Received June 22, 2020, accepted July 5, 2020, date of publication July 15, 2020, date of current version July 27, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3009432

Optimizing Heap Memory Object Placement in the Hybrid Memory System With Energy Constraints

TAEUK KIM[1], SAFDAR JAMIL[2], JOONGEON PARK[2], AND YOUNGJAE KIM[2]
[1]TmaxSoft, Seongnam 13607, South Korea
[2]Department of Computer Science and Engineering, Sogang University, Seoul 04107, South Korea

Corresponding author: Youngjae Kim (youkim@sogang.ac.kr)

This work was supported by the Next-Generation Information Computing Development Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT, under Grant 2017M3C4A7080243.

ABSTRACT Main memory significantly impacts the power and energy utilization of the overall server system. Non-Volatile Memory (NVM) devices, are suitable candidates for the main memory to reduce static energy consumption. But unlike DRAM, the access latencies and the dynamic energy consumption of write operation of the NVM devices are higher. Thus, Hybrid Main Memory Systems (HMMS) employing DRAM and NVM have been proposed to reduce the overall energy depletion of main memory while optimizing the performance of application. However, memory object placement is crucial for optimal performance and energy efficiency in HMMS due to high write latency and energy consumption of NVM devices. This paper proposes *eMap*, an optimal heap memory object placement planner for HMMS. *eMap* takes into account the object-level access patterns and energy consumption to provide an ideal placement policy for objects to mitigate performance and energy consumption. In particular, *eMap* is equipped with two modules, *eMPlan* and *eMDyn*. *eMPlan* is a static placement planner which provides one-time placement policies for memory objects to meet the energy budget. *eMDyn* is a runtime model to consider the requests of changes in the energy constraint during the application execution. Both modules are based in Integer Linear Programming(ILP) and consider three major constraints, namely decision, capacity and energy constraints to optimally placing the memory objects in HMMS. We evaluate the proposed solution with two scientific application benchmarks, NAS Parallel Benchmark (NPB) and Problem-based Benchmark Suit (PBBS), on two testbeds by emulating the NVM using QUARTZ [32]. Our extensive experiments in comparison with Memory Object Classification and Allocation (MOCA) framework showed that our solution is 4.17× less costly in terms of the memory object profiling and reduce the energy consumption up to 14% with the same performance. On the other hand, *eMDyn* module also meets the performance and energy requirement during the application execution by considering the migration cost in terms of time and energy.

INDEX TERMS Hybrid main memory system (HMMS), memory object placement, energy consumption.

I. INTRODUCTION

In the computing system, there are two major components to account for most of the energy dissipation, CPU and main memory. Recent statistics state that CPU consumes 30%-60% of the system power [10]. Several techniques to reduce that energy consumption are designed and adopted [4], [6], [7], [28]. Dynamic Voltage and Frequency Scaling (DVFS) [6] and Dynamic Power Management (DPM) [7] are the two state-of-the-art approaches to compensate for the power and energy consumption of CPU. DPM blocks the power to the processor when it is in idle state while DVFS dynamically adjusts the clock cycles and voltages of the CPU.

On the other hand, 20%-48% of the energy consumption is attributed to the main memory [3], [10], [19]. Traditional main memory systems are composed of homogeneous memory modules, mainly DRAM which is a volatile, high bandwidth and low latency memory device. But it consumes significant energy due to volatility, destructive read operations, and refresh energy. CPU-based energy reduction

The associate editor coordinating the review of this manuscript and approving it for publication was Laurence T. Yang.

TABLE 1. Specification of NVM devices and normalized energy of memory command/byte in nano-Joules [1], [11], [18], [24], [30], [36].

Memory Device	BW (GB/s)	Latency (ns)	Endurance	Row Read Energy (nJ)	Row Write Energy (nJ)	Refresh Energy (nJ)
DRAM	25.6	10-50	10^{16}	3.15	3.23	0.94
STT-RAM	10.6	32-72	10^{15}	2.86	7.68	0
PCM	3.5	50-100	10^{8}-10^{9}	1.39	34.55	0

methodologies are also studied for DRAM as well, like powering down the memory ranks, controlling the base memory voltage and frequency [9], [15]. However, these techniques do not fulfill the performance and energy requirements per application. Whereas, using DPM and DVFS at memory level degrades the overall system performance due to state transition latency. Specifically, these approaches only enable system-level power control, which cannot meet the performance requirements of various applications. Some applications need more computation while others frequently access memory to perform read and write operations.

New materials to design memory devices, such as Spin-Transfer Torque RAM (STT-RAM), Phase Change Memory (PCM), Magnetic RAM (MRAM), and 3D-XPoint, are being studied to either use as main memory or in conjunction with traditional memory, DRAM. On the other hand, these devices do not have idle energy consumption, which makes them suitable for reducing energy consumption. These Non-Volatile Memory (NVM) devices, such as STT-RAM and 3D-XPoint, make it a more suitable alternative than DRAM as the main memory due to specific properties such as byte-addressability, persistence, high density, and less energy consumption [14], [18], [27]. However, NVM offers lower bandwidth and longer latency than DRAM. Therefore, it cannot serve as a complete replacement of DRAM. Thus, Hybrid Main Memory System (HMMS) has been proposed that incorporates both DRAM and NVM on the processor memory bus [12], [17], [22], [26], [34].

The energy consumption at the application level depends on the nature of the application workloads and the access characteristics of its memory variables. Application energy consumption varies with different workloads as the application's memory object access patterns, such as lifetime, size, accessed volume, read/write ratio, spatial & temporal locality, and sequentiality, change with the workload [16]. Further, various memory devices such as DRAM, PCM, and STT-RAM exhibit different characteristics for performance and energy consumption, as shown in Table 1. In HMMS, optimally placing memory variables to a specific memory module will lead to optimized performance and high energy efficiency [12]. For example, a write-intensive variable will consume more energy at the memory module, which has high write energy, so it will be efficient to place that variable to a memory module that consumes less write energy. Thus, placing memory objects on NVM devices by considering their characteristics is likewise essential.

Several works to place memory objects in HMMS have been proposed [12], [22], [34]. These works classify the memory objects in an application into several categories, such as bandwidth, latency, streaming objects, and pointer tracking objects, and assign the application to the most appropriate memory module [12], [34]. Memory Object Classification and Allocation framework (MOCA) [22] optimizes the performance of the ternary HMMS by placing memory objects in the best-suited memory module. It considers their access behavior specifically based on the rate of the Last-Level Cache misses per kilo instruction (LLC MPKI) and reduce the energy consumption through object placement. The major goal of MOCA is to improve the performance of HMMS by selectively placing memory objects meanwhile, through placement, it reduces the energy consumption as well. However, only considering the LLC MPKI does not provide an optimal placement of memory objects in HMMS to optimize the performance and energy efficiency. As other access behaviors of the memory objects, such as lifetime, size, and accessed-bytes, play an important role in the performance and energy consumption of the application. MOCA only provides a static placement and does not consider the energy consumption requirement during the application execution time.

In this paper, we propose *eMap*, which is an optimal memory object placement algorithm based on object level profiling information and ILP-based placement algorithm. *eMap* considers the fine-grained memory objects access patterns and per-object energy consumption of an application to provide optimal placement policies for memory objects to meet the energy limiting constraint while optimizing performance in HMMS. *eMap* is equipped with two placement modules, *eMPlan* and *eMDyn*. The *eMPlan* is a static module that determines static placements of objects before applications begin to run. It optimizes the application performance while reducing the energy consumption to a specific rate by optimally placing memory objects in HMMS. The *eMDyn* is a dynamic module that reduces the energy consumption while optimizing an application's performance by re-evaluating the object placement and migrating those objects if necessary to satisfy the energy requirement during application runtime.

This paper provides following specific contributions:

- *eMPlan* employs the memory object profiler, Integer Linear Programming (ILP) based Energy Estimator, Placement Planner, and a Runtime Memory Allocator. In *eMPlan*, the memory profiler analyzes the diverse access patterns of memory objects of applications using a Two-Pass memory profiler [16]. The Energy Estimator considers the energy consumption and characteristics of both devices, DRAM and NVM, in HMMS respectively.

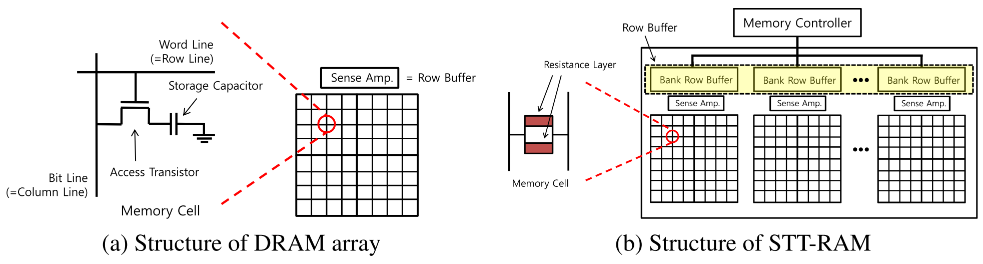


FIGURE 1. **Comparison of DRAM and STT-RAM cell architectures.**

The Placement Planner calculates the optimal placement of memory objects by considering the object access patterns and the energy consumption obtained from the Energy Estimator. The Runtime Memory Allocator allocates memory objects to respective memory modules according to the placement policies obtained by Placement Planner.

- *eMDyn* consists of an ILP-based Migration Planner and Migration Executor. While *eMPlan* decides the placement of objects to optimize the performance and meet the given energy limiting constraints, the runtime memory allocator of *eMDyn* can re-allocate the objects following the decided placement during the application execution. *eMDyn* changes the optimal placement decision at runtime of the application as the energy constraint or the request to reduce the more energy consumption can be placed by the user or system. *eMDyn* considers the incoming energy change request and anticipates the memory objects migration by considering their access patterns and obtain a new placement. *eMDyn* only migrates those memory objects which are already allocated while newly allocated objects are now placed by considering new optimal placements.
- We evaluate the proposed *eMap* using real-time application benchmarks, such as NAS Parallel Benchmark (NPB) [2] and Problem Based Benchmark Suite (PBBS) [29]. We evaluate the proposed *eMap* on two different testbed configurations. Testbed I is an IBM server, whereas, Testbed II is an Intel based Non-Uniform Memory Access (NUMA) servers. Due to the lack of actual device, we emulated STT-RAM over DRAM using the emulation platform, QUARTZ [32]. We compared our solution with MOCA framework [22]. The evaluation results show that our *eMPlan* outperforms MOCA in reducing the energy consumption up to 14% with the same performance on both of the application benchmarks. Our proposed *eMDyn* also meets the performance and energy requirements for NAS benchmark with negligible migration cost in terms of migration time and energy consumption. The average energy efficiency of *eMDyn* is up to 4% with considering migration cost.

