004f7: e8 c8 ff ff ff   callq  4004c4 <foo>
4004fc: c9               leaveq
...
4004d1: e8 e2 fe ff ff   callq  4003b8 <malloc@plt>
4004d6: 48 89 45 f8      mov    %rax,-0x8(%rbp)
```

Naming Information:

return addr.	start addr.	size
0x4004ee	0x602010	16
0x4004d6 0x4004fc	0x602030	20

Fig. 3: An example of memory object naming convention.

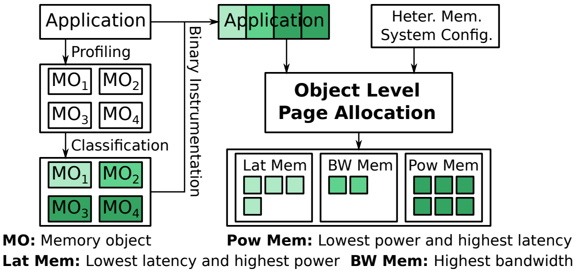

Application
Profiling
MO1 MO2
MO3 MO4
Classification
MO1 MO2
MO3 MO4
Binary Instrumentation
Application
Heter. Mem. System Config.
Object Level Page Allocation
Lat Mem
BW Mem
Pow Mem
MO: Memory object
Pow Mem: Lowest power and highest latency
Lat Mem: Lowest latency and highest power
BW Mem: Highest bandwidth

Fig. 4: The workflow of MOCA. The profiling stage uniquely names memory objects and profiles their memory access behavior. Classification stage uses this information to classify objects. At runtime, each memory object is allocated with pages from the best-fitting memory module based on object's type.

A. Memory Object Profiling

The profiling stage uniquely names memory objects and collects statistics for each of them. To name memory objects, we use the return address of each dynamic memory allocation function (e.g., malloc, calloc, etc. in C) and record the virtual address of its caller function in the stack. These two addresses are unique to every object. In addition, we also record the size of each object. Our naming convention for an example C code is shown in Fig. 3. In the figure, the memory object *array* is instantiated directly from the main function. So, the object is named with the return address of the corresponding malloc function and start address of the caller function. The memory object *string*, being instantiated inside a function *foo*, is named with the return address of its malloc function inside *foo* and the return address of *foo* in the main function. This naming convention enables us to distinguish memory objects instantiated via the same function, even when the function is invoked from different locations in a program.

Once we name all memory objects within an application, we need metrics to characterize their memory access behavior. We record the LLC MPKI for each object as LLC MPKI provides an indication of the memory access intensity. In addition, we collect average ROB head stall cycles per load miss [16] for each object. ROB head stall time is computed as the average cycles spent waiting at the head of the ROB for load misses. This metric has been identified and used as an effective measure for MLP in prior work [3], [16].

B. Memory Object Classification

In classification stage, we use the collected statistics from profiling (memory objects, their LLC MPKI and ROB head stall times) to classify objects as being either latency-sensitive, bandwidth-sensitive, or neither. Memory objects with high LLC MPKI are generally memory-intensive. Among high-LLC-MPKI memory objects, the ones exhibiting low ROB head stall time have high MLP (memory latencies are largely hidden as indicated by low ROB stalls), i.e., such objects benefit from a high-bandwidth memory module. We classify such objects as bandwidth-sensitive. The rest of the memory-intensive objects with low MLP (high ROB head stall time) are sensitive to access latency of memory modules (i.e., higher latency leads to larger ROB stall time) and we classify them as latency-sensitive objects. Objects with low LLC MPKI are not sensitive to either latency or bandwidth. Such objects can be placed in low-power memory modules without affecting performance, thereby reducing memory power consumption.

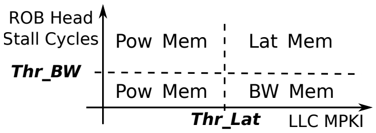


Fig. 5: Classification of memory objects based on thresholds

Figure 5 depicts this classification where *Lat Mem* is a *RLDRAM* module, *Pow Mem* is an *LPDDR* module and *BW Mem* is an *HBM* module. We classify objects with LLC MPKI greater than the latency threshold (*Thr_Lat*) as memory-intensive. The rest of the objects do not access memory frequently and can be safely allocated to *Pow Mem*. For memory-intensive objects, we classify the ones with ROB head stalls greater than the bandwidth threshold (*Thr_BW*) as being latency-sensitive due to the lack of MLP and objects with ROB head stalls lower than *Thr_BW* are classified as bandwidth-sensitive and allocated to *BW Mem*. In MOCA, we empirically set the latency and bandwidth sensitivity thresholds based on the energy/performance characteristics of the memory modules in the given system (see Sec. IV-C).

C. Page Allocation for Memory Objects

The offline profiling and classification stages collectively provide the name and characteristics of every memory object in an application. We instrument the memory object classification information into application binaries. Specifically, we modify the standard memory allocation functions (e.g., malloc) to enable specifying object "types": bandwidth-/latency-sensitive or neither. MOCA splits the heap memory space accordingly with these types (see Fig. 6). The physical memory address space is also divided based on the available memory module types in the heterogeneous memory system.

