Object-Level Memory Allocation and Migration in Hybrid Memory Systems

Haikun Liu, *Member, IEEE*, Renshan Liu, Xiaofei Liao, *Member, IEEE*, Hai Jin, *Fellow, IEEE*, Bingsheng He, *Member, IEEE*, and Yu Zhang, *Member, IEEE*

Abstract—Hybrid memory systems composed of emerging *non-volatile memory* (NVM) and DRAM have drawn increasing attention in recent years. To fully exploit the advantages of both NVM and DRAM, a primary goal is to properly place application data on the hybrid memories. Previous studies have focused on page migration schemes to achieve higher performance and energy efficiency. However, those schemes all rely on online page access monitoring (costly), and data migration at the page granularity may cause additional overhead due to DRAM bandwidth contention and maintenance of cache/TLB consistency. In this article, we present *Object-level memory Allocation and Migration (OAM)* mechanisms for hybrid memory systems. OAM exploits a profiling tool to characterize objects' memory access patterns at different execution phases of applications, and applies a performance/energy model to direct the initial static memory allocation and runtime dynamic object migration between NVM and DRAM. Based on our newly-developed programming interfaces for hybrid memory systems, application source codes can be automatically transformed via static code instrumentation. We evaluate OAM on an emulated hybrid memory system, and experimental results show that OAM can significantly reduce system energy-delay-product by 61 percent on average compared to a page-interleaving data placement scheme. It can also significantly reduce data migration overhead by 83 and 69 percent compared to the state-of-the-art page migration scheme CLOCK-DWF and 2PP, respectively, while improving application performance by up to 22 and 10 percent.

Index Terms—Non-volatile memory (NVM), hybrid memory systems, object migration, memory allocation, data placement

1 Introduction

Emerging byte-addressable *non-volatile memory* (NVM) technologies promise high density, near-zero static energy consumption, and low cost per byte compared to DRAM. Despite these advantages of NVM, it is not a direct replacement of DRAM currently because of the relatively higher write latency, write energy consumption, and limited write endurance [1], [2], [3]. A practical way of using NVM is to architect it in conjunction with commodity DRAM. However, one primary challenge is how to design smart data placement strategies to fully exploit the advantages of two memory technologies and to overcome their drawbacks.

Operating systems (OSes) often have no priori knowledge to guide the initial data placement in hybrid memory systems. A naive policy may place hot data on slow memory, and thus result in poor application performance [4]. Many studies have proposed page migration techniques to improve memory access performance and energy efficiency [5], [6], [7], [8], [9], [10]. Those schemes all rely on the recency and frequency of page accesses to decide data placement on DRAM or NVM.

- *Haikun Liu, Renshan Liu, Xiaofei Liao, Hai Jin, and Yu Zhang are with the National Engineering Research Center for Big Data Technology and System, Service Computing Technology and System Lab, Cluster and Grid Computing Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan 430074, China. E-mail: {hkliu, rsliu, xfliao, hjin, yuzh}@hust.edu.cn.*
- *Bingsheng He is with the School of Computing, National University of Singapore, Singapore 117418. E-mail: hebs@comp.nus.edu.sg.*

Manuscript received 14 May 2019; revised 30 Nov. 2019; accepted 26 Jan. 2020. Date of publication 12 Feb. 2020; date of current version 7 Aug. 2020. (Corresponding author: Xiaofei Liao.) Recommended for acceptance by B. Childers. Digital Object Identifier no. 10.1109/TC.2020.2973134

Unfortunately, today's commodity x86 hardware does not support page access counting. Prior hardware-assist page migration schemes require significant modifications of the current hardware architectures [5], [6], [11]. Some other work resorts to OSes for memory access monitoring [12], [13]. A typical page table entry (PTE) maintained by OS contains an "accessed" bit. OS can monitor this "accessed" bit and knows that a page is accessed. However, this single bit is insufficient to identify whether this page is hot (frequently accessed) or cold. Thus, software-based memory access monitoring approaches usually invalidate *Translation Lookaside Buffer* (TLB) to track each memory reference [12], [13]. This page access counting mechanism often incurs significant performance overhead. Moreover, page migration usually needs to take a period of time to detect the hot pages, and thus diminishes the benefit of page migration. Also, the predicted page access pattern may not match the future access behavior, resulting in unnecessary page migrations.

On the other hand, OSes usually are hard to exploit application-level semantics for fine-grained data migration. A page may contain a large number of small application objects, and each may have distinct access pattern. Also, a hot page may contain only a small fraction of frequently-accessed objects, while the remaining objects are cold (see more results in Section 2). As a result, data migration at the page granularity is not efficient to adapt to different access patterns of individual objects. Furthermore, huge pages have been increasingly used for big data applications and virtualization platforms [13], [14]. The coarse-grained page migration may even degrade system performance due to inefficient use of DRAM capacity and bandwidth.

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

To make a better data layout in hybrid memory systems, a recent work employs offline profiling tools to characterize memory access pattern at the granularity of application objects, and then guides their static placements on either DRAM or NVM [15]. This scheme only considers a holistic view of memory access behaviors, and ignores the potential variation of memory access patterns, i.e., the object access frequency (hotness) may change dynamically in different execution phases. To this end, another work combines static object placement with dynamic migration of selected pages [16] to handle these fluctuations. However, it still relies on hardware extensions in the memory controller to perform online page access monitoring and migration.