II. BACKGROUND

This section provides the background on the object placement in HMMS, our candidate NVM device, and the object profiling.

A. SPIN-TRANSFER TORQUE RAM (STT-RAM)

From various NVM devices, STT-RAM is one of the rapidly developing memory technology. The characteristics of STT-RAM as shown in Table 1 categorize it as one of the most suitable candidates for this work as it has low latency and high durability. However, one disadvantage of STT-RAM is that it has a high write energy consumption. A recent study states that STT-RAM has more than twice of energy consumption in writing to a memory array than DRAM [13]. To deal with this, we adopted the partial write methodology as one of the energy optimization methods of STT-RAM [18].

Figure 1 shows the comparison of the memory cell structure of DRAM and STT-RAM. As shown in Figure 1(a), DRAM memory cell stores data in a storage capacitor. When a memory row is read, charge sharing occurs between the precharged bit line and the storage capacitor. This destroys the data stored in the cell. Due to this destructive read, DRAM must perform a restore operation, which requires the sense amplifier to re-write the sensed data to the memory cell. Therefore, sense amplifier should maintain the data in itself, which acts as a row buffer in DRAM. However, since STT-RAM performs non-destructive reads, its row buffer and sense amplifier exist separately and act independently from each other as shown in Figure 1(b). Thus, when STT-RAM array write occurs, updates are first made to the row buffer. If memory access whose address is not fetched to the row buffer, a row buffer conflict occurs. The row buffer write-back is operated and effective memory array write is done.

In addition, this mechanism incurs unnecessary energy consumption. The [18] states that when an STT-RAM row buffer conflict occurs, the data in row buffer is clean more than 60% and it is less than 6% that the number of dirty cache blocks in a row buffer is more than four. That is, a large portion of row buffer is unmodified at row buffer conflict, and if it was modified, the number of modified blocks in row buffer is generally less than 4 cache blocks. But, without any optimization, the whole row buffer should be written back though most of the blocks are clean and it incurs severe energy consumption in STT-RAM. To mitigate this problem, [18] proposed an optimization method, partial-write, which writes back only the dirty blocks when a row buffer conflict occurs by holding dirty bits of all cache blocks of row buffer in the memory controller. When the row buffer is 4 KB, only 64 bits of space is required. Therefore, it is spatially feasible and the energy consumption of the STT-RAM can be reduced by upto 70%.

In this work, we target STT-RAM specifically as the energy model of STT-RAM is already presented, while for other NVM devices, there is no any energy model. Besides, the adoption of our work on PCM is part of the future work as it requires considering some architectural choices as for STT-RAM, we have to consider the Row Buffer. There is no such architectural design presented for PCM yet.

B. OBJECT PLACEMENT IN HMMS

Memory devices such as High Bandwidth Memory (HBM), Reduced Latency DRAM (RLDRAM), and Low Power DDR (LPDDR) are being produced and studied in a considerable pace [24]. On the other hand, PCM and STT-RAM are the two most rapidly growing NVM devices to be placed on a processor-memory bus in conjunction with DRAM to enable HMMS [12], [20], [22], [23], [34], [35], [37]. Various works [12], [22], [34] have already been studied to place memory objects on different memory modules in HMMS by considering their characteristics.

Nevertheless, one type of memory cannot satisfy various demands at the same time as these memory devices have different read/write access latency, density, and energy utilization. For example, RLDRAM has low latency whereas, power and energy consumption are five times higher than the DRAM. 3D-XPoint has 750 times higher density than DRAM, but the latency is 1,000 times higher than DRAM [22]. If the major workload in the system requires both low latency and high density, then the main memory configuration with either RLDRAM or 3D-XPoint will not produce optimal results. However, if the main memory is configured by using those two types of memory together, it can achieve optimal results by placing latency-sensitive objects in RLDRAM and large-sized objects in 3D-XPoint. Therefore, several studies have shown interest in enabling performance efficient options to allocate memory objects in HMMS [12], [22], [33], [34]. Our target memory system is HMMS environment comprised of DRAM and NVM, where DRAM has high performance and NVM has high density and power-efficiency.

In addition, usually the different applications exhibit varying characteristics according to their object-level access patterns. In HMMS, placing memory objects according to their object-level information can be helpful in optimizing the performance and reducing the energy efficiency [12]. For example, scientific applications majorly work on their dynamically allocated memory objects and different objects exhibit different properties [16]. Now if an application allocate two objects, i.e., A and B. If object A is read-intensive which means that application will mostly read that object while object B is write-intensive. Now placing both objects in homogeneous memory system will lead to high energy consumption due to object B. On the contrary, in HMMS, placing these objects will be non-trivial as if the read-intensive object will be placed in high latency memory module than it will degrade the performance and if the write-intensive object will be placed in energy sensitive device than it will consume high amount of energy. So, in HMMS, optimally placing these objects being aware of access patterns and device properties will lead to optimal performance and high energy efficiency.

C. OBJECT-LEVEL MEMORY PROFILING

As the memory device type has an effect on energy consumption, the memory object access patterns also play a vital role in energy dissipation. For example, a write-intensive object will consume more energy in a memory device with high write energy. Therefore, the placement of objects on the basis of the object access pattern and NVM device characteristics will lead to optimized performance and energy efficiency. In this paper, we adopted the two-pass memory profiler to extract object access patterns information [16]. We utilize that extracted information to estimate the energy consumption for objects to be placed on HMMS, and the device specific energy model is explained in the Section IV.

Two-pass memory profiler targets the dynamically allocated heap memory variables and extracts basic information such as size, lifetime, and call-stack. We have extended the two-pass profiler to extract the fine-grained object access patterns and the details are provided in section IV-A1. The distinction between variables and objects is based on a call-stack consisting of the order of memory allocation function calls and the order of allocation function return addresses. The two-pass profiler workflow is shown in Figure 2.

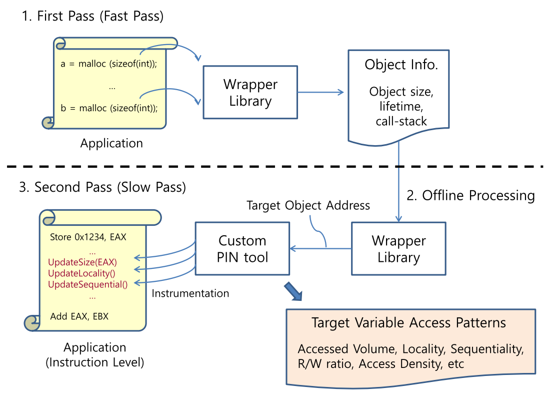

1. First Pass (Fast Pass)
a = malloc (sizeof(int));
...
b = malloc (sizeof(int));
Application
Wrapper Library
Object Info.
Object size, lifetime, call-stack
2. Offline Processing
3. Second Pass (Slow Pass)
Target Object Address
Store 0x1234, EAX
...
UpdateSize(EAX)
UpdateLocality()
UpdateSequential()
...
Add EAX, EBX
Application (Instruction Level)
Custom PIN tool
Wrapper Library
Instrumentation
Target Variable Access Patterns
Accessed Volume, Locality, Sequentiality, R/W ratio, Access Density, etc

FIGURE 2. Two-pass memory profiler [16].

III. HEAP MEMORY OBJECT PLACEMENT SYSTEM

This section describes the design goals, various components, and the interactions between components in the *eMap* system.

A. GOALS

In this section, we discuss our key design principles. ***Optimal Object Placement:*** The high access latency of NVM devices makes them ill-suited to replace the main memory. Using NVMs in conjunction with DRAM forms a HMMS. It helps in reducing the high access latency of NVM through intelligently placing memory objects in DRAM and NVM.This

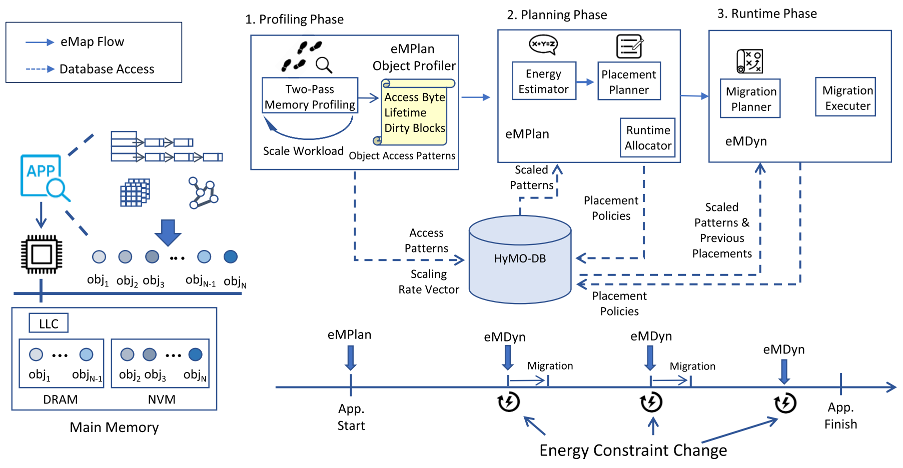


FIGURE 3. Description of various components of *eMap* and how they interact.

first goal of *eMap* is to obtain the optimal placement for heap memory objects by considering their detailed access patterns, such as lifetime, size, accessed volume, and dirty cache-lines, for an HMMS.

Energy Efficiency: The idle energy consumption of DRAM makes it a power-hungry device. On the other hand, NVMs do not have idle energy utilization, which makes them a suitable candidate for reducing the energy efficiency of the system. But, the dynamic energy consumption of the NVM devices is high, specifically when writing. So, placing memory objects in HMMS effectively will help in reducing the energy efficiency of the system. The Second goal of *eMap* is to optimize the energy efficiency of the HMMS by optimally placing the memory objects.

To achieve these goals, we proposed *eMap*, the methodology to place the memory objects in the HMMS by considering their detailed access patterns. In particular, we developed an Integer-Linear Programming based memory object placement planner to efficiently allocate the memory objects in the HMMS while meeting the energy requirements of the system.

B. OVERVIEW

Figure 3 depicts the interaction between various components of *eMap* for HMMS, composed of DRAM and STT-RAM. The left side of the diagram shows the execution of an application in HMMS where it allocates some of the memory objects in DRAM while some in STT-RAM. The right side of the figure shows three phases for *eMap*, Profiling, Planning, and Runtime. The profiling and planning phases are the part of our static module of *eMap*, while the runtime phase belongs to the dynamic module.