At runtime, MOCA uses the object-level information to perform page allocation. When a memory object is instantiated through MOCA's modified memory allocator (including the extra arguments of the object types), that object is allocated

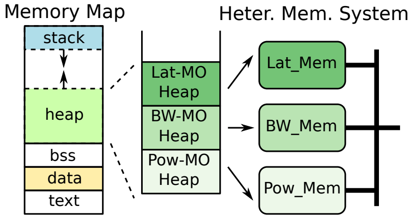


Fig. 6: Virtual and physical memory space separation for MOCA support in real systems

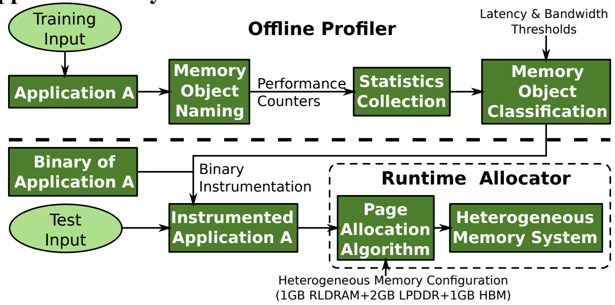

Training Input
Offline Profiler
Latency & Bandwidth Thresholds
Application A
Memory Object Naming
Performance Counters
Statistics Collection
Memory Object Classification
Binary of Application A
Binary Instrumentation
Runtime Allocator
Test Input
Instrumented Application A
Page Allocation Algorithm
Heterogeneous Memory System
Heterogeneous Memory Configuration
(1GB RLDRAM+2GB LPDDR+1GB HBM)

Fig. 7: Real-system flow of MOCA framework

with virtual pages from the heap space based on its type. For example, an object with high LLC MPKI and low ROB stalls gets placed into the bandwidth-sensitive heap. In the page translation process, based on the memory object's virtual page number, the OS identifies the type of the memory object and maps a physical frame from the memory module corresponding to its type, as shown in Fig. 6.

If there is enough space in the best-fit memory module, the memory object gets the physical pages from this memory module. If the best-fitting memory module capacity is exhausted, MOCA proceeds to the next best memory module (e.g., next best for HBM is LPDDR, etc.) and continues allocation until all objects get physical pages.

IV. Implementation

Section III has discussed the operation of MOCA. This section provides implementation details of MOCA in a real system. Figure 7 depicts the real-system implementation flow.

A. Memory Object Profiler (Offline)

As introduced in Sec. III-A, we use the return addresses of the memory allocation functions and addresses of their caller functions to uniquely name memory objects. To implement the naming process, we modify the memory allocation functions to get the return addresses of each memory allocation function and its caller function using a built-in function *__builtin_return_address()*. We create a shared library of the modified memory allocation functions and preload this library while executing an application. We add a profiler flag to our compiler to maintain all the objects within an application in a lookup table (LUT). This LUT contains all the information of every object (call stack, size, start address, LLC MPKI, ROB head stall cycles per load miss).

B. Statistics Collection and Object Classification (Offline)

MOCA uses the hardware performance counters of the processor to record the LLC misses and the ROB head stall cycles for each memory object. Each time an object is read/written to, if the ROB stalls for a memory read or if there is an LLC miss, we identify the accessed memory object (based on the requested address) and increment the corresponding counter for that memory object in the LUT. We also update the object's size as needed.

C. Classification Threshold Setup (Offline)

In MOCA, we empirically set the *Thr_Lat* and *Thr_BW* that are used in classifying the memory objects. While experimenting with our profiled applications, we search for the lowest LLC MPKI value of a memory object where placing that object in a RLDRAM module results in an overall memory energy efficiency improvement. We set this LLC MPKI value as *Thr_Lat*. Similarly, *Thr_BW* is the highest ROB stall time value of a memory object that gives memory energy efficiency improvements for the given system when this object is placed in an HBM module.

For our target heterogeneous system, we set *Thr_Lat* as 1 and *Thr_BW* as 20. In a similar fashion, one can perform a sensitivity analysis in a given heterogeneous memory system to identify the breakeven points where the RLDRAM and HBM (or another available module) starts giving energy efficiency improvements. *Thr_Lat* and *Thr_BW* need to be customized for a given system, as memory, cache, and core microarchitecture parameters significantly impact performance and energy efficiency.

D. Page Allocator (Runtime)

MOCA's runtime page allocation algorithm runs on top of the existing OS memory management. As noted earlier, we use the classification information of objects to instrument an application binary with a "type" for each of its heap objects. The OS, upon encountering a heap object, then knows from which memory type it should allocate that object's pages[2].

When the CPU issues a memory request, it goes to the L1 cache and, at the same time, the CPU searches translation lookaside buffer (TLB) for the physical page number of this memory request. On a hit, the TLB sends the physical page corresponding to the requested virtual page. Otherwise, there is a page fault, followed by a page walk, which searches through the OS-maintained page table to find the required virtual-physical page translation. Then, the requested page table entry (PTE) is returned and inserted into TLB. The OS maintains the starting, ending, and the next available page number of each memory module in a heterogeneous memory system. Whenever a PTE returns from a page table walk, MOCA determines the physical page number for the next available page from the desired memory module of the corresponding memory object. The OS is also given the priorities of memory modules for different memory object types in case the most desired memory module is full (i.e., next best module if the ideal one is full).