To reduce the performance overhead of online page monitoring and migration, we propose *Object-level memory Allocation and Migration (OAM)* mechanisms for hybrid memory systems. Similar to the work [15], [16], OAM exploits an offline profiling approach to capture object-level memory access patterns, and employs a performance/energy model to guide object memory allocation and migration. Unlike the studies [15], [16] that only use the profiling traces to generate a holistic view of object access behaviors, we go further to analyze objects' access statistics in a more fine-grained manner. More specifically, we analyze the memory traces in fine-grained time slots for each object, and adopt our performance/energy model to identify different execution phases in which objects' memory access pattern changes. We find out the location in applications' source codes where an execution phase would change in the future, and automatically inject object migration instructions via static code instrumentation. When the modified program is running, it performs object migration itself by taking account of the DRAM usage and the net benefit of data migration.

In summary, we make the following contributions:

1) We provide an offline profiling tool to characterize applications' memory access pattern in details, and propose a performance/energy model to direct the initial memory allocation and dynamic migration of application objects, without any hardware modification and OS intervention for online memory monitoring.
2) We extend the *Glibc* library and Linux kernel to provide new application programming interfaces (APIs) for hybrid memory management. Programmers can explicitly allocate DRAM or NVM to application objects, and migrate them between DRAM and NVM.
3) We develop a static code instrumentation tool to automatically transform object-level memory allocation and migration in application source codes, without burdening application programmers.
4) We evaluate OAM in a DRAM-based hybrid memory emulator with a wide range of workloads. Our experimental results demonstrate the feasibility of object-level data placement/migration to narrow the performance gap between NVM and DRAM. The performance difference between OAM and DRAM-only is only 2 percent on average. OAM is able to significantly reduce data migration overhead and achieve better performance than the state-of-the-art page migration schemes. For instance, compared to CLOCK-DWF [5] and 2PP [16], OAM reduces migration overhead by 83 and 69 percent on average, while improving application performance by up to 22 and 10 percent.

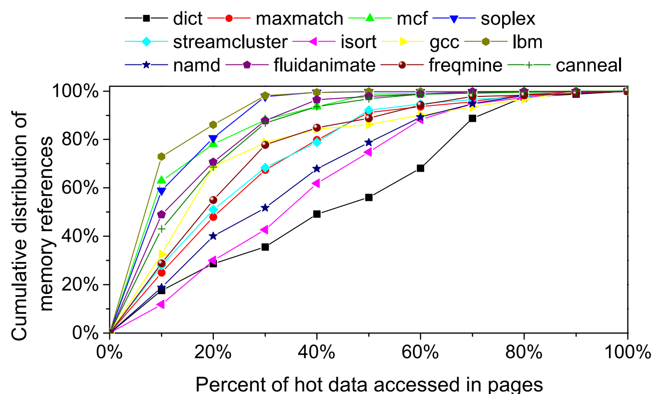

Fig. 1. Cumulative distribution of memory references (Y-axis) versus the percentage of hot data (X-axis).

The remainder of this paper is organized as follows. Section 2 introduces the background and experimental observations that motivate our system design. Section 3 presents the design and implementation of OAM mechanisms. Experimental results are presented in Section 4. We review the related work in Section 5 and conclude in Section 6.

2 Background and Motivation

A number of studies have proposed to organize DRAM and NVM in a cache/memory hierarchy [17], [18], however, this paper focuses on DRAM/NVM management in a flat-addressable architecture [6], in which DRAM and NVM are organized in a single (flat) address space and uniformly managed by OSes. The flat architecture can fully exploit the capacity of both DRAM and NVM, but relies on more sophisticated data placement policies to improve the performance and energy efficiency of hybrid main memories. More specifically, frequently accessed data should be placed in DRAM because NVM accesses incur relatively higher latency.

In the following, we use LLVM [19] to profile several representative applications in SPEC CPU2006 [20] and PBBS [21], and investigate the memory access statistics in term of data access frequency (hotness), application object size and lifetime. We have the following key observations.

Observation 1. For many applications, the hot data in each page only accounts for a small fraction of applications' total memory footprints, but leads to a large proportion of applications' total memory references. As shown in Fig. 1, the x-axis presents the percentage of frequently-accessed data traffic (in descending order of access frequency) in applications' all memory pages, and the y-axis presents the cumulative distribution of application's total memory references. For soplex, almost 35 percent hot data accounts for about 98 percent total memory references. This implies that migrating the fraction of hot data (variables and objects) in soplex is more benefical than migration of a whole page. However, for some applications such as dict, the memory accesses distribute evenly in the application's address space, and thus would benefit less from data migration.

Observation 2. A large portion of application objects are much smaller than a page. Fig. 2 shows the cumulative

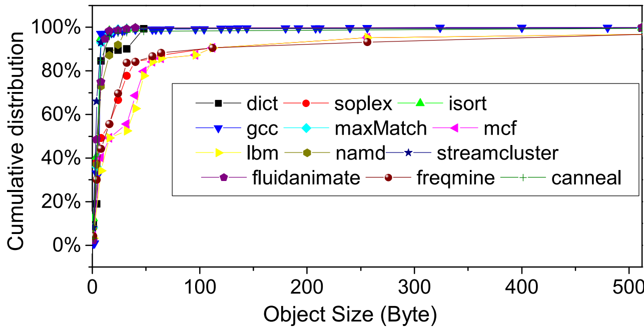

Fig. 2. Cumulative distribution function of objects' size.

distribution function of objects' size for these applications. We can find that over 95 percent of objects are less than 50 bytes for dict, isort, gcc, and maxMatch. For soplex, the fraction of small objects also exceed 85 percent. As the majority of application-level memory references are objects and variables, programmers can fully exploit the application semantic to optimize data placement/migration at the object granularity. In contrast, a page can contain many objects which may have different access behaviors, the synthesized page-level access statistics may become complicated and unpredictable.