Profiling Phase: It adopted Two-Pass memory profiler for the extraction of memory-level object access patterns such as size, lifetime, accessed volume, and last-level cache (LLC) miss counts [16]. These extracted object access patterns are stored in the database named Hybrid Memory Object Database (HyMO-DB), as shown in Figure 3. HyMO-DB also stores the device-level characteristics and the placement decisions of the memory objects from planning and runtime phases.

Planning Phase: It is an ILP-based algorithm and employs three major constraints, i.e., (i) Decision, (ii) Capacity, and (iii) Energy. The pseudo-code of planning phase is shown in Algorithm 1. The object access patterns for an application are fetched from HyMO-DB and the placement decisions are generated.

- In the first step (lines 1 to 4), the ILP model is loaded using a third party library [5] and the Decision constraint is defined for each memory object from the HyMO-DB of an application. Decision constraint is bound to be binary (either 0 or 1) as our target HMMS consists of 2 memory devices, DRAM and STT-RAM.
- In the second step (lines 5 to 9), the Capacity constraint is defined for all the memory objects. Capacity constraint is bound not to exceed the memory modules' capacity for the placement of memory objects, which means that the number of objects that are placed on each memory module should not exceed the memory device's capacity individually.
- In the third step (lines 10 to 13), the Energy constraint is defined to reduce energy efficiency. As one of the primary goals of our proposed algorithm is to reduce the energy efficiency by optimally placing the memory objects in HMMS by considering the energy requirement, we take the rate of energy consumption to be reduced from DRAM energy consumption as input and bound the ILP constraint to not exceed for each memory object.
- In the fourth step (lines 14 to 16), we define the objective function of our proposed algorithm, which is to optimize the performance, determine the overall latency of each memory object, and bind the objective function according to the latency values.

Algorithm 1 ILP-Based Object Placement Algorithm

```
   Input: Object access patterns and energy rate
   Output: Placement decisions
1  Load ILP model;
   // Decision Constraint
2  while objects do
3      Add Constraint;
4      Bound to be binary;
   // Capacity Constraint
5  while Objects do
6      set objects.size → ILP Format;
7  Add Constraint;
8  Bound constraint ≤ DRAM capacity;
9  Bound constraint ≤ NVM Capacity;
   // Energy Constraint
10 while Objects do
11     set object.energy → ILP Format;
12 Add Constraint;
13 Bound constraint ≤ energy.DRAM * Energy Rate;
   // Objective Function
14 while Objects do
15     set object.latency → ILP Format;
16 Load ILP_Objective_Function;
   // Compute Model
17 set_minim(ILP_model) ;        // Optimize for
   minimization
18 write_ILP(ILP_model) ;    // Write the model
   with all the constraints
19 sovel_ILP(ILP_model) ;    // Solve the model
20 return Object_placement ;    // Return object
   placement to HyMODB
```

- Last but not the least step (lines 17 to 19), we optimize our ILP algorithm for minimum values so that the performance is optimized. Then we write the ILP model with all the above three constraints and compute the model for the optimal placement decisions. Once the placement is calculated, it is stored in the HyMO-DB, and the application is executed with static placements.

Runtime Phase: eMDyn plays a vital role during the execution of the application, as there may be a need to change the energy limiting constraint during the application execution. In the Runtime phase, the migration planner is triggered which re-evaluates the placement of memory objects by considering their current states, such as where an object is placed and how much lifetime of the object is remaining, and obtains a new placement policy for all the major objects. The migration planner is also based on the ILP algorithm shown in Algorithm 1 with just modifications in the computation part of the energy consumption. Once the new placement is obtained, the object is either migrated based on the placement decision from its previously placed memory module to the new memory module that is from DRAM to NVM and vice versa by *eMDyn* migration executor module.

IV. DESIGN AND IMPLEMENTATION

In this section, we explain the details of our proposed *eMap* approach.

A. eMPlan: STATIC OBJECT PLACEMENT

This section provides design details of the *eMPlan*.

1) OBJECT PROFILER

eMPlan profiles the memory object access patterns and estimates the energy consumption of the object with device specific energy model and extracted object access patterns. We extended the Two-Pass memory profiler [16] to extract fine-grained profiling information of heap memory objects. As shown in Figure 2, Two-Pass Profiler operates in two passes, i.e., Fast-pass and Slow-pass. Fast-pass identifies all the heap memory allocations using the call-stack and assign a hashed identifier and the size of the objects are also obtained. In the offline processing (when application is not being executed), target memory objects are selected for detailed profiling. For effectiveness and to reduce the complexity of profiling, we only take into account those memory objects whose accessed size is larger than 1 MB, called major objects.[1]

The Slow-pass then considers the target objects selected in the offline processing and extracts the detailed access patterns. The Slow-pass utilized customized PIN-Tool [21] which can easily be extended to extract all the necessary object-level access patterns at instruction-level. Two-Pass profiler provides a wrapper library for tracing the heap memory allocation calls, such as malloc, realloc, and calloc, and each heap memory object access goes through the custom analysis code which is based on PIN-tool. We extended the wrapper library to extract the required object access information using the PIN-Tool. We traced all store instructions to the heap-allocated objects and calculated the various object access patterns. Also, we used the Performance API (PAPI) [31], a hardware event counter, in the custom analysis code to count the cache misses.

For STT-RAM row buffer, we set a buffer to have the same size in the virtual memory of profiler. This virtual buffer is used to count the number of dirty cache blocks which will be written back to the memory array when a row buffer conflict occurs. The assumption here is that when a single process runs on the machine, the number of dirty cache blocks in the virtual memory buffer and the actual device's row buffer will exhibit a consistent pattern if the sizes of both buffers are identical.

For DRAM page policy, we take into account the closed page policy, which always flushes open row buffer to corresponding row in the memory array. While open page policy

[1]Our solution only considers the major objects for the placement and we interchangeably used the terms major object, heap memory objects and simply the objects.

has different types, for example fixed open page policy and adaptive open page policy, we did not consider open page policy concepts because we just deal with the object placement in HMMS, not optimization of DRAM memory controller. Thus, we assumed general memory controller page policy.

The remaining memory objects that are less than 1 MB are placed in DRAM. And the baseline placement of memory objects and the code of the application is DRAM. To estimate the energy consumed by the objects in HMMS, the following memory access information is required: (i) object size, (ii) total amount of memory read and written by object, accessed volume (iii) object lifetime, and (iv) the total number of dirty cache blocks in a certain size of row buffer.

2) SCALING RATE VECTOR

When an application's workload changes its access patterns are also varied accordingly. However, [16] states that 98.1% of the objects are scaled or fixed as the input workload size scales. This means that when input workload scales, object access patterns also scale consistently (with a scaling rate of 1 for a fixed object). Thus, the profiling of target application is not required for every time the workload changes and a scaling rate vector can be derived. The scaling rate vector of the access patterns is based on the profiling information of various workloads of the application so it can be stored in the HyMO-DB shown in Figure 3.

The input size of the application can be set by user which makes the derivation of scaling rate vector easy. For example, if the workloads of the application are 'N', we can derive the average rates of access patterns among the 'N' input sizes and compose the vector with these rates. A generalized view of the scaling rate vector for various access patterns can be shown in Equation 1 where ap_i is a particular access pattern, such as size, lifetime and LLC miss count, in_i is the workload size of the target application, and N is the total number of workload a target application provides.

$$avg\_grad = \sum_{i=1}^{N-1} \{(ap_{i+1} - ap_i)/(in_{i+1} - in_i)\}/(N-1) \quad (1)$$

For example, if the i-th object has size (S_i) is 10 MB for the workload in_1, $S_i = 19\ MB$ for the workload in_2 and $S_i = 25\ MB$ for the workload in_3 then the scaling rate vector for the size of i-th object can be derived as:

$$\{(19\ MB - 10\ MB)/(in_2 - in_1) + (25\ MB - 19\ MB)/(in_3 - in_2)\}/2$$

3) ENERGY ESTIMATOR

The energy estimation in *eMPlan* is the key component as it provide the estimated energy consumption to compute the optimal placement of memory objects. The energy estimator calculates the per-object energy consumption for both of the memory modules of HMMS where all the objects are placed to either of the memory devices, respectively. We adopted STT-RAM as an example of NVM device and suggest the energy model of DRAM and STT-RAM based on the methodology [18]. In this work, we target STT-RAM specifically as the energy model and the architectural details are provided in [18] while other NVM devices architectural details and the energy models are not yet determined.

TABLE 2. Energy consumption of memory command per byte [18].

Memory Command	Energy (nJ)
DRAM Activate+Pre-charge	3.07
DRAM Read/Write	1.19
DRAM Refresh	0.35
STT-RAM Activate+Pre-charge	2.68
STT-RAM Row Buffer Access	1.00
STT-RAM Write-Back	2.83

The memory commands are classified as Activate (ACT), Pre-charege (PRE), Read/Write (RD/WR), Refresh (REF), Row Buffer Access (RBA), and Write-Back (WB) [18]. ACT is the command which activates the accessed bank and row before memory RD/WR in both memory devices. PRE pre-charges the bit-line to prepare the next memory access and to restore the read or written data in memory array of DRAM. RD/WR are the actual memory read and write. REF recharges the voltage to storage capacitor of memory cell to prevent a data loss due to current leakage in DRAM. RBA is the cost to access the row buffer and WB is the cost of writing row buffer data back to memory array when a row buffer conflict occurs in STT-RAM. Table 2 shows the per-byte energy consumed by above mentioned commands.

Our proposed energy model calculates the energy consumption on per-object basis. Equation 2 represents the energy consumption of the i-th object when it is placed in DRAM. The accessed volume (AV_i) of the object represents how much in total an object is being accessed during its lifetime which is extracted during the profiling phase. It is reasonable to multiply (AV_i) with DRAM ACT, PRE and REF energy consumption. For DRAM refresh energy (dE_{REF}), we assumed selective refresh policy per row (4 KB). In addition, the refresh energy of DRAM is also being considered with the lifetime (T_i) and actual size (S_i) of the object. The reason to consider the accessed volume and size separately is to comprehensively take into account all the read and write operations that are being performed for object during its lifetime. Equation 3 represents the energy consumption of the i-th object when it is placed in STT-RAM. As shown in the section II-A, STT-RAM read/write operations fall back to Row Buffer that's why we have considered the row buffers exclusively while considering the read/write operations to STT-RAM. Same as DRAM, STT-RAM also bears the cost of ACT and PRE. In addition, we have considered the write backs to STT-RAM in terms of number of dirty cache blocks (N_{DC}) and the cache block size (V_{CB}). Table 3 defines the notations used in the equations.

$$DE_i = dE_{A+P} \cdot AV_i + dE_{RW} \cdot AV_i + dE_{REF} \cdot S_i \cdot T_i \quad (2)$$

$$NE_i = nE_{A+P} \cdot AV_i + nE_{RBA} \cdot AV_i + nE_{WB} \cdot N_{DC} \cdot V_{CB} \quad (3)$$

TABLE 3. Notations used in the equations where *i* represents the *i*th object.