[2] Alternatively, one could instrument the application binary with object statistics (LLC MPKI and ROB stalls) and pass the *Thr_Lat* and *Thr_BW* thresholds to the OS.

TABLE I: Microarchitectural details of simulated system

Execution Core	1GHz x86 ISA with out-of-order execution Fetch/Decode/Dispatch/Issue/Commit width 3, 84-entry ROB, 32-entry LQ, tournament branch predictor with 4K BTB entries
On-chip caches	64KB split L1 I and D cache, 2-way, 2 cycle , 64B line size, 4 MSHR Unified L2, 512KB, 16-way, 20 cycles, 64B line size, 20 MSHR
Memory Controller	Address mapping RoRaBaChCo, 4 channels, FR-FCFS scheduling

As for the virtual pages that do not belong to heap objects, we assign them physical pages from the LPDDR and update the page table accordingly.

E. Overhead of MOCA

The profiling and classification of memory objects are conducted offline and do not impact system performance at runtime. The profiler needs to register every memory object and update its statistics in the LUT. We measure the performance overhead of running our applications with profiling turned on, and observe only 0.59% slowdown on average.

In our page allocation algorithm, the performance overhead comes from the OS selecting the best-fit memory module at runtime for every memory object. Since the OS needs to perform allocations for objects only at their instantiation, this overhead is negligible. Note that, in contrast to page migration policies that need to monitor runtime information, MOCA only slightly modifies the page allocation method in the OS [19].

V. Experimental Methodology

We present the details of our simulation framework, the baseline and target memory systems, as well as the workload sets in this section.

A. Simulation Framework

To demonstrate the benefits of heterogeneous memory systems, we conduct experiments with both a single-core system and a multicore system with four cores. We model each core based on the core architecture of AMD Magny-Cours processor [20]. The microarchitectural parameters are listed in Table I. We modify Gem5 [21] as explained in Sec. IV and use it for full-system architectural simulations. We use a Linux 2.6.32 disk image as the host operating system. For each application, we generate simpoints [22] for the training input data. We fast-forward to these simpoints and run the applications for 100 million instructions to collect memory object statistics for each application, and then take a weighted value of metrics at these simpoints to perform memory object classification. Memory objects instantiated during both the fast-forward phase and the execution phase are all recorded for profiling and classification purpose. We consider five levels of return addresses in our callstack for naming memory objects.

We use the reference input set for the comparison among different page allocation techniques. For each workload set, we fast-forward to five billion instructions and run for one billion instructions. We feed the Gem5 output statistics to McPAT [23] for core and cache power calculation. For better simulation accuracy, we calibrate the runtime dynamic core power values using measurements collected on the AMD Magny-Cours processor. We calculate the dynamic core power from power simulation across workloads and calculate the calibration factor to scale McPAT raw data to target power scale. Such calibration approaches have been performed in prior work [24]. For our multicore system, we observe an average total core power of 21W. We model performance characteristics of our memory system designs in Gem5 and use MICRON's DRAM power calculators for DDR3, RLDRAM and LPDDR2 [25]–[27] to calculate memory power consumption. This calculator takes in memory read and write access rates as inputs and provides detailed DRAM power traces for each banks. For HBM, we scale down the DDR3 precharge and power-down current [28], and then estimate memory power from SDRAM power calculator [25]. We assume that the I/O power and the on-chip bus power are negligible compared to total chip power.

TABLE II: Timing and architectural parameters for various memory modules used in this work

	DDR3 [25]	HBM [25], [28]	$RLDRAM$ [27]	$LPDDR2$ [26]
Burst length	8	4	8	4
# of banks	8	8	16	8
Row buffer size	128B	$2kB$	$16B$	$1kB$
# of rows	$32K$	$32K$	$8K$	$8K$
Device width	8	128	8	32
$t_{CK}(ns)$	1.07	2	0.93	1.875
$t_{RAS}(ns)$	35	33	6	42
$t_{RCD}(ns)$	13.75	15	2	15
$t_{RC}(ns)$	48.75	48	8	60
$t_{RFC}(ns)$	160	160	110	130
Standby Power(mW/GB)	256	335	30	6.5
Active Power(W/GB)	1.5	4.5	1.1	0.4

B. Homogeneous Memory System

We have a system with 2GB DDR3 module as the baseline for homogeneous memory systems, as most high-end servers employ this memory type. We also test three other homogeneous memory systems comprising of 2GB LPDDR2, 2GB RLDRAM, and 2GB HBM, respectively. Each of the four memory controllers on the processor is connected to 512MB memory. The architectural, timing and power parameters of these memory modules are shown in Table II.

C. Heterogeneous Memory System

Our target heterogeneous memory system consists of four memory channels and each channel is connected to a type of memory module. We model this memory system to consist of a 768MB HBM module, a 256MB RLDRAM module, and two 512MB LPDDR2 modules. We use a dedicated memory controller for each memory channel as the device timing parameters differ for different memory modules. The HBM and RLDRAM modules are connected to one memory controller each on the processor and we employ two memory controllers to connect two 512MB LPDDR2 modules separately.