Observation 3. A large portion of application objects show a long lifetime. Although previous studies have demonstrated that objects of many programs tend to have a relatively short lifetime [15], [22], [23], we find that there are still a large fraction of long-lived objects for some cases. Fig. 3 shows the cumulative distribution function of objects' lifetime for some applications. We find that almost all objects in gcc, isort, and soplex have a lifetime longer than 1000 seconds. Particularly, the lifetime of all objects in gcc is equal to the application's total execution time.

Observation 4. A portion of application objects show highly mutated memory access frequency (hotness) at different execution phases. Although many objects exhibits relatively consistent access behavior during their whole lifetime (so-called immutable objects), there are still a large portion of objects that show highly mutated access frequencies at different execution phases (so-called mutable objects), as shown in Table 1. This implies there is still large room to optimize data placement via object migration at runtime.

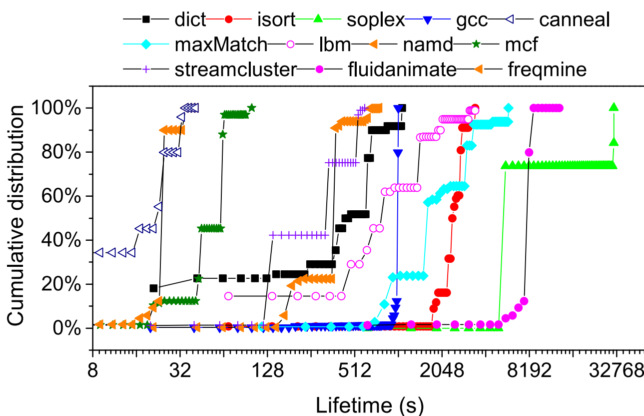

Fig. 3. Cumulative distribution function of objects' lifetime.

TABLE 1
The Proportion of Mutable Objects

Application	mutable objects(%)	Application	mutable objects(%)
dict	46%	isort	33%
maxmatch	44%	mcf	45%
gcc	42%	soplex	22%
lbm	32%	namd	9%
canneal	28%	fluidanimate	35%
freqmine	31%	streamcluster	12%

In summary, Observations 1 and 2 clearly show that data access monitoring and memory allocation/migration should be on objects, rather than pages. Observations 3 and 4 imply that a careful design on object migration is needed in hybrid memory systems.

3 Design and Implementation

In this Section, we first introduce the system overview of OAM, and then present the details of offline memory profiling, utility-based performance/energy model, object classification and execution phase identification, initial object memory allocation and runtime object migration mechanisms.

3.1 System Overview

Fig. 4 shows an overview of our Object-level memory Allocation and Migration schemes for hybrid memory systems. OAM mainly includes two modules: Offline Profiling and Code Instrumentation (*OPCI*) and Online Memory Allocation and Migration (*OMAM*). In order to place object appropriately on DRAM or NVM, OPCI exploits an offline application profiling tool to analyze memory access pattern at the granularity of objects, and then use an analytical performance/energy model to make an initial decision on object placement. This initial placement can significantly reduce unnecessary data migrations for the immutable objects. Moreover, for mutable objects, we need to determine when and where these objects should be migrated to further optimize the data placement at runtime. This requires more fine-grained memory access profiling to accurately identify different execution phases of applications.

Since DRAM is usually insufficient in hybrid memory systems, we should also consider DRAM resource utilization when an object is created. Although the utility model suggests that an object is preferred to be allocated in DRAM, we should make sure that there is enough DRAM resource for holding this object. Thus, the OMAM module appropriately allocates NVM or DRAM to an object according to current DRAM resource utilization and the object placement decisions.

As described in Fig. 4, we provide several application programming interfaces (*DRAM_alloc*, *NVM_alloc*, *DyMalloc* and *MigrateToX*) for object-level memory allocation/migration. *DRAM_alloc* and *NVM_alloc* are used to allocate DRAM and NVM, respectively. *DyMalloc* takes DRAM resource utilization into account to determine whether an object should be placed on DRAM or NVM. *MigrateToX* (*X* denotes DRAM or NVM) is used to migrate objects between DRAM and NVM. The details of *DyMalloc* and *MigrateToX* are described in Sections 3.5 and 3.6, respectively. We develop a static code instrumentation tool to modify the

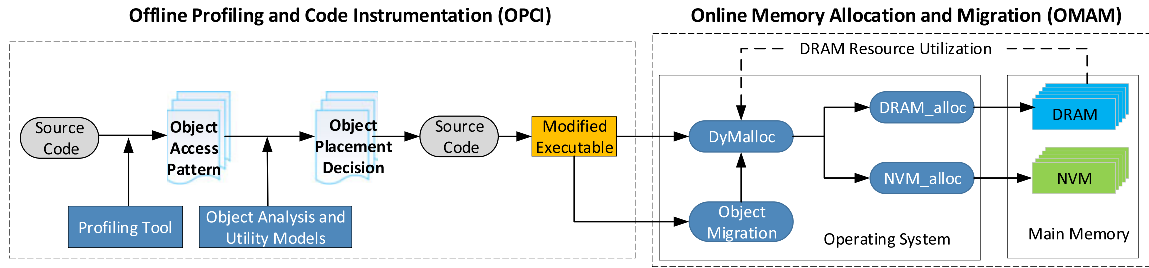

Fig. 4. An overview of object-level memory allocation and migration in hybrid memory system.

application source code with these APIs, and generate the modified executable codes automatically. In this way, OMAM migrates mutable objects between DRAM and NVM at runtime by the applications themselves, without modifications of hardware and OSes.

3.2 Object-Level Memory Access Profiling

We exploit the compiler infrastructure LLVM-3.8.1 to profile application's memory references. The generated trace file records every objects' memory access information, including access frequency, lifetime and size.