Notation	Description
dE_{A+P}	DRAM activate+pre-charge
dE_{RW}	DRAM read/write
dE_{REF}	DRAM refresh
nE_{A+P}	STT-RAM activate+pre-charge
nE_{RBA}	STT-RAM row buffer access
nE_{WB}	STT-RAM write-back
DE_i	DRAM energy consumption
NE_i	NVM energy consumption
CP_i	Previous placement policy
dnE_i	Migration energy from DRAM to NVM
ndE_i	Migration energy from NVM to DRAM
$MigCE1_i$	Migration energy cost from DRAM to NVM
$MigCE2_i$	Migration energy cost from NVM to DRAM
T_i	Lifetime
sT_i	Allocation time
fT_i	De-allocation time
$MigCT_i$	Total migration time cost
dnL_i	Total latency from DRAM to NVM migration
ndL_i	Total latency from NVM to DRAM migration
$MigTD_i$	Migration time from DRAM to NVM
$MigTN_i$	Migrate time from NVM to DRAM

4) PLACEMENT PLANNER

The Placement Planner of the *eMPlan* determines the optimal placement of memory objects to optimize the performance while satisfying the energy limiting constraint that is requested externally. It utilizes the per-object energy estimation model for DRAM and STT-RAM. We modeled the Integer-Linear Programming (ILP) algorithm for the Placement Planner. Our model is based on three major constraints for the implementation of Placement Planner, Decision Constraint, Capacity Constraint, and Energy Limiting Constraint. We adopt a third-party shared library, lp_solve [5], to implement these constraints.

a: DECISION CONSTRAINT

The Decision Constraint is to make the placement decision for each memory object that whether a particular object will be placed on DRAM or NVM. This placement can be represented by an ILP variable, X_i, which represents 0 for NVM and 1 for DRAM as shown in equation 4.

$$0 \le X_i \le 1 \quad \text{for } i = 1, 2, \ldots, N \tag{4}$$

b: CAPACITY CONSTRAINT

The second constraint takes into account the limited capacities of memory devices. It checks that all the allocated objects sizes should not exceed the capacity of memory device. Equation 5 shows the capacity constraint for both memory devices. C_D represents DRAM while C_N is NVM capacity.

$$\sum_{i=1}^{N} X_i \cdot S_i \le C_D \quad \sum_{i=1}^{N} (1 - X_i) \cdot S_i \le C_N \tag{5}$$

c: ENERGY CONSTRAINT

The third constraint considers the energy limitation requests issued by client or the remaining battery lifetime of the system. The external energy limit constraint is given as a specific ratio of existing energy consumption. All objects of target application must be allocated not to exceed the required ratio of the energy which is consumed when all objects are placed in DRAM. Equation 6 shows the energy limiting constraint. Let the required ratio be R, then the sum of energy consumption of all the objects placed in HMMS should not exceed R times the energy consumption of objects placed entirely in DRAM (DE_i).

$$\sum_{i=1}^{N} \{X_i \cdot DE_i + (1 - X_i) \cdot NE_i\} \le \sum_{i=1}^{N} DE_i \cdot R \tag{6}$$

d: OBJECTIVE FUNCTION

The goal of *eMPlan* is to minimize memory access latency while satisfying the above constraints. That is, the sum of whole HMMS access time should be minimized. Each device access time can be derived by multiplying the total access counts of objects and the latency of the device. The Performance API [31] is used to count the actual memory access in profiling step to get the total LLC miss counts. This objective is presented in Equation 7 where L_{DRAM} and L_{NVM} indicate the latency of DRAM and NVM respectively, and $L3M_i$ indicates the LLC miss count of i-th object.

$$f = \sum_{i=1}^{N} \{X_i \cdot L_{DRAM} \cdot L3M_i + (1 - X_i) \cdot L_{NVM} \cdot L3M_i\} \tag{7}$$

5) RUNTIME ALLOCATOR

Once the Placement Planner decides the placements for all the major variables, the target application is executed in real-time and the Runtime Allocator of *eMPlan* operates to allocate those objects. The Runtime Allocator configures the object allocation table with the determined placement at the initialization step. In the object allocation table, the identification of objects is achieved with the hash values of the call-stack of dynamic allocation functions. Once the target application starts execution, *eMPlan* hooks all the dynamic memory allocation functions on every object and calculates the hash value from its call-stack and compares with the object allocation table to identify the target objects. If the allocated object is the placement target object, the placement decision of the object is referred from the object allocation table. If the mapped device is DRAM, existing allocation functions such as malloc is used. If the allocated device is NVM then NVM allocation API, which is provided by NVM emulation tool QUARTZ [32], is used.

B. eMDyn: DYNAMIC OBJECT PLACEMENT

eMDyn is the second module of *eMap* and it considers the energy limiting requests at the runtime and re-evaluate the placement of memory objects and migrate then to meet the new energy constraint. *eMDyn* is based on two sub-modules, migration planner and migration executor.

1) MIGRATION PLANNER

The migration planner is an ILP-based algorithm to re-calculate the placement of major objects to meet the new energy requirements. Shuffling the memory objects to meet the energy constraint also incurs some energy consumption of migration, i.e., migration cost. So, it considers the access patterns, migration costs in terms of energy and performance, and the new energy limiting constraint to satisfy the energy while optimizing the performance of application in HMMS. It is also based on similar three major constraints, Migration Decision, Capacity, and Energy Constraint.

a: DECISION CONSTRAINT

The migration decision (X_i) shows that if it is beneficial to migrate an object from its current placement to new one. It is similar to Equation 4. If it is beneficial to migrate than X_i will be 1 otherwise 0.

b: CAPACITY CONSTRAINT

Similar to section IV-A4.b, it considers that the migrated objects size should not exceed the capacity if memory devices. Let CP_i be the previous placement of the object before energy constraint change. Due to space limitation, we have omitted the equations of Migration Decision and Capacity Constraint as they are equivalent to *eMPlan*.

c: ENERGY CONSTRAINT

The major goal of *eMDyn* is to meet the new energy limiting constraint while optimizing the performance. For that migration planner calculates the total energy consumption including the migration cost and then decide the new placement. The energy consumed by the objects that are being migrating from DRAM to NVM and NVM to DRAM are shown in Equation 8 and Equation 9, respectively.

$$dnE_i = DE_i \cdot \frac{t - sT_i}{T_i} + MigCE1_i + NE_i \cdot \frac{fT_i - t}{T_i} \quad (8)$$

$$ndE_i = NE_i \cdot \frac{t - sT_i}{T_i} + MigCE2_i + DE_i \cdot \frac{fT_i - t}{T_i} \quad (9)$$

Here, t indicates the time point during the application execution when the request of energy constraint change occurred. In addition, the migration cost for energy consumption ($MigCE1_i$ & $MigCE2_i$) for DRAM to NVM and vice-versa is equivalent to Equation 2 and Equation 3, respectively. The major difference is instead of counting the total accessed volume, here we only consider the size of the object and the migration cost in terms of time deemed with DRAM REF energy. Due to space limitations, we excluded the equation representation.

Using Equations 8 and 9, the total amount of energy consumption involving object migration can be presented in Equation 10.

$$E_{total} = \sum_{i=1}^{N} \Big[X_i \cdot \{CP_i \cdot dnE_i + (1 - CP_i) \cdot ndE_i\} + (1 - X_i) \cdot \{CP_i \cdot dE_i + (1 - CP_i) \cdot nE_i\} \Big] \quad (10)$$

Equation 10 is the left-hand side of the energy limit constraint inequality. In the meantime, the right-hand side of the inequality may vary according to the purpose of the external request. The requested energy constraint can be categorized into two possibilities. First, the new energy constraint is effective only when the limit is strictly kept. That is, if the object migration cannot satisfy the new energy constraint, *eMDyn* does not shuffle the current placement of memory objects. Second, the new energy constraint does not require tight limiting. For instance, user may require to reduce energy consumption regardless of meeting the energy constraint. In this case, *eMDyn* shuffles the memory objects.

To consider these different demands, the Migration Planner provides an additional flag, F, as an input parameter whose value is 1 when the purpose belongs to case (i) and 0 otherwise. By considering these cases, the requirement Rq can be shown as Equation 11.

$$Rq = F \cdot \Big[\sum_{i=1}^{N} dE_i \cdot R_n\Big] + (1 - F) \cdot \Big[\sum_{i=1}^{N} \{CP_i \cdot dE_i + (1 - CP_i) \cdot nE_i\}\Big] \quad (11)$$

Equation 11 becomes the right-hand side of the energy limiting inequality and R_n indicates the newly required energy constraint. Therefore, the total energy constraint shown in Equation 10 should be less than and equation to the required energy shown in Equation 11.

d: OBJECTIVE FUNCTION

The Migration Planner aims to minimize the memory access latency, and it can be calculated by the sum of total latency due to objects that are either migrated or not. If an object which is assigned to DRAM currently is a migration candidate to NVM, the total latency (dnL_i) of that object can be shown as Equation 12.

$$dnL_i = L_D \cdot L3M_i \cdot \frac{t - sT_i}{T_i} + MigCT_i + L_N \cdot L3M_i \cdot \frac{fT_i - t}{T_i} \quad (12)$$

The Placement Planner of *eMPlan* profiles the number of LLC misses ($L3M_I$) of memory object in advance to count the number of actual memory accesses. However, the information of how many LLC misses would occur during the object migration cannot be measured before runtime. Thus, we assume that the access to the memory device would occur for the whole object both in reading and writing. The migration cost ($MigCT_i$) for the time taken to migrate can be calculated by considering the access latency of each device (L_D & L_N) and the size of the memory object.