We compare our proposed object-level page allocation in heterogeneous memory system with an application-level allocation [3], where all the memory objects in one application are

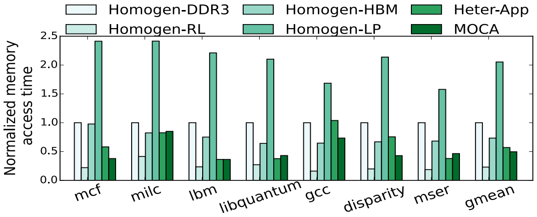


Fig. 8: Memory performance of homogeneous and heterogeneous memory systems for single workloads

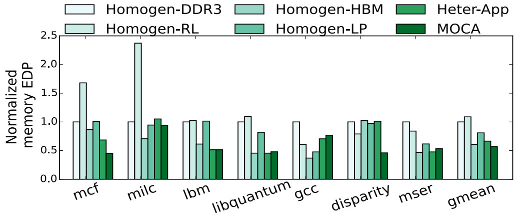


Fig. 9: Memory EDP of homogeneous and heterogeneous memory systems for single workloads

TABLE III: Benchmarks Classification

L (latency sensitive)	mcf, milc, libquantum, disparity
B (bandwidth sensitive)	mser, lbm, tracking
N (non-memory intensive)	gcc, sift, stitch

allocated to that application's best-fit memory module. When there are no pages left in the best-fit module, the objects are then allocated to this application's next-best memory module.

D. Workloads

We run selected C-based applications from SPEC CPU2006 [12] and SDVBS [13]. For SPEC benchmarks, we conduct profiling using the training input sets and perform allocation on reference input sets. In case of SDVBS benchmarks, we select two different images from MIT-Adobe fivek dataset [29] for profiling and allocation. The applications from these benchmark suites are chosen such that we cover a wide spectrum of LLC MPKI and ROB stall time. As shown in Table III , we categorize the applications as latency-sensitive (L), bandwidth-sensitive (B) and non-memory-intensive (N) for application-level allocation. To run workloads on a multicore system, we create workload sets consisting of a diverse mix of these applications. E.g., *2L1B1N* represent a workload set with two latency-sensitive applications, one bandwidth-sensitive application and one non-memory-intensive application. 2

VI. Experimental Results

We present the experimental results and analysis in this section. We first conduct experiments with single-application workloads running on a single-core system. We then run experiments on a multicore system with multi-programed workloads as modern data-centers tend to collocate multiple applications on the same machines [30].

A. Single-Core System Performance and Energy Efficiency

First, we demonstrate the benefits of heterogeneous memory systems in a single-core computing system. Using the same single-core processor, we compare the memory access time and memory energy-delay-product (EDP) under different memory systems ranging from homogeneous memory systems (based on DDR3, RLDRAM, HBM, and LPDDR2, separately) to heterogeneous memory systems with application-level allocation (*Heter-App*) and object-level allocation (*MOCA*). We calculate memory access time by adding up the queue latency, bus latency and the time required for the memory request to get serviced. We compute memory EDP by multiplying memory power and memory access latency. The memory performance and memory EDP results are shown in Fig. 8 and Fig. 9, respectively. Both plots are normalized to the results of the homogeneous memory system based on DDR3 memory module (*Homogen-DDR3*[3]).

We can observe from these figures that, on average, *MOCA* reduces the memory access time by 51% and the memory EDP by 43% over *Homogen-DDR3*. Overall, *Homogen-RL* unsurprisingly has the lowest memory access time whilst the worst energy efficiency. On the other hand, *Homogen-LP* has the worst performance among all memory systems, but due to its low power cost, it still has better EDP than *Homogen-RL* and Homogen-DDR3. Since the memory footprint of the applications under consideration is always higher than the individual memory module capacity in our heterogeneous memory system, all the objects in an application are never allocated to the best-fit memory module. Hence, the performance of *Heter-App* is generally lower than the corresponding homogeneous memory counterpart. *MOCA* achieves the best energy efficiency among all experimented memory systems and stays closest to *Homogen-RL*'s performance.

When comparing with *Heter-App*, *MOCA* provides more benefits in performance and energy efficiency for latency-sensitive applications, such as *disparity*. As shown in Fig. 2, *disparity* has two major memory objects, one with a high L2MPKI and the other with a relatively low L2MPKI. *Heter-App* first allocates the lower-L2MPKI object in RLDRAM module since it is the first one identified during runtime. Since RLDRAM module capacity is used up by this object, the higher-L2MPKI object is allocated in HBM module. On the other hand, *MOCA* is aware of both objects' characteristics, and thus, allocates the higher-L2MPKI object in RLDRAM and the lower-L2MPKI one in HBM, which improves the memory performance and reduces the memory EDP. Similarly, all the objects in *gcc* are allocated in LPDDR module in *Heter-App*, while *MOCA* allocates the higher-L2MPKI object into RLDRAM module and thereby improves performance. There is a slight drop in performance for *milc* and *mser* using *MOCA*. This is because apart from two or three memory-intensive objects, all other objects are non-memory-intensive. *MOCA* places the low-L2MPKI objects in LPDDR modules while *Heter-App* places all of them in RLDRAM or HBM modules.