At first, we use LLVM to generate Intermediate Representation (IR) from the source codes. To count object-level memory references, we exploit LLVM PASS to traverse all load/store/malloc/calloc/mmap instructions and insert probes in the IR. We run the executable file generated from the modified IR, and collect all objects' access information in a trace file by LLVM PASS automatically. Generally, it is more costly to profile applications with more LOCs. However, if an application with less LOCs have more LOOP statements, its profiling cost may be even heavier than that of applications with more LOCs. Moreover, the trace size also depends on the input dataset. A large size of dataset usually generates more memory trace. We note that our profiling scheme still works if there are only binary codes available. Since OAM tracks memory allocation/access on the basis of LLVM IR, we can exploit some other tools such as "llvm-mctoll" to statically (ahead-of-time compilation) translates binaries and executables to LLVM IR.

The memory trace contains 5-item tuples including "*memory access type*, *virtual address*, *object name*, *object type*, and *access time*". It is easy to count objects' total memory references and lifetime with the trace. Because an object may contain child objects, we identify the child objects by checking the parameters of *GetElementPtr* instructions in LLVM. For a given object, the number of memory references is the sum of its all child objects' memory access counts.

The trace file contains all memory allocation/access instructions, however, only a portion of them result in actual data traffic to the main memory. The reason is that the on-chip cache can filter a large amount of memory accesses to hot data. In order to evaluate the impact of cache filtering on actual memory access statistics, we develop a cache simulator that can model different cache architectures [24], including hierarchy, cache sizes, associativities and replacement policies. Generally, the cache size and associativity both have a significant impact on the cache performance. A cache with higher associativity usually suffers less cache misses, however, it increases power, chip area, and latency for checking whether the accessed data is hit on a cache set. On the other hand, a cache with a larger size improves the cache hit rate, at the expense of larger cost, power, chip area, and latency. Because the CPUs in our experiments have 25 MB 16-way set-associative LLC, we model a 16-way set-associative LLC managed by a pseudo-LRU replacement algorithm. We configure the LLC size assigned to applications to be 16 MB empirically in consideration of that a portion of LLC capacity may be occupied by the OS and system software. In practice, the cache parameters such as size, associativity can be easily configured in our cache simulator by users.

The cache simulator takes the trace file generated by LLVM as a input, and filters programs' all virtual addresses that are hit in the simulated cache. The remaining virtual addresses reflect cache misses or cache eviction operations, and result in actual memory accesses to main memory. Fig. 5 shows the distribution of all data accesses on the cache and main memory for different applications. We find that the on-chip cache is able to filter 68 percent of total memory references on average. For gcc, even 87 percent data accesses are filtered. As a result, applications' memory access patterns may be very different when the impact of cache filtering is considered. Some very hot data may be always hit in the cache, and only results in a very small number of data accesses to main memory. Thus, it is unnecessary to migrate those data from the NVM to the DRAM. With our cache filter, we can obtain real memory access statistics on the NVM, and avoid unnecessary object migrations at runtime.

3.3 Utility Model for Object Placement

Because the asymmetric performance and energy features between DRAM and NVM, performance and energy consumption are two major concerns when we decide to place an object on DRAM or NVM [9], [10], [15]. We propose an utility function to calculate the benefit of placing objects on DRAM or NVM during a given time period (T_i), as shown

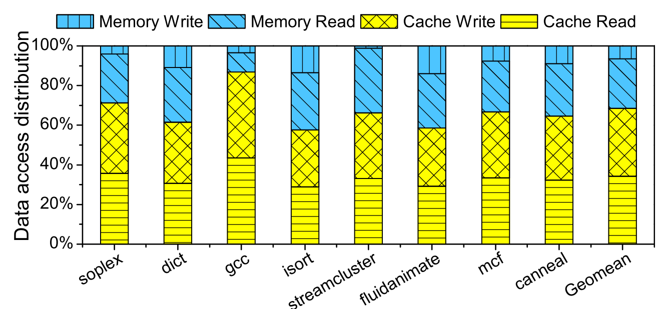

Fig. 5. Effect of cache filtering.

in Equation (1). This is a general model to take into account both memory performance and energy consumption.

$$U(X) = U_D(X) \cdot U_E(X) = \frac{D_{NVM}(X)}{D_{DRAM}(X)} \cdot \frac{E_{NVM}(X)}{E_{DRAM}(X)}, \quad (1)$$

where $U_D(X)$ and $U_E(X)$ reflect the utility of memory latency and energy consumption, respectively, $D_{NVM}(X)$ and $D_{DRAM}(X)$ are the total access delay of object X on NVM and DRAM, respectively. $E_{NVM}(X)$ and $E_{DRAM}(X)$ are the total energy consumption of object X on NVM and DRAM, respectively.

If applications need to meet certain performance constrains, we can simply set $U_E(X) = 1$ and only consider the utility of memory latency. Of course, we can only consider the memory energy consumption by setting $U_D(X) = 1$. In this paper, we exploit the energy-delay-product (EDP) model to balance memory energy consumption and access delay. For a given threshold ϵ ($\epsilon > 1$), $U(X) > \epsilon$ implies DRAM is a preferable site for placing object X, and a larger $U(X)$ means more benefit when placing object X on DRAM. In contrast, $U(X) < \epsilon$ implies the object X should be placed on NVM. Correspondingly, we deem object X as a hot object if $U(X)$ is larger than ϵ.

Generally, the total access delay D_σ of object X on a specific memory medium σ is mainly determined by memory access latency and total memory access counts. It can be calculated by Equation (2)

$$D_\sigma(X) = C_r(X) \cdot D_{\sigma,r} + C_w(X) \cdot D_{\sigma,w}, \quad (2)$$

where σ represents different memory medium (i.e., DRAM or NVM), and $C_r(X)$ and $C_w(X)$ are memory read and write counts in a given time period T_i, respectively. $D_{\sigma,r}$ and $D_{\sigma,w}$ are the average read and write latencies of the memory medium σ, respectively.