Likewise, an object which was placed to NVM previous is the migration candidate then its total access latency can be presented as Equation 13.

$$ndL_i = L_N \cdot L3M_i \cdot \frac{t - sT_i}{T_i} + MigCT_i + L_D \cdot L3M_i \cdot \frac{fT_i - t}{T_i} \quad (13)$$

Thus, the total delay time of the objects for migration can be presented as shown in Equation 14.

$$f = \sum_{i=1}^{N} \Big[X_i \cdot \{CP_i \cdot dnL_i + (1 - CP_i) \cdot ndL_i\} + (1 - X_i) \cdot L3M_i \cdot \{L_D \cdot CP_i + L_N \cdot (1 - CP_i)\} \Big] \quad (14)$$

2) MIGRATION EXECUTOR

Once the migration decision for all the target objects is made, the Migration executor operates to perform the migration task and relocate the memory objects between respective memory modules. The steps of the object migration are as follow:

1) A new object with the same size of the candidate object is allocated in the respective memory module.
2) The currently stored data of the candidate object is copied to the new object.
3) The pointer of the candidate object is revised to point towards the newly allocated object.
4) The candidate object is de-allocated.

In step 3, Migration executor should maintain the address values of not only the pointer directly referring to the object in the allocation time, but also all general pointer variables which point to the object in the target application. In this work, we implement a member function that registers application pointers' addresses, and we have called it in every pointer reference on major objects in the target application. But, this method incurs application code modification to register the pointer addresses for Migration executor. To deal with this problem, the proxy pointer concept, which is similar to the proxy object suggested by [8], can be applied. By maintaining one proxy pointer per major object, Migration executor can set all application pointers to refer to this proxy pointer. Migration executor will be able to migrate the objects only with changing the destination of proxy pointer. In case of migration, two minor issues that need to be consider during the migration are the migration scenarios and the case of failure at the migration.

a: MIGRATION CASES

Some of the example of migration cases can be: (1) when the user deliberately wants energy efficiency due to high charges of supporting systems from the data-center service providers and (2) when the system is required to reduce the energy consumption for long running applications to provide resources to the other applications.

b: FAILURE-SAFE MIGRATION

The memory objects are migrated in a failure-safe manner across the memory modules. For instance, if the failure occurs at Step 2 of the migration executor, the application will still access the previous object pointer as the pointer in the application is not updated or if the failure occurs during the Step 3 of the migration executor, the application will still access the previous pointer as the new pointer is not updated completely.

V. EVALUATION

A. EXPERIMENTAL SETUP

We evaluate our proposed *eMap* system on two different testbed configurations and we have evaluated two benchmarks, Problem-based Benchmark Suit (PBBS) [29] and NAS Parallel Benchmark (NPB) [2] as shown in Table 4.

TABLE 4. Testbed specifications and benchmark workloads.

Configuration	Component	Value
Test-bed I	Processor	Intel Xeon E5-2650V4, 2 Sockets, 8 cores per socket
	L1 Cache	32KB 8-way set-associative (per core)
	LLC	20MB 8-way set-associative (shared)
	Memory	2 channel, 16GB, 16 banks, 16KB row buffer
	DRAM Latency	Read: 200 (ns), Write: 200 (ns)
	STT-RAM Latency	Read: 640 (ns) Write: 1440 (ns)
Test-bed II	Processor	Intel Xeon CPU E5-4640, 4 Sockets, 10 cores per socket
	L1 Cache	64KB 8-way set-associative (per core)
	LLC	20MB 8-way set-associative (shared)
	Memory	2 channel, 8GB, 16 banks, 16KB row buffer
	DRAM Latency	Read: 400 (ns), Write: 400 (ns)
	STT-RAM Latency	Read: 840 (ns) Write: 1640 (ns)
NVM Emulation	Emulation Tool	Quartz
Benchmark	Benchmark	NPB [2], PBBS [29]
	Applications	NPB: Conjugate Gradient (CG), Fourier Transform (FT)
		PBBS: Breadth First Search (BFS), Spanning Forest (SF)
	Memory Footprint	NPB: CG: 1.08GB, FT: 1.08GB
		PBBS: BFS: 6.78GB, SF: 4.3GB

For the emulation of NVM in HMMS, we adopted the QUARTZ emulation platform [32]. The read and write latency of DRAM is considered as 10ns [36] while the read latency of STT-RAM is 32ns and write latency is 72ns [30]. In our system configurations, the memory latency before emulation is measured to be 200 ns with QUARTZ [32]. We computed the ratio of DRAM to STT-RAM latency and shown in Table 4 for emulation. For the evaluation of the energy consumption in both testbed configurations, we calculated the estimated energy consumption with suggested equations. Equation variables (memory access patterns) are derived from object-level profiling. For the evaluation, we only present the estimated energy consumption of the memory system excluding the CPU and caches. As measuring the energy consumption of memory systems in real-time is not possible due to lack of measuring tools.

We compared *eMPlan* with MOCA [22], which improves both performance and energy by selectively placing memory objects in HMMS. MOCA measures the LLC MPKI of objects in HMMS consisting of high-bandwidth, low-latency, and low-power memory modules. In addition, MOCA also considers the memory-level parallelism in profiling which is beyond the scope of this work. It allocates memory-intensive objects which have high LLC MPKI values to high-bandwidth and low-latency memory modules.

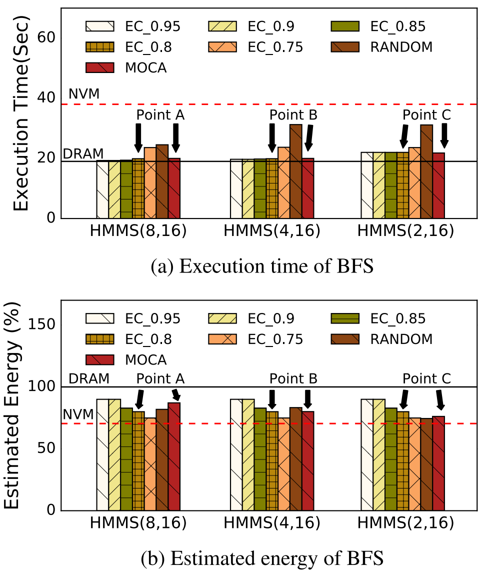
This methodology is applicable to HMMS that composed of DRAM and NVM by considering DRAM as high-bandwidth and low-latency memory.

B. eMPlan PERFORMANCE AND ENERGY EVALUATION

In this section, we will present the performance and energy estimation evaluation of our static module of *eMap*, *eMPlan*, using the PBBS and NPB on Testbed I.

1) ANALYSIS ON PBBS APPLICATIONS (BFS, SF)

In this section, we compare the results of *eMPlan* placement with multiple energy limiting constraints by counter-part placement methodology, MOCA [22].



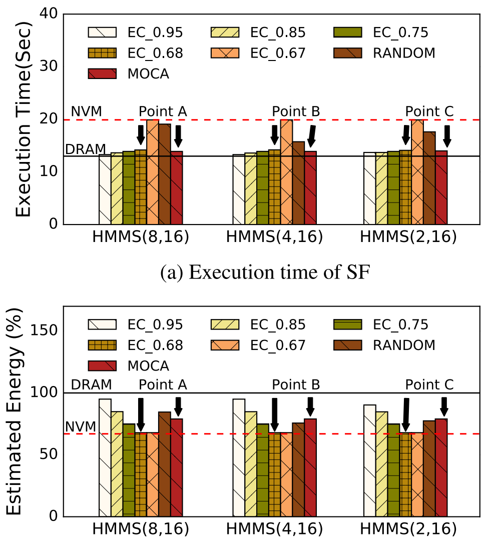
(a) Execution time of BFS

(b) Estimated energy of BFS

FIGURE 4. The performance and energy consumption of the PBBS BFS Application. The x-axis represents different HMMS configurations in terms of capacities of DRAM and NVM. HMMS(8, 16) determines that DRAM is 8 GB while NVM is 16 GB. While y-axis shows the execution time and estimated energy consumption percentage for both, respectively.

Figure 4(a) and (b) show the performance and estimated energy consumption of the proposed *eMPlan* for BFS application, respectively. Considering the larger density of NVM, we conduct various experiments while reducing the capacity of DRAM in HMMS. The memory footprint of the workload is shown in Table 4 and the selection of HMMS configuration was from extreme limited to enough capacity in terms of DRAM. In Figure 4, the EC_X shows the energy limiting constraint in contrast to DRAM-only that the allocated objects will not exceed X times of the energy consumption. For example, $EC\_0.9$ is the case where objects are allocated in HMMS to consume energy less than 90% of the DRAM-only case and so on. On the other hand, the random case shows the execution time and estimated energy consumption when objects are randomly allocated without any placement decision in a range which does not exceed the capacity of given memory devices.

The MOCA in the experimental results are the object allocation followed by the methodology [22]. In Figure 4(a),



(a) Execution time of SF

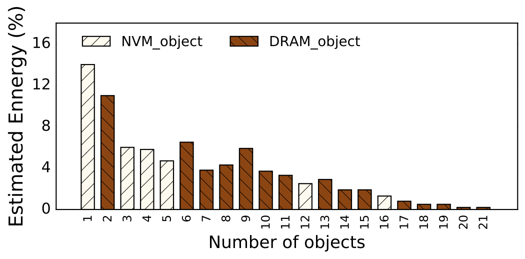
(b) Estimated energy of SF

FIGURE 5. The performance and energy consumption of the PBBS SF Application. The x-axis represents different HMMS configurations while y-axis shows the execution time and estimated energy consumption percentage for both, respectively.

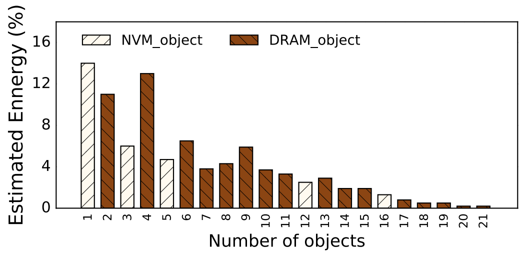
the execution time increased as the energy constraint becomes more restricted such as $EC\_0.8$ and above. The reason is *eMPlan* gives priority to performance-critical objects to be placed on DRAM and once the energy limit constraint becomes more strict than 80%, it starts to allocate the performance-critical objects to NVM. Nevertheless, if an application user wants to sacrifice some performance to reduce more energy, one may need intense constraint over 80%. Random and MOCA methodologies cannot consider the placement which decreases further energy consumption with performance trade-off. Figure 4(b) shows that *eMPlan* meets the given energy constraints. The random placement has shown the worst energy efficiency where its execution time is longer than $EC\_0.75$.