In general *MOCA* outperforms *Heter-App* by 14% in memory access time and 15% in memory EDP, demonstrating that an object-level page allocation is able to unearth more of

[3] Similarly, we name the homogeneous memory systems composed of RLDRAM, HBM, and LPDDR memory modules as *Homogen-RL*, *Homogen-HBM*, and *Homogen-LP*, respectively.

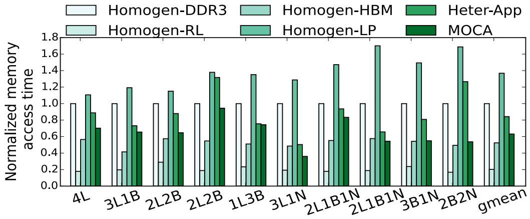


Fig. 10: Memory performance of homogeneous and heterogeneous memory systems for multi-program workloads

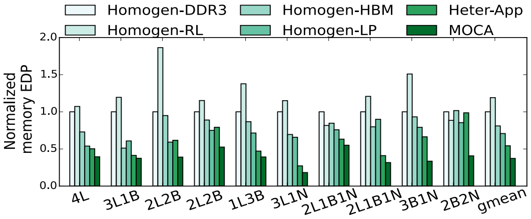


Fig. 11: Memory EDP of homogeneous and heterogeneous memory systems for multi-program workloads

heterogeneous memory systems' potential than an application-level page allocation.

B. Multicore System Performance and Energy Efficiency

We next evaluate *MOCA*'s benefits on the target multicore system with four cores. For different workload sets from Table III, we measure the total memory access time and calculate the memory power consumption for every memory system. Figure 10 and Figure 11 show the normalized memory access time and memory EDP, respectively, for all tested memory systems.

Similar to single-application results, multi-program workloads also exhibit lower EDP values with heterogeneous memory systems. Even though *MOCA* achieves higher memory access time compared to *Homogen-RL* and *Homogen-HBM*, its energy efficiency improvement is 63% over *Homogen-DDR3* and 40% over *Homogen-LP*, which makes *MOCA* the most energy-efficient one among all tested memory systems. In addition, *MOCA* reduces the memory access time by 26% and the memory EDP by 33% over *Heter-App*.

Workload sets comprising of latency-sensitive and bandwidth-sensitive applications contend more extensively for RLDRAM and HBM than the rest of workload sets. *MOCA* prioritizes the high-L2MPKI objects to RLDRAM and the high-MLP objects to HBM, thereby reducing overall memory access time. In addition, *MOCA* also places the non-memory-intensive objects to LPDDR modules, thereby reducing the memory power consumption significantly. Thus, we see energy efficiency improvement from *MOCA* over *Heter-App* – which tries to place all objects in RLDRAM module. On the contrary, the last five workload also consists of non-memory-intensive applications. The higher-L2MPKI objects from these applications are allocated to the RLDRAM modules, so the performance improvement of *MOCA* is as high as 59% (2B2N) than *Heter-App*. Although the power consumption increases, the reduced memory access time results in lower EDP.

We also see the impact of *MOCA* on the system performance and energy efficiency in Fig. 12 and Fig. 13, respectively. On average, *MOCA* is close to *Homogen-HBM* or *Homogen-RL* in performance and achieves better energy efficiency compared to all the other memory systems. *MOCA* improves system energy efficiency by up to 15% compared to *Homogen-DDR3*. Compared to *Heter-App*, we get 10% improvement in both performance and energy efficiency using *MOCA*.