The total memory energy consumption E consists of dynamic and static energy consumption. The dynamic portion is caused by memory read and write operations, while the static portion is mainly attributed to refreshing operations for data retention. For DRAM, the static power is usually a constant, and the static energy consumption is mainly determined by the duration of time. For NVM, the static energy consumption is almost zero due to the non-volatility of NVMs. Given a period of time T_i, assume the object X has $C_r(X)$ read operations and $C_w(X)$ write operations on memory medium σ, respectively. Let $E_{\sigma,r}$ and $E_{\sigma,w}$ denote energy consumption of a single read and write operation on memory medium σ, respectively. The total energy consumption E of object X can be calculated by Equation (3)

$$E_\sigma(X) = C_r(X) \cdot E_{\sigma,r} + C_w(X) \cdot E_{\sigma,w} + E_{\sigma,static}. \quad (3)$$

3.4 Phase Identification and Object Classification

An application usually contains a number of objects, and different objects may present different memory access patterns. Basically, these objects can be classified into *immutable* and *mutable* objects. The memory access frequency of immutable objects can be simply deemed as a constant. For these objects, we only need to place them either on DRAM or on NVM based on our utility function, without considering object migration at runtime. In contrast, the memory access frequency of mutable objects dynamically changes at different execution phases, so a better object placement should also adapt to such changes. We thus exploit object migrations to achieve better energy-delay-product for those mutable objects.

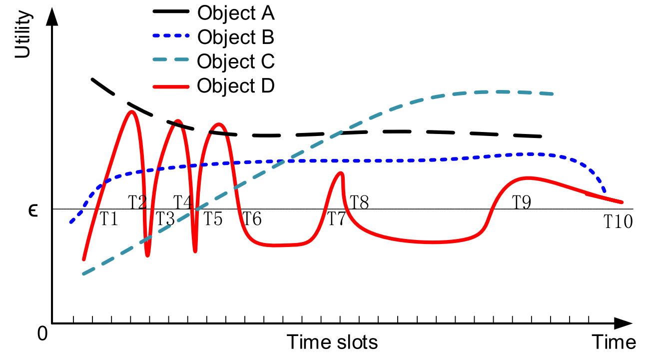

Fig. 6. An example of phase identification and object classification.

We describe application execution phase identification and object classification in the following. At first, we count read/write operations on each object in a fixed time slot. We calculate the utility of objects in each time slot according to Equation (1). Based on the threshold ϵ, we divide the objects' memory access pattern into different phases. We then find out all time slots in which the object's utility is larger than ϵ. If the object's utility values in two sequential time slots are larger than ϵ, these two time slots can be merged into one execution phase. However, some objects' utility values may fluctuate sharply around ϵ, and the duration of identified phases are rather short, as illustrated by object D in Fig. 6. In these cases, we would merge these short phases into other long phases if the duration of a phase is too short (for example, empirically set as one-twentieth of the object's lifetime). At last, we classify all objects into immutable and mutable objects according to the number of phases. For example, if an object only have one phase, the object is undoubtedly classified as an immutable object. Otherwise, the object is classified as a mutable object.

Fig. 6 illustrates an example of phase identification and object classification. Assume there are four objects A, B, C, and D, and the curves of utility function are shown in Fig. 6. The utility value of object A always exceeds ϵ, and thus can be classified as an immutable object. For object B, although it shows a very short phase in which its utility value is less than ϵ, this phase can be merged into another phase that spans most of its lifetime. As a result, object B is also classified as an immutable object. Object C is a mutable object because it shows two distinct phases according to our object classification scheme. For object D, we can divide its lifetime into 10 phases according to the utility function. The utility values of 5 phases (i.e., $T1 - T2$, $T3 - T4$, $T5 - T6$, $T7 - T8$, $T9 - T10$) are larger than ϵ. However, the duration of three phases ($T2 - T3$, $T4 - T5$, and $T7 - T8$) are very short, and thus we can merge them into the adjacent phases. At last, object D only have 3 phase ($T1 - T6$, $T6 - T9$ and $T9 - T10$) after phase merging operations. When a phase changes (i.e., at the time T6 and T9), we make trade-off between the benefit and cost of object migration. If the net benefit is positive, we insert an object migration statement at a proper place where the phase changes.

Finally, we make an initial decision on object placement. For immutable objects, if their average utilities are larger than ϵ, these objects would be placed on DRAM. For mutable objects, the average utility of its first phase determines the site where the object should be placed, and object migration should be performed at the time when a phase changes. In Fig. 6, object A, B and D should be allocated on DRAM at first. Object C should be allocated on NVM in the beginning, and then should be migrated to DRAM.

3.5 Initial Memory Allocation for Objects

In hybrid memory systems, we assume the expensive DRAM resource is usually limited and NVM is always sufficient for all workloads. Thus, the availability of DRAM resource is a prerequisite for successful DRAM allocation at runtime. We should take the DRAM resource utilization into account when allocating memory to an object. We adopt a memory reservation mechanism to allocate DRAM resource at runtime. The basic principle is to reserve enough DRAM resource for hotter objects in the near future when an object is created.

Let $U(O_i)$ denote the utility of object O_i, $S(O_i)$ denote the size of object O_i, $DRAM_{available}$ denote the available DRAM resource, and $DRAM_{reserved}$ denote the size of DRAM that is reserved for more hot objects. Let $U_{threshold}$ denote the threshold of utility value for which an object should be placed on DRAM. For a default threshold ϵ, only when $U(O_i) \geqslant \epsilon$, the object O_i would be placed on DRAM. Let $Utilization_{DRAM}$ denote the utilization of DRAM resource, and φ denote the threshold of DRAM utilization at which objects are apt to be placed on DRAM if $U(O_i) \geqslant \epsilon$.