Figure 4(b) has also shown that *eMPlan* is more energy-efficient than the MOCA methodology. The A, B, and C points in Figure 4 show that the placement policies of *eMPlan* and MOCA are almost similar, however, at point A, the energy consumption of MOCA is 8.2% higher than *eMPlan*. In contrast, at point B the performance and energy consumption of both are almost identical. In addition, at point C, the energy consumption of MOCA is less than *eMPlan* as the proposed approach prefers to place performance critical objects on DRAM to optimize the performance of HMMS. On the other hand, MOCA only provides one-time placement policies for memory objects without considering the user requirements for performance and energy efficiency.

To better understand this, we analyze the per-object energy consumption of the BFS application as shown in Figure 6. The object placement decision of both techniques is almost similar except 4-th object, where *eMPlan* has placed that object in NVM while MOCA has placed it on DRAM.



(a) *eMPlan* $EC\_0.8$



(b) MOCA

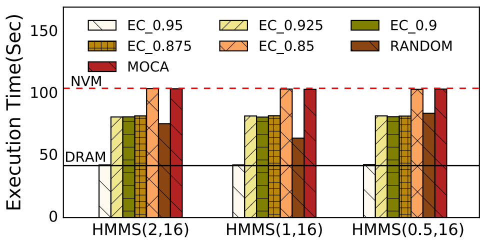
FIGURE 6. **Per-object energy consumption of PBBS BFS (normalized to all-DRAM energy consumption). The x-axis shows the number of memory objects while y-axis is the estimated energy consumption.**

4-th object has the second-longest lifetime among objects of BFS and occupies the largest memory usage (826 MB), so if it is placed to DRAM, the amount of energy consumed in refreshing is large. Also, though the LLC MPKI value of 4-th object is bigger than the threshold (0.025), it is not too large enough to impact performance. Therefore, when 4-th object is allocated to DRAM, it does not only result in significant performance improvement but also consumes over 2.2× more energy. The *eMPlan* module places 4-th object to NVM by considering this in advance, but MOCA places it in DRAM because MOCA cannot consider object access pattern and memory devices characteristics.

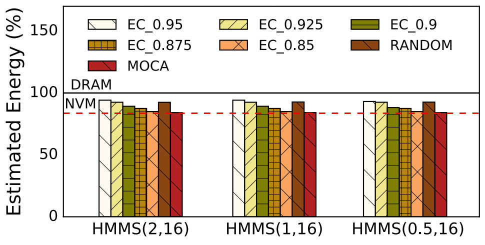
Spanning Forest (SF) of PBBS benchmark also shows consistent results with BFS. Figure 5(a) and (b) show the performance and estimated energy consumption at given energy constraint. Figure 5(a) shows that the execution of *eMPlan* is same until the EC_0.67 as the application latency increases as the energy constraint go beyond 67% of DRAM-only. This is because *eMPlan* effectively works to minimize the latency while satisfying the energy constraint until 68% of DRAM-only as it place the latency-insensitive objects to NVM in order to further reduce the energy consumption on 67% of DRAM-only energy constraint. Figure 5(a) and (b) also show that *eMPlan* is more energy efficient than MOCA. The A, B, and C points in Figure 5 show that *eMPlan* placement decisions at energy constraint EC_0.68 have the same application execution time as MOCA. However, the estimated energy consumption is 14% more efficient than MOCA as *eMPlan* places the memory objects by considering detailed access patterns and the characteristics of memory devices. Thus our methodology places only those objects to NVM which has better energy efficiency than MOCA. On the other hand, MOCA only considers the Last-Level Cache misses and memory-level parallelism to decide the placement of memory objects which leads to sub-optimal memory placement decisions and results in high energy consumption consequently.

2) ANALYSIS OF NPB BENCHMARK (CG, FT)

We also evaluate the NPB benchmark, a high-performance computing workload, to analyze the results by changing energy limit constraints. The applications used in this experiment are CG and FT. Figure 7(a) and (b) show the performance and the estimated energy consumption of CG application with varying energy constraints. Fig 7(a) shows that performance deteriorates as the energy constraint is becoming strict due to the small number of objects that actually affect the performance. In CG, five out of 14 objects occupy almost 99% of DRAM size. Thus, as energy constraint increases, major objects are placed in the NVM causing performance degradation. Figure 7(b) shows that all the placement methodologies satisfied the energy constraint. But for CG, the MOCA placement has the lowest energy consumption but the longest execution time. If a low energy limit and fast execution time are required, the current MOCA methodology cannot satisfy this requirement.



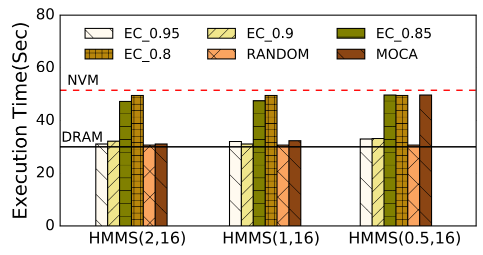
(a) Execution time of CG



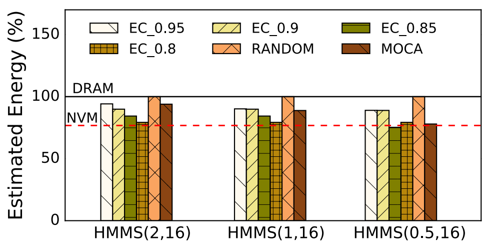
(b) Estimated energy of CG

FIGURE 7. **The performance and energy consumption of the NPB CG Application. The x-axis represents different HMMS configurations while y-axis shows the execution time and estimated energy consumption percentage for both, respectively.**

FT application of the NPB benchmark exhibits the same execution patterns as CG. Figure 8(a) shows the execution time of FT, as the energy constraint is becoming strict the application performance is degraded due to the limited number of major objects. FT has only six objects in total where four of them occupy 99.7% of total DRAM space. After certain energy constraints, objects that have major impacts on the execution time must be placed to NVM in order to



(a) Execution time of FT



(b) Estimated energy of FT

FIGURE 8. The performance and energy consumption of the NPB FT application. The x-axis represents different HMMS configurations while y-axis shows the execution time and estimated energy consumption percentage for both, respectively.

meet the required energy limit. Thus, when those objects are allocated to NVM, performance is decreased rapidly. Figure 8(b) shows that *eMPlan* meets the given energy constraint. In FT, the placement policy of *eMPlan* at energy constraint EC_0.9 has a similar execution time as of MOCA in HMMS (2, 16) configuration while it has 4.3% more energy efficiency. This is because when the DRAM capacity is sufficient, the *eMPlan* module can take advantage of energy consumption by calculating the object placement which MOCA methodology cannot account for. However, when DRAM capacity is reduced, placement cases of *eMPlan* are strictly limited, so the energy difference between *eMPlan* and MOCA placements decreases.

C. ENERGY CONSUMPTION COMPARISON MOCA VS eMPlan

In this experiment, we modified the MOCA methodology and configure it to meet the energy constraint. In original MOCA [22], objects are allocated on the basis of the specific threshold of the LLC MPKI, if the object has met the threshold then it will be placed in high performant memory otherwise placed at low-performing memory device. The LLC MPKI threshold is derived from several experiments for efficient performance while maintaining energy consumption. MOCA can also satisfy the energy limiting constraints if we set the LLC MPKI threshold effectively. However, MOCA cannot estimate the amount of energy consumed by each object in DRAM and NVM based binary HMMS because it does not consider the detailed object access patterns and NVM device characteristics. Through MOCA, the threshold to satisfy the energy constraint cannot be calculated but it must be empirically set by performing several experiments repeatedly.

MOCA samples the LLC MPKI and ROBH stall cycle information at a fixed interval (i.e. 1000 instructions) when the target application is executed and records the information along with the call-stack. After application execution is finished, MOCA maps those information to the object via allocation function call-stack, and then calculates the memory object energy consumption with per-object information. That is, if we assume that MOCA uses object access pattern profiling and energy models of DRAM and NVM in this research, MOCA can calculate the total energy consumption after the execution of the target application.

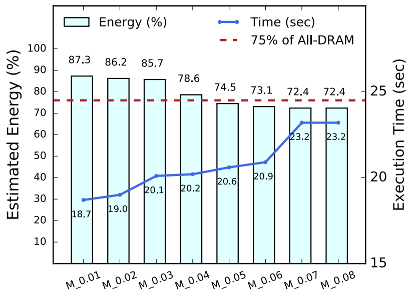


FIGURE 9. PBBS BFS execution time & energy consumption estimation of MOCA on various LLC MPKI values. (Estimated energy consumption is normalized to all-DRAM energy). The x-asis is the LLC MPKI threshold, M_*Thr*.

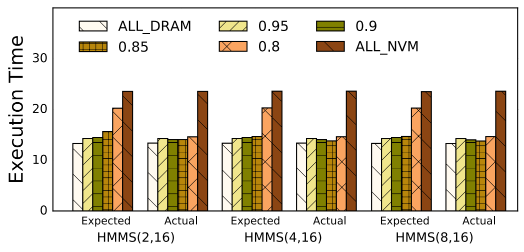
Figure 9 shows the estimated energy consumption based on various LLC MPKI values of MOCA placement in the PBBS BFS. It shows if the LLC MPKI threshold varies by same unit, then the change on energy consumption does not have any consistent pattern. Thus, to find the LLC MPKI value satisfying the energy limit constraint through MOCA, the *Thr* should be searched by a certain unit. For example, to meet the energy constraint that consumes less than 75% of energy to DRAM-only, MOCA should search by a certain unit increase in LLC MPKI. In our experimental environment, *eMPlan* module takes up to 23.635 seconds of placement computation time and object allocator overhead, which is only related to BFS application. On the other hand, MOCA should execute the application four times, which takes 78 seconds, to find the adequate LLC MPKI threshold to meet the energy constraint. With including real execution time that takes 20.6 seconds in M_0.05, MOCA placement spends 4.17× than *eMap* execution time in this example.

Also, there are cases where the fluctuation of LLC MPKI threshold value, which affects energy consumption and performance, is extremely minimal. In the case of NPB CG, for example, when the LLC MPKI threshold is 0.0024, the execution takes 68.715 seconds, and its energy consumption is equivalent to 91.9% of DRAM-only. But, when the LLC MPKI threshold is lowered to 0.0023, the execution takes 42.473 seconds and consumes 95.1% of energy of DRAM-only. If the energy consumption limit less than 92% of DRAM-only is required, the effective LLC MPKI threshold can be obtained by performing LLC MPKI value search in units of 0.0001. When the search is performed in a smaller unit, the search overhead increases accordingly and process need to be repeated every time the energy limit is changed.