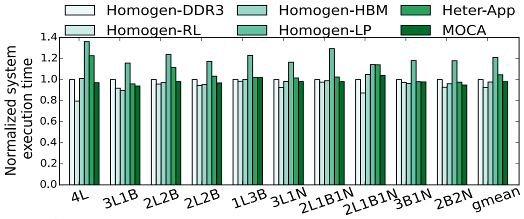


Fig. 12: System performance with homogeneous and heterogeneous memory systems for multi-program workloads

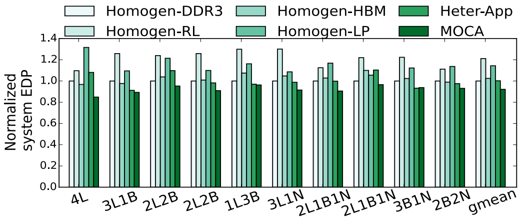


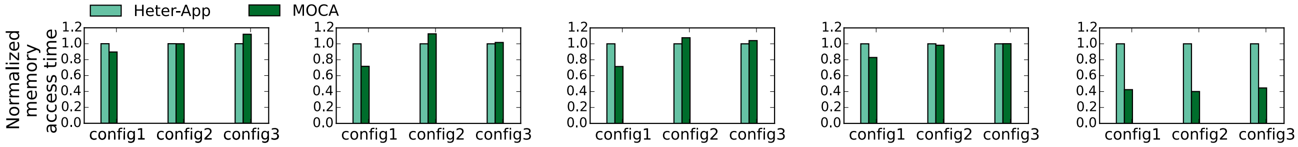
Fig. 13: System EDP with homogeneous and heterogeneous memory systems for multi-program workloads

C. Different Heterogeneous Memory System Configurations

We also conduct an investigation on heterogeneous memory system configuration's impact on performance and energy efficiency and evaluate *MOCA*'s scalability under different heterogeneous memory systems. We choose the following three configurations:

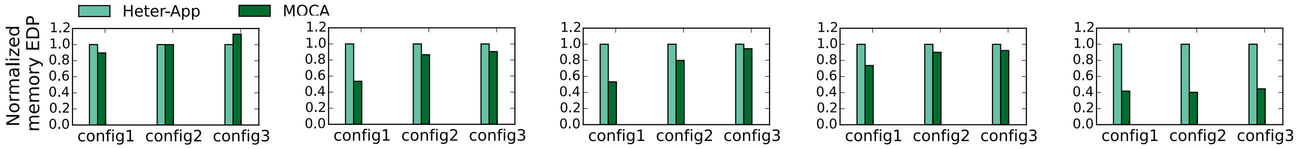
1) 256MB RLDRAM, 768MB HBM and 1GB LPDDR2
2) 512MB RLDRAM, 512MB HBM and 1GB LPDDR2
3) 768MB RLDRAM, 768MB HBM and 512MB LPDDR2

The memory performance and energy efficiency results are shown in Fig. 14 and Fig. 15, respectively, for five workload sets across the three memory system configurations. All results are normalized to *Heter-App* results. For memory-intensive workload sets (e.g., 3L1B, 1L3B and 3L1N), *MOCA* achieves lower memory access time with *config1* than *Heter-App*. This is because *config1* has a lower RLDRAM capacity, and these workload sets contend highly for RLDRAM module in *Heter-App*. Hence, some of the high-L2MPKI objects are not allocated to RLDRAM due to such contention. However, *MOCA* prioritizes these high-L2MPKI objects to RLDRAM and thus improves the performance under *config1*. As we increase the RLDRAM capacity in *config2* and *config3*, the memory performance of both *Heter-App* and *MOCA* improves. However, *Heter-App* has better performance than *MOCA* because it allocates all objects to the RLDRAM module. For all workload sets except for 2B2N, the performance of *Heter-App* is either on par or better than *MOCA*. This workload mainly consists of non-memory-intensive applications, so the memory performance does not vary much across different configurations.



(a) 3L1B (b) 1L3B (c) 3L1N (d) 2L1B1N (e) 2B2N

Fig. 14: Normalized memory access time for application- and object-level allocation with different memory system configurations



(a) 3L1B (b) 1L3B (c) 3L1N (d) 2L1B1N (e) 2B2N

Fig. 15: Normalized EDP for application- and object-level allocation with different memory system configurations

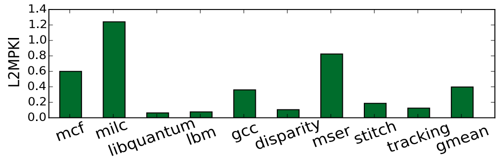


Fig. 16: L2 MPKI of stack and code segment for all applications

However, the memory power consumption of *config2* and *config3* increases significantly due to higher RLDRAM capacity. Since *MOCA* prioritizes only high-L2MPKI objects to RLDRAM and HBM, and low-L2MPKI ones to LPDDR as opposed to *Heter-App*, which allocates all objects from a high-L2MPKI set to RLDRAM, *MOCA* provides better energy efficiency. Since *config1* provides the best memory system energy efficiency among all three configurations, we select this one for all our experiments.

D. Classifying Stack Data and Code Segment

In this work, we mainly consider memory objects allocated in the heap space. In addition, there are also memory accesses from the code segment as well as the stack space. However, the memory access intensity of these segments is considerably lower than that of the heap objects. Figure 16 shows the L2MPKI for stack and code segments of the target applications. These segments exhibit lower L2MPKI values due to the higher locality of code segment and lower data size of the stack segment. Therefore, we allocate pages from LPDDR module for these segments in *MOCA*.

VII. Related Work

Heterogeneous memory systems have been studied in various contexts in prior work. We first review related work that discusses the use of on-chip scratchpad memory, 3D-stacked DRAM (Hybrid Memory Cube or HBM) or other memory technologies like PCM to construct heterogeneous memory systems. We also look at methodologies that exploit heterogeneity in memory accesses and allocation policies to obtain high performance and energy efficiency.

A. Heterogeneous Memory Systems

Several prior works construct a heterogeneous memory system comprising of either on-chip scratchpad memory [31], [32] or 3D stacked memory [33]–[36] or non-volatile memory such as PCM [37], [38] with a traditional DRAM. Many of these works employ optimal page-level allocation policies to utilize the lowest latency memory module in the system. To do so, they either track frequently accessed pages [33], [35], [36], [38] or control the amount of memory mapped to such a module based on bandwidth utilization [34]. Heterogeneous memory systems have also been proposed for reduced data processing time in cloud computing [9]. Intel's Knights Landing Processor (KNL) [5] incorporates an HBM in addition to DDR4 memory. In KNL, the programmer explicitly allocates workloads' critical data in HBM using functions built on top of existing API (e.g., libnuma) or compiler annotation. Our work uses an offline profiler to study the behavior of memory objects on training input sets, and the OS automatically chooses the best-fitting memory module for the objects at runtime.