Algorithm 1. DyMalloc

Input: $DRAM_{available}$, $DRAM_{reserved}$, $U(O_i)$, $S(O_i)$, $U_{threshold}$, ϵ, φ
1 **for** *each object* O_i **do**
2   **if** $U(O_i) < \epsilon$ **then**
3   Allocate O_i using NVM_alloc ;
4   **else**
5   **if** $Utilization_{DRAM} < \varphi$ **then**
6   Allocate O_i using $DRAM_alloc$;
7       $U_{threshold} \leftarrow \epsilon$;
8   **else**
9   Find an object O_j in the next time slot where its utility $U(O_j)$ is the largest ;
10       $DRAM_{reserved} \leftarrow S(O_j)$;
11   **if** $(DRAM_{available} - DRAM_{reserved}) \geqslant S(O_i)$ **then**
12   Allocate O_i using $DRAM_alloc$;
13         $U_{threshold} \leftarrow U(O_i)$ // update the threshold to place only hotter objects on DRAM;
14   **else**
15   Allocate O_i using NVM_alloc;

Algorithm 1 presents the details of memory allocation for objects (*DyMalloc*). For all objects whose utility values are less than the default threshold ϵ, they should be placed on NVM absolutely. For other objects ($U(O_i) \geqslant \epsilon$), if current DRAM resource is sufficient ($Utilization_{DRAM} < \varphi$), we place these objects on DRAM to improve the DRAM utilization. However, when the DRAM resource utilization becomes high ($Utilization_{DRAM} \geqslant \varphi$), we find a object O_j whose utility $U(O_j)$ is the largest in the next time slot, and reserve DRAM resource for the hotter object. If the object O_i is successfully allocated on DRAM, we update the threshold $U_{threshold}$ with current object's utility value $U(O_i)$, and then in the future only objects whose utility values are much larger can be allocated on DRAM. This dynamic utility threshold setting allows our system to allocate DRAM only for hotter and more beneficial objects when the DRAM resource is under high pressure.

We note that the application source code should invoke the API $DyMalloc$ to allocate memory for each object. When the program begins to execute, $DyMalloc$ takes into account the DRAM resource utilization to allocate DRAM or NVM to objects finally. We extend the Glibc library to provide $DRAM_alloc$ and NVM_alloc APIs for hybrid memory systems. Accordingly, we modify Linux kernel to differentiate NVM pages from DRAM pages in the virtual address space (VMA). As a result, we divide the main memory into two regions logically. We mark NVM pages in the VMA, and provide a new branch NVM_mmap for NVM allocation.

3.6 Object Migration at Runtime

As described in Section 3.4, online migration of mutable objects is complementary to the initial object memory allocation. It can further improve object memory access performance and energy efficiency. In our system, we employ static code instrumentation to add object migration instructions in the application's source code. At runtime, the application itself performs object migrations, without any intervention of operation systems.

To insert object migration instructions in the application's source code, the primary question is how to find the accurate location in the source code. We observe that plenty of intensive memory accesses are mainly attributed to loop statements [25]. As a result, we divide the source code into several code fragments using loops as breakpoints. At first, we track objects' access counts and the timestamps at the beginning and the end of loops with LLVM. For each object, we calculate its utility values during the execution of loops. A sudden change of utility values at the beginning and the end of an loop implies that the memory access pattern has changed, and the object should be migrated to another kind of memory before the loop. We locate this specific loop statement with the timestamp. Finally, we insert object migration statements at the beginning of loops using LLVM.

Fig. 7 shows an example of code instrumentation for hybrid memory allocation and object migration. We traverse all the objects in the static single assignment (SSA) form based on abstract syntax tree (AST) in LLVM, and change all $malloc$ in the source codes with $DyMalloc$. We create objects using *shared_ptr*, a smart pointer that retains shared ownership of an object through a pointer. This allows several *shared_ptr* objects to share the same object's memory. The *shared_ptr* uses a control block to record all pointers pointing to the target object and the number of total references. For example, in Fig. 8, the two pointers pointing to object 3 are recorded in the control block. The managed object is destroyed only when the reference counter becomes zero. For the mutable objects, we use $MigrateToX$ API to copy the objects to another kind of memory, and then all stored pointers pointing to the managed object now should point to the new memory address, as illustrated in Fig. 8. The object migration instructions are inserted at a proper place (most likely before loop statements) where the objects' access

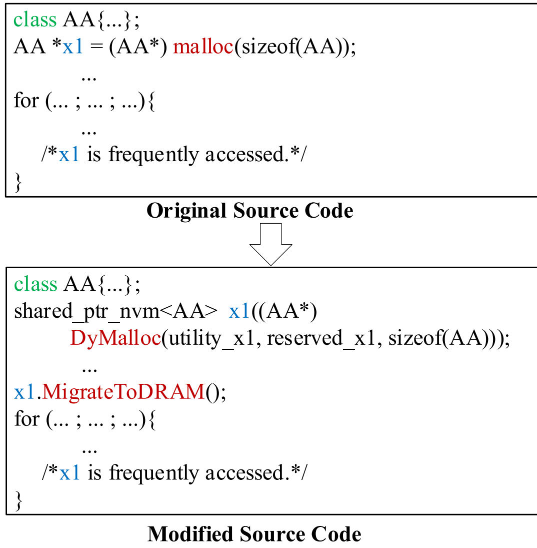

Fig. 7. An example of static code instrumentation.

pattern changes. In this way, we change the objects' virtual addresses after migration, and update all the references to the objects at runtime.