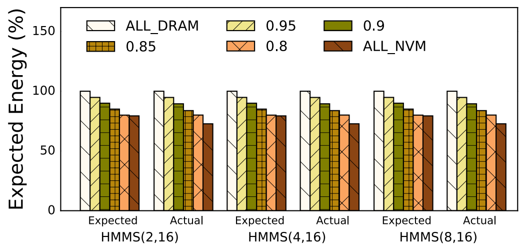
D. ACCURACY OF SCALING RATE VECTOR

In this section, we evaluated the accuracy of our proposed scaling rate vector to avoid the profiling of the application whenever the workload changes. As the workload of an application varies, the access information also changes accordingly and application workloads can be categorized into three groups; fixed, scaling, and irregular [16]. Most of the applications from scientific group lies in the scaling category as their access patterns scale with the scaling workload. For this experiment, we profiled the BFS application with various workloads and calculated the scaling rate vector for all the major variables as explained in section IV-A2. We present the accuracy of our proposed scaling rate vector in terms of the placement of memory objects in HMMS. We evaluated this experiment on Testbed II.

BFS is categorized in the scaling class that as the input workload scales the access patterns of the variables also scales but the ratio of scaling is not consistent for most of the objects. So, we adopted the generalized way to calculate the scaling rate vector and shows the effectiveness of our proposal. Figure 10 show the performance and expected energy consumption with various HMMS configurations and energy constraints (EC_X). The evaluation shows that most of the time it accurately places the memory object to their respective memory module, which ultimately omits the huge cost of profiling the application again with scaled workload.



(a) Execution time of the BFS

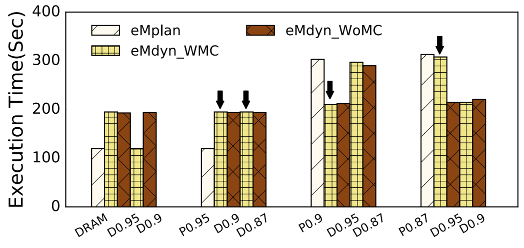


(b) Estimated energy of the BFS

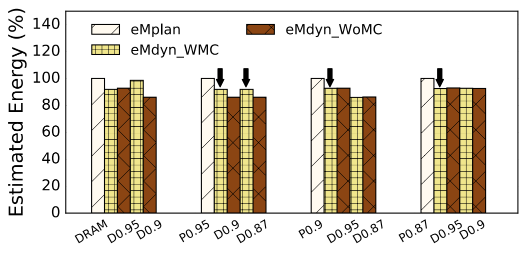
FIGURE 10. The performance and energy consumption comparison with scaled and actual object placement policies. The x-axis shows the expected (computed using proposed scaling vector), actual (through profiling), and various HMMS configurations.

E. eMPlan PERFORMANCE AND ENERGY EVALUATION

In this section, we evaluate the performance and energy efficiency of the second module of *eMap* system, *eMDyn*. Experiments elaborated in this section are performed on Testbed II. We evaluate NPB Benchmark CG and FT applications for *eMDyn* due to their simple code-base and design. We have modified both of the applications to call the member function to register the application pointer addresses as explained in section IV-B. Due to limited space, we show the evaluation results of only one configuration of HMMS, i.e., HMMS(2,16) where the DRAM capacity is 2 GB and STT-RAM capacity is 16 GB. In the following experiments, we only consider the migration case (1) where an application user deliberately requests for energy efficiency.



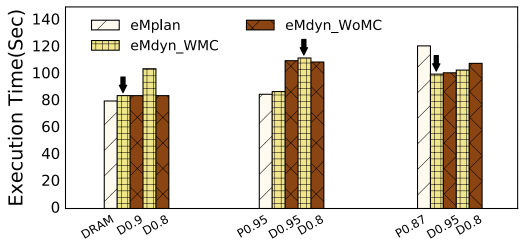
(a) Execution time of CG



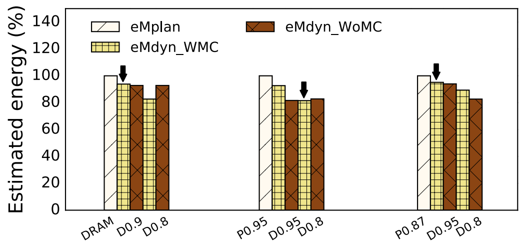
(b) Estimated energy of CG

FIGURE 11. The performance and energy consumption of the NPB CG Application. The x-axis is energy constraint where Px shows the energy constraint of *eMPlan* as baseline and Dy is the changed energy constraint through *eMDyn*.

Figure 11 shows the performance and estimated energy of the CG application under various energy limiting constraints. During the application execution, the request to change the energy limiting constraint occurs and *eMDyn* module is triggered. It re-evaluates the placement of memory objects and shuffles them accordingly. To compute the placement, *eMDyn* interrupts the execution of the application and performs its task and resumes the execution of the application from the same point where it interrupted. In Figure 11, *eMdyn_WoMC* shows the *eMDyn* without considering the migration cost while *eMdyn_WMC* is with migration cost. Figure 11(a) shows that the performance deteriorates as the energy constraint becomes more strict while the performance is improved with week energy constraint. Figure 11(b) shows that the *eMDyn* reduces the energy consumption as the energy constraint becomes more restricted while the energy consumption is increased if the requested energy limiting constraint is to get more performance. The execution time and the energy consumption of *eMdyn_WoMC* is almost similar to the *eMdyn_WMC* with energy consumption but at the points shown through arrows in the Figure 11 eMDyn_WoMC did not meet the performance and energy criteria. This inconsistency of eMDyn_WoMC is due to not considering the migration cost in terms of energy and performance.



(a) Execution time of FT



(b) Estimated energy of FT

FIGURE 12. The performance and energy consumption of the NPB FT Application. The x-axis is energy constraint where P*x* shows the energy constraint of *eMPlan* as baseline and D*y* is the changed energy constraint through *eMDyn*.

Figure 12(a) and (b) show the performance and energy efficiency of *eMDyn* for the FT application. *eMDyn* shows a similar pattern as of CG application. The performance is degraded as the requested energy constraint is more restricted while the energy consumption is reduced. eMdyn_WoMC also showed a consistent pattern in FT application as CG while the *eMDyn* satisfied the energy and performance in all the cases.

Figure 13 shows the overall execution time of CG application with *eMPlan* and *eMDyn* with various configurations. We modified the CG application and triggered the *eMDyn* on the basis of number of iterations as CG application consists of main loop for computation. We changed the energy limiting constraint during the application execution and the number inside each breakdown of the bar in Figure 13 shows the changed energy constraint. For the first two bars, we triggered the *eMDyn* after the half number of iterations and show the overhead of *eMDyn*. The other two bars show when the *eMDyn* is triggered after every 20th iteration. From Figure 13, it is shown that the overall overhead of *eMDyn* is negligible and it can be called for several times during the execution of the application. But it should be noted that this overhead can be increased according to the number and size of the objects being migrated.

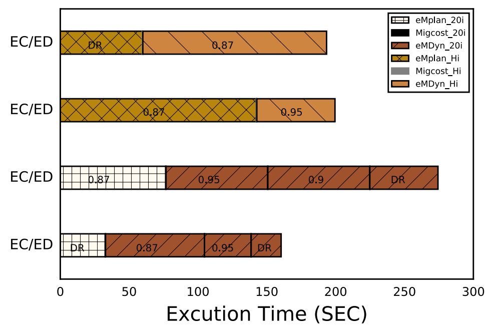


FIGURE 13. Analysis of time break down of the CG application.

VI. RELATED WORK

Various works have been done to optimize the performance and energy efficiency of HMMS through the placement of memory objects. Dullor *et al.* [12] classif y an object into streaming, random, and pointer-chasing patterns based on the dependency and sequentiality of memory access and determine the placement to optimize performance using a greedy algorithm. Wu *et al.* [34] classified memory objects into bandwidth- and latency-sensitive based on the number of memory accesses and the time taken for the object to optimize performance in MPI applications. However, these works only focus on optimizing performance in the assumption that NVM consumes less power and energy than DRAM. They do not consider that memory energy consumption is affected by the characteristics of NVM devices and object access patterns of application. Further, they also did not consider the energy consumption requirement of various settings.

In addition, the HMMS which is comprised of high-bandwidth, low-latency, and low-power memory modules is also being studied. MOCA [22] and Phadke *et al.* [25] have proposed their solutions for it, which place the object in the most suitable memory device to improve performance and energy efficiency. Phadke *et al.* [25] classified the applications in bandwidth, latency, and power-sensitive and allocates the objects of the application to a best-fit memory module. It only optimizes the performance of the HMMS and does not consider energy efficiency. MOCA [22] considers the performance and energy consumption of the ternary HMMS at a finer granularity. They profile the application to obtain the access behavior in terms of LLC MPKI and provide one-time placement of memory objects. MOCA methodology can be applied to binary HMMS consisting of DRAM and NVM. However, MOCA has limitations that it does not estimate the energy consumption by considering the characteristics of the NVM device and the detailed access pattern of the memory object. It also did not take into account the energy requirements during the runtime of the application.

Existing studies do not consider the amount of energy an object consumes due to its various access patterns and the different characteristics of NVM devices in HMMS. We can optimize the performance and energy efficiency of HMMS through detailed profiling of memory objects access patterns and the NVM device specification.

VII. CONCLUSION

HMMS is a promising solution for an energy-efficient memory system. Albeit, it requires intelligent data placement solutions. Prior solutions either placed application-level or obtained sub-optimal placement of memory objects and only provide static placement schemes. This paper proposed an optimal memory object placement solution by considering both memory access patterns and the nature of memory devices of HMMS. *eMap* calculates the expected energy consumption of objects and allocates the objects to achieve optimal performance, as well as to satisfy the energy limiting constraint. *eMap* provides static (*eMPlan*) and dynamic (*eMDyn*) placements of memory objects. *eMPlan* places the memory objects at the start of the application by considering their various access patterns and the energy limiting requirements, while *eMDyn* takes into account the changes in energy limiting constraint during the runtime of the application. Our proposed solution meets the energy requirement of 4.17 times less cost while compared to the state-of-the-art memory allocation and classification framework MOCA. It reduces energy consumption by up to 14% without compromising the performance.

ACKNOWLEDGEMENT

T. Kim is currently with TmaxSoft, South Korea, but most of the work is done when he was in Sogang University.

REFERENCES

[1] *3D-Xpoint Specification*. Accessed: Jul. 16, 2020. [Online]. Available: https://ark.intel.com/products/187936

[2] D. H. Bailey, *NAS Parallel Benchmarks*. Boston, MA, USA: Springer, 2011, pp. 1254–1259.