B. Exploiting Heterogeneity in memory accesses

Phadke *et al.* [3] employ latency, bandwidth, and power optimized memory modules and choose a single optimal memory module for an application based on offline profiling. Our work shows that we can uncover substantial performance and energy savings by placing memory objects within an application in suitable memory modules. Chatterjee *et al.* [4] place critical words in a cache line in latency-optimized memory module and rest of the cache line in power-optimized modules. In contrast, our proposal uses application profiles to deduce latency/bandwidth sensitivity of memory objects and places them accordingly in memory modules. Agarwal *et al.* [39] propose a page placement policy, which places highly accessed pages in bandwidth-optimized memory in a heterogeneous memory system. This is conducive to GPU programs that hide memory latency well. In contrast, we design a framework that addresses both latency and bandwidth sensitivity of memory objects and places them accordingly.

C. Profiling Policies to guide data placement

Shen *et al.* [31] use PIN-based profiling [40] to track array allocations for placing frequently accessed and low-locality arrays in the on-chip scratchpad. They also decompose larger arrays into smaller chunks for fine-grained data placement. Dulloor *et al.* [37] profile memory access patterns of data structures as either sequential, random, or involving pointer chasing. Data structures exhibiting latency-sensitive patterns (e.g. pointer chasing) are placed in DRAM and the rest are placed in PCM. Peon-Quiros *et al.* [32] track the access frequency and changing memory footprint over time of dynamically allocated data structures to place them in either on-chip SRAM or off-chip DRAM modules. While future memory systems will employ such varied forms of memory technologies as envisioned in these prior work, our work instead, aims to highlight the heterogeneity benefits in main memory and can be employed in tandem with these works.

VIII. Conclusions

Heterogeneous memory system is a promising solution for an efficient memory system; it albeit needs intelligent data placement. Prior solutions employ either application-agnostic or application-level data placement. This paper points out that memory objects within an application exhibit substantial heterogeneity in their memory access behavior. We exploit this observation to design an intelligent data placement method for a heterogeneous memory system. We present MOCA, a framework for heterogeneous memory systems, which profiles an application and places each object in a memory module that best suits that object's memory access behavior.

Acknowledgement

We thank Schuyler Eldridge and Nathaniel Michener for their help on the initial Gem5 framework setup. This work has been partially supported by NSF grants CCF-1716352, CAREER-1149773 and CCF-1527301.

References

[1] W. A. Wulf and S. A. McKee, "Hitting the memory wall: implications of the obvious," in *Proc. Intl. Symp. on Computer Architecture (ISCA)*, 1996, pp. 90–101.

[2] K. Lim *et al.*, "Disaggregated memory for expansion and sharing in blade servers," in *Proc. ISCA*, 2009, pp. 1–12.

[3] S. Phadke and S. Narayanasamy, "MLP aware heterogeneous memory system," in *Proc. Design, Automation and Test in Europe (DATE)*, 2011, pp. 1–6.

[4] N. Chatterjee *et al.*, "Leveraging heterogeneity in dram main memories to accelerate critical word access," in *Proc. IEEE/ACM Intl. Symp. on Microarchitecture (MICRO)*, 2012, pp. 13–24.

[5] A. Sodani, "Knights landing (knl): 2nd generation intel® xeon phi processor," in *Proc. IEEE Hot Chips Symposium (HCS)*, 2015, pp. 1–24.

[6] S. Kannan, A. Gavrilovska, V. Gupta, and K. Schwan, "Heteroos: Os design for heterogeneous memory management in datacenter," in *Proc. ISCA*, 2017, pp. 521–534.

[7] S. P. Olarig, D. J. Koenen, and C. S. Heng, "Method and apparatus for supporting heterogeneous memory in computer systems," Mar. 4 2003, US Patent 6,530,007.

[8] O. Avissar, R. Barua, and D. Stewart, "Heterogeneous memory management for embedded systems," in *Proc. Intl. Conf. on Compilers, Architecture, and Synthesis for Embedded Systems*, 2001, pp. 34–43.

[9] K. Gai, M. Qiu, and H. Zhao, "Cost-aware multimedia data allocation for heterogeneous memory using genetic algorithm in cloud computing," *IEEE Trans. on Cloud Computing*, pp. 1–1, 2016.

[10] M. Awasthi *et al.*, "Handling the problems and opportunities posed by multiple on-chip memory controllers," in *Proc. Intl. Conf. Parallel Architectures and Compilation Techniques*, 2010, pp. 319–330.

[11] J. Macri, "Amd's next generation gpu and high bandwidth memory architecture: Fury," in *Proc. HCS*, 2015, pp. 1–26.

[12] J. L. Henning, "SPEC CPU2006 benchmark descriptions," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 4, pp. 1–17, 2006.