For large objects that are much bigger than a page, the coarse-grained migrations are relatively costly, and may even hurt application performance. To address this problem, we partition the large object into multiple chunks that are smaller than a page. The complier tool can profile memory access statistics for each chunk instead of the whole object. Based on the partitioning-based memory profiling, the memory allocation and migration schemes described above are still applicable to the data chunks. One challenging problem is that the *shared_ptr* only records all pointers pointing to the whole object, rather than the partitioned chunks. We thus extend the APIs of smart pointers to support runtime fine-grained memory reference tracking and access counting at the chunk level for large objects. In this way, our method can still optimize the data placement for large objects.

Object migrations at runtime would cause non-trivial performance overhead due to memory allocation for the new object, on-chip cache flushing and TLB shootdown associated with the old object [18], copying the object data, freeing the old object, and page table updating. However, the overhead can be mitigated by overlapping the data movement with the execution of applications. We use a helper thread to migrate the mutable objects before the execution of phase changes. This proactive object migration scheme takes the data movement off the critical path, and doesn't necessarily stall the applications.

We note that programmers are not required to be aware of the hardware features. They can still use the traditional *malloc*

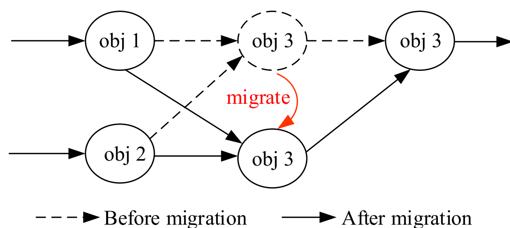

Fig. 8. An illustration of object migration.

TABLE 2
System Configuration

CPU	2 deca-core Intel Xeon(R) E5-2650 v3 @ 2.30 GHz, 25 MB LLC
DRAM	32 GB, 64 GB/s Bandwidth, 27 ns read latency, 57 ns write latency
PCM	32 GB, 32 GB/s Bandwidth, 39 ns read latency, 342 ns write latency
Power/Energy consumption	
DRAM	Voltage: 1.2V, Standby: 77 mA; refresh: 160 mA, pre-charge: 37 mA; Read/Write energy consumption: 1.17 pJ/bit and 0.39 pJ/bit;
PCM	Read/Write energy consumption: 2.47 pJ/bit and 16.82 pJ/bit;

API to program, and then exploit our offline profiling and code instrumentation tools to adapt the codes to hybrid memory systems. Our tools hide the programming difficulty in hybrid memory environments. Moreover, they can make an even better decision on object placement than programmers. Also, our OAM mechanism can easily transform many legacy applications to hybrid memory systems. The code size increased by OAM is mainly attributed to object migration statements, the control blocks for recording all pointers pointing to objects, and the helper thread for data movement. For most application in our experiments, the increased code size is less than 1 percent.

4 Evaluation

We evaluate the effectiveness of static memory allocation and dynamic migration of application objects incrementally.

4.1 Experimental Methodology

Testbed. As the commodity NVMs are still not available, we evaluate the proposed OAM mechanism using a hybrid memory emulator (HME) [26], [27], which use a portion of DRAM to emulate the performance features of NVM. HME exploits the DRAM thermal control interface provided by commodity Intel CPUs to limit the maximum memory bandwidth, and periodically injects additional software-created delay (the difference between NVM and DRAM latencies) to emulate the NVM access latency. Also, HME estimates total energy consumption of NVM by counting the joules consumed by each NVM read/write operation. Because applications are executed on real hardware, HME can run workloads with large memory footprints very fast. In our experiments, we choose PCM as the representative NVM because it has been widely studied. The timing and energy parameters of PCM are referred to the previous work [1], [10]. Table 2 presents the details of system configuration.

Benchmarks. Benchmarks are chosen from problem based benchmark suite (PBBS) [21], SPEC CPU2006 [20], Parsec 3.0 [28], Graph500 [29], and NUMA-stream [30]. We also evaluate OAM using some multi-programmed workloads, as shown in Table 3.

4.2 Effectiveness of Static Memory Allocation

At first, we evaluate application performance improved by the static memory allocation for objects, without considering object migrations, namely "OAM w/o migration". We compare it with page-interleaving, an NUMA memory allocation policy that places pages on DRAM and NVM alternately. We

TABLE 3
Benchmarks

PBBS	dict, isort, maxmatch
SPEC CPU2006	gcc, mcf, lbm, soplex, namd
PARSEC 3.0	streamcluster, fluidanimate, freqmine, canneal
Other Benchmarks	Graph500, NUMA-stream
mix1	namd + freqmine + canneal
mix2	fluidanimate + soplex + streamcluster
mix3	freqmine + canneal + fluidanimate + soplex
mix4	canneal + fluidanimate + soplex + streamcluster

also run all applications in a DRAM-only memory system, and the experimental results are referenced as an upper bound of performance optimization for hybrid memory systems.

Fig. 9 shows the execution time and memory energy consumption of 16 applications using "OAM w/o migration", "2PP w/o migration" and "DRAM-only" policies, all normalized to the page-interleaving policy. "OAM w/o migration" scheme can reduce applications' execution time and memory energy consumption by 33 and 35 percent on average, respectively. "2PP w/o migration" shows a little more reduction of execution time and memory energy consumption than "OAM w/o migration" because 2PP exploits the global memory access statistics for the initial data placement. In contrast, OAM performs a fine-grained profiling scheme to characterize dynamic changes of memory access patterns into different execution phases, and place the mutable objects on the DRAM or NVM based on the utility of its first execution phase. When DRAM is sufficient for storing all frequently-accessed objects, the initial placement policy of 2PP achieves a little higher performance than that of OAM. However, 2PP can not identify some short phases of frequently accessing an object that is deemed as cold from a global view, and thus lose many opportunities for migrating those short-term hot objects to fast DRAM, as illustrated in Section 4.3 later.