[3] L. A. Barroso and U. Hölzle, "The case for energy-proportional computing," *Computer*, vol. 40, no. 12, pp. 33–37, Dec. 2007.

[4] L. Benini and G. De Micheli, "Networks on chips: A new SoC paradigm," *Computer*, vol. 35, no. 1, pp. 70–78, 2002.

[5] M. Berkelaar, K. Eikland, and P. Notebaert. (2004). *Lpsolve 5.5, Open Source (Mixed-Integer) Linear Programming system*. Software. Accessed: Jul. 16, 2020. [Online]. Available: http://lpsolve.sourceforge.net/5.5/

[6] E. Calore, A. Gabbana, S. F. Schifano, and R. Tripiccione, "Evaluation of DVFS techniques on modern HPC processors and accelerators for energy-aware applications," *Concurrency Comput., Pract. Exper.*, vol. 29, no. 12, Jun. 2017, Art. no. e4143.

[7] E.-Y. Chung, L. Benini, A. Bogliolo, Y.-H. Lu, and G. De Micheli, "Dynamic power management for nonstationary service requests," *IEEE Trans. Comput.*, vol. 51, no. 11, pp. 1345–1361, Nov. 2002.

[8] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson, "NV-Heaps: Making persistent objects fast and safe with next-generation, non-volatile memories," in *Proc. SIGPLAN*, vol. 39, pp. 105–118, Mar. 2011.

[9] H. David, C. Fallin, E. Gorbatov, U. R. Hanebutte, and O. Mutlu, "Memory power management via dynamic voltage/frequency scaling," in *Proc. 8th ACM Int. Conf. Autonomic Comput. ICAC*, 2011, pp. 31–40.

[10] M. Dayarathna, Y. Wen, and R. Fan, "Data center energy consumption modeling: A survey," *IEEE Commun. Surveys Tuts.*, vol. 18, no. 1, pp. 732–794, 1st Quart., 2016.

[11] G. Dhiman, R. Ayoub, and T. Rosing, "PDRAM: A hybrid PRAM and DRAM main memory system," in *Proc. 46th Annu. Design Autom. Conf. ZZZ DAC*, Jul. 2009, pp. 664–669.

[12] S. R. Dulloor, A. Roy, Z. Zhao, N. Sundaram, N. Satish, R. Sankaran, J. Jackson, and K. Schwan, "Data tiering in heterogeneous memory systems," in *Proc. 11th Eur. Conf. Comput. Syst. EuroSys*, 2016, p. 15.

[13] F. Hameed, C. Menard, and J. Castrillon, "Efficient STT-RAM last-level-cache architecture to replace DRAM cache," in *Proc. Int. Symp. Memory Syst. MEMSYS*, 2017, pp. 141–151.

[14] Y. Huai, "Spin-transfer torque MRAM (STT-MRAM): Challenges and prospects," *AAPPS Bull.*, vol. 18, pp. 33–40, Jan. 2008.

[15] I. Hur and C. Lin, "A comprehensive approach to DRAM power management," in *Proc. IEEE 14th Int. Symp. High Perform. Comput. Archit.*, Feb. 2008, pp. 305–316.

[16] X. Ji, C. Wang, N. El-Sayed, X. Ma, Y. Kim, S. S. Vazhkudai, W. Xue, and D. Sanchez, "Understanding object-level memory access patterns across the spectrum," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, Nov. 2017, p. 25.

[17] J. Kim, Y. Kim, A. Khan, and S. Park, "Understanding the performance of storage class memory file systems in the NUMA architecture," *Cluster Comput.*, vol. 22, no. 2, pp. 347–360, Jun. 2019.

[18] E. Kultursay, M. Kandemir, A. Sivasubramaniam, and O. Mutlu, "Evaluating STT-RAM as an energy-efficient main memory alternative," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw. (ISPASS)*, Apr. 2013, pp. 256–267.

[19] C. Lefurgy, K. Rajamani, F. Rawson, W. Felter, M. Kistler, and T. W. Keller, "Energy management for commercial servers," *Computer*, vol. 36, no. 12, pp. 39–48, Dec. 2003.

[20] F. X. Lin and X. Liu, "Memif: Towards programming heterogeneous memory asynchronously," in *Proc. 21st Int. Conf. Architectural Support Program. Lang. Operating Syst.*, New York, NY, USA: Association for Computing Machinery, 2016, pp. 369–383.

[21] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proc. ACM SIGPLAN Conf. Program. Lang. design Implement. - PLDI*, 2005, pp. 190–200.

[22] A. Narayan, T. Zhang, S. Aga, S. Narayanasamy, and A. Coskun, "MOCA: Memory object classification and allocation in heterogeneous memory systems," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. (IPDPS)*, May 2018, pp. 326–335.

[23] I. B. Peng, R. Gioiosa, G. Kestor, P. Cicotti, E. Laure, and S. Markidis, "RTHMS: A tool for data placement on hybrid memory system," in *Proc. ACM SIGPLAN Int. Symp. Memory Manage. ISMM*, 2017, pp. 82–91.

[24] I. B. Peng and J. S. Vetter, "Siena: Exploring the design space of heterogeneous memory systems," in *Proc. SC Int. Conf. High Perform. Comput., Netw., Storage Anal.*, Nov. 2018, pp. 427–440.

[25] S. Phadke and S. Narayanasamy, "MLP aware heterogeneous memory system," in *Proc. Design, Autom. Test Eur.*, Mar. 2011, pp. 1–6.

[26] M. K. Qureshi, V. Srinivasan, and J. A. Rivers, "Scalable high performance main memory system using phase-change memory technology," in *Proc. 36th Annu. Int. Symp. Comput. Archit. ISCA*, 2009, pp. 24–33.

[27] L. E. Ramos, E. Gorbatov, and R. Bianchini, "Page placement in hybrid memory systems," in *Proc. Int. Conf. Supercomput. ICS*, 2011, pp. 85–95.

[28] G. Semeraro, G. Magklis, R. Balasubramonian, D. H. Albonesi, S. Dwarkadas, and M. L. Scott, "Energy-efficient processor design using multiple clock domains with dynamic voltage and frequency scaling," in *Proc. 8th Int. Symp. High Perform. Comput. Archit.*, Feb. 2002, pp. 29–40.

[29] J. Shun, G. E. Blelloch, J. T. Fineman, P. B. Gibbons, A. Kyrola, H. V. Simhadri, and K. Tangwongsan, "Brief announcement: The problem based benchmark suite," in *Proc. 24th ACM Symp. Parallelism Algorithms Archit. SPAA*, 2012, pp. 68–70.

[30] R. Takemura, T. Kawahara, K. Miura, H. Yamamoto, J. Hayakawa, N. Matsuzaki, K. Ono, M. Yamanouchi, K. Ito, H. Takahashi, S. Ikeda, H. Hasegawa, H. Matsuoka, and H. Ohno, "A 32-mb SPRAM with 2T1R memory cell, localized bi-directional write driver and '1'/'0' dual-array equalized reference scheme," *IEEE J. Solid-State Circuits*, vol. 45, no. 4, pp. 869–879, Apr. 2010.

[31] D. Terpstra, H. Jagode, H. You, and J. Dongarra, "Collecting performance data with papi-c," in *Tools for High Performance Computing*, M. S. Müller, M. M. Resch, A. Schulz, and W. E. Nagel, Eds. Berlin, Germany: Springer, 2009, pp. 157–173.

[32] H. Volos, G. Magalhaes, L. Cherkasova, and J. Li, "Quartz: A lightweight performance emulator for persistent memory software," in *Proc. 16th Annu. Middleware Conf.*, 2015, pp. 37–49.

[33] C. Wang, S. S. Vazhkudai, X. Ma, F. Meng, Y. Kim, and C. Engelmann, "NVMalloc: Exposing an aggregate SSD store as a memory partition in extreme-scale machines," in *Proc. IEEE 26th Int. Parallel Distrib. Process. Symp.*, May 2012, pp. 957–968.

[34] K. Wu, Y. Huang, and D. Li, "Unimem: Runtime data managementon non-volatile memory-based heterogeneous main memory," in *Proc. Int. Conf. for High Perform. Comput., Netw., Storage Anal.*, Nov. 2017, p. 58.

[35] K. Wu, J. Ren, and D. Li, "Runtime data management on non-volatile memory-based heterogeneous memory for task-parallel programs," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage, Anal. (SC)*, Nov. 2018, pp. 1–13, Art. no. 31.

[36] Y. Zhang and S. Swanson, "A study of application performance with non-volatile main memory," in *Proc. 31st Symp. Mass Storage Syst. Technol. (MSST)*, May 2015, pp. 1–10.

[37] B. Zhao, "Improving phase change memory (PCM) and spin-torque-transfer magnetic-ram (STT-MRAM) as next-generation memories: A circuit perspective," Tech. Rep., Jan. 2014.

TAEUK KIM received the B.S. and M.S. degrees in computer science engineering from Sogang University, Seoul, South Korea, in 2017 and 2019, respectively. He is currently a Researcher with Tmax Cloud, Seongnam, South Korea. His research interests include parallel and distributed file system and memory-level energy consumption.

SAFDAR JAMIL received the B.E. degree in computer systems engineering from the Mehran University of Engineering and Technology (MUET), Jamshoro, Pakistan, in 2017. He is currently pursuing the M.S. degree with Sogang University, Seoul, South Korea. His research interests include memory-centric computing and system energy optimization.

JOONGEON PARK received the B.S. degree from the Department of Information and Communication Engineering, Myongji University, Yongin, South Korea, in 2019. He is currently pursuing the M.S. degree with Sogang University, Seoul, South Korea. His research interests include system energy optimization and embedded systems.

YOUNGJAE KIM received the B.S. degree in computer science from Sogang University, South Korea, in 2001, the M.S. degree from KAIST, in 2003, and the Ph.D. degree in computer science and engineering from Pennsylvania State University, University Park, PA, USA, in 2009. He is currently an Associate Professor with the Department of Computer Science and Engineering, Sogang University, Seoul, South Korea. Before joining Sogang University, he was a Staff Scientist with the U.S. Department of Energy's Oak Ridge National Laboratory, from 2009 to 2015, and an Assistant Professor with Ajou University, Suwon, South Korea, from 2015 to 2016. His research interests include distributed file and storage, parallel I/O, operating systems, emerging storage technologies, and performance evaluation.