[13] S. K. Venkata *et al.*, "Sd-vbs: The san diego vision benchmark suite," in *Proc. IEEE Intl. Symp. on Workload Characterization*, 2009, pp. 55–64.

[14] C. Toal *et al.*, "An RLDRAM II implementation of a 10Gbps shared packet buffer for network processing," in *Proc. IEEE Conf. on Adaptive Hardware and Systems*, 2007, pp. 613–618.

[15] J. Standard, "High bandwidth memory (HBM) DRAM," *JESD235*, 2013.

[16] O. Mutlu, H. Kim, and Y. N. Patt, "Efficient runahead execution: Power-efficient memory latency tolerance," *IEEE Micro*, vol. 26, no. 1, pp. 10–20, Jan 2006.

[17] F. Pereira, T. Mitchell, and M. Botvinick, "Machine learning classifiers and fmri: a tutorial overview," *Neuroimage*, vol. 45, no. 1, pp. S199–S209, 2009.

[18] P. Dollár, C. Wojek, B. Schiele, and P. Perona, "Pedestrian detection: A benchmark," in *Proc. IEEE Conf. Computer Vision and Pattern Recognition (CVPR)*, 2009, pp. 304–311.

[19] M. M. Tikir and J. K. Hollingsworth, "Hardware monitors for dynamic page migration," *Journal of Parallel and Distributed Computing*, vol. 68, no. 9, pp. 1186–1200, 2008.

[20] P. Conway *et al.*, "Blade computing with the amd opteron processor ("magny-cours")," in *Proc. HCS*, 2009, pp. 1–19.

[21] N. Binkert *et al.*, "The Gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, July 2011.

[22] G. Hamerly, E. Perelman, J. Lau, and B. Calder, "Simpoint 3.0: Faster and more flexible program phase analysis," *Journal of Instruction Level Parallelism*, vol. 7, no. 4, pp. 1–28, 2005.

[23] S. Li *et al.*, "McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Proc. MICRO*, 2009, pp. 469–480.

[24] R. Kumar *et al.*, "Single-isa heterogeneous multi-core architectures: The potential for processor power reduction," in *Proc. MICRO*, 2003, pp. 81–92.

[25] "DDR3 SDRAM power calculator." [Online]. Available: https://www.micron.com/products/dram/ddr3-sdram

[26] "LPDDR2 SDRAM power calculator." [Online]. Available: http://www.micron.com/products/dram/lpdram

[27] "RLDRAM3 power calculator." [Online]. Available: http://www.micron.com/products/dram/rldram-memory

[28] B. Li *et al.*, "Exploring new features of high-bandwidth memory for gpus," *IEICE Electronics Express*, vol. 13, no. 14, pp. 20 160 527–20 160 527, 2016.

[29] V. Bychkovsky *et al.*, "Learning photographic global tonal adjustment with a database of input/output image pairs," in *Proc. CVPR*, 2011, pp. 97–104.

[30] T. Eilam *et al.*, "Managing the configuration complexity of distributed applications in internet data centers," *IEEE Communications Magazine*, vol. 44, no. 3, pp. 166–177, 2006.

[31] D. Shen, X. Liu, and F. X. Lin, "Characterizing emerging heterogeneous memory," in *Proc. Intl. Symp. on Memory Management*, 2016, pp. 13–23.

[32] M. Peón-quirós *et al.*, "Placement of linked dynamic data structures over heterogeneous memories in embedded systems," *ACM Trans. Embed. Comput. Syst.*, vol. 14, no. 2, pp. 37:1–37:30, Feb. 2015.

[33] M. R. Meswani *et al.*, "Heterogeneous memory architectures: A hw/sw approach for mixing die-stacked and off-package memories," in *Proc. Intl. Symp. on High Performance Computer Architecture*, 2015, pp. 126–136.

[34] L. Tran *et al.*, "Heterogeneous memory management for 3D-DRAM and external DRAM with QoS," in *Proc. Asia and South Pacific Design Automation Conference*, 2013, pp. 663–668.

[35] X. Dong *et al.*, "Simple but effective heterogeneous main memory with on-chip memory controller support," in *Proc. High Performance Computing, Networking, Storage and Analysis*, 2010, pp. 1–11.

[36] M. Lee, V. Gupta, and K. Schwan, "Software-controlled transparent management of heterogeneous memory resources in virtualized systems," in *Proc. Memory Systems Performance and Correctness*, 2013, pp. 1–6.

[37] S. R. Dulloor *et al.*, "Data tiering in heterogeneous memory systems," in *Proc. European Conf. on Computer Systems*, 2016, pp. 1–16.

[38] M. Pavlovic, N. Puzovic, and A. Ramirez, "Data placement in hpc architectures with heterogeneous off-chip memory," in *Proc. IEEE Intl. Conf. on Computer Design*, 2013, pp. 193–200.

[39] A. Agarwal *et al.*, "Page placement strategies for gpus within heterogeneous memory systems," in *Proc. Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2015, pp. 607–618.

[40] C.-K. Luk *et al.*, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proc. ACM Conf. on Programming Language Design and Implementation*, 2005, pp. 190–200.