We find that there are about 13 percent performance gap between "OAM w/o migration" and the DRAM-only policy because a large portion of cold data is still placed on NVM to save energy consumption. Compared to the DRAM-only system, "OAM w/o migration" is able to reduce memory energy consumption by 51 percent on average. These results demonstrate that our initial data placement policy of OAM is effective for improving the performance and energy efficiency of hybrid memory systems. The execution time of *lbm*

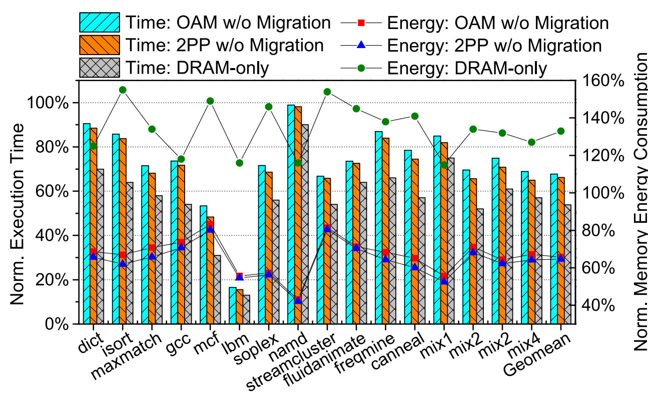

Fig. 9. Normalized execution time and energy consumption, all relative to the NUMA page-interleaving policy.

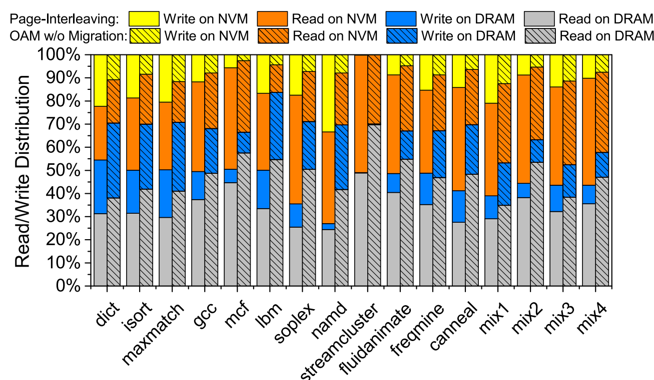

Fig. 10. Distribution of reads/writes on NVM and DRAM.

and *mcf* is significantly reduced by 83 and 46 percent, respectively, because they are both memory-intensive applications and hot objects are placed on fast DRAM with OAM. In contrast, the page-interleaving policy places about one half of hot data on slow NVM, and thus significantly degrades application performance. As *namd* is a CPU-bound application, and there are a few memory accesses during the execution, the static memory allocation scheme leads to very limited performance improvement, but reduces 56 percent of memory energy consumption. For the page-interleaving policy, we find that namd performs much more NVM write operations than that of lbm, as shown in Fig. 10. Because the energy consumption of NVM writes is about 40 times higher than that of DRAM writes, namd can achieve more energy saving than lbm by delivering more memory write operations from NVM to DRAM. For *dict* and *isort*, "OAM w/o migration" introduces very limited performance improvement because they show random memory access patterns and there are very few hot objects. Generally, our utility model guided data placement scheme can significantly improve the performance of memory-intensive applications with good data locality, and can also reduce memory energy consumption of applications by placing cold data on NVM.

To better understand the performance gain of our memory allocation scheme, Fig. 10 shows the distribution of read/write instructions on DRAM and NVM for each application. For the page-interleaving policy, memory accesses are almost evenly distributed on DRAM and NVM because pages are interleaved on the two memory nodes. However, we find over 80 percent total execution time of *lbm* is spent in accessing to NVM because NVM is several times slower than DRAM. For "OAM w/o migration", since hot objects are placed on DRAM, almost 70 percent memory read/write operations are distributed on DRAM. As a result, "OAM w/o migration" achieves much better application performance and lower energy consumption.

4.3 Effectiveness of Online Object Migration

In the following, we further evaluate the effectiveness of online object migration for mutable objects. Because C is not an object-oriented programming language, and our mechanism only provides object-level migration interfaces for applications that are written in C++, we only evaluate 6 applications written in C++ and 4 multi-programmed workloads.

Fig. 11 shows the execution time of these workloads using OAM with object migration, all normalized to OAM without

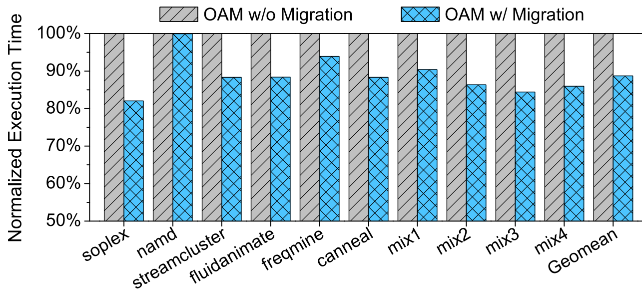

Fig. 11. Normalized execution time

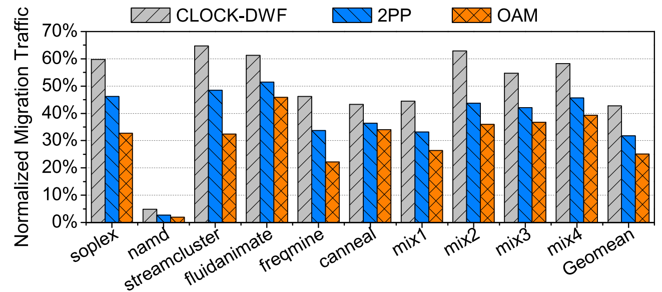

Fig. 13. Normalized data migration traffic.

object migration. Compared to the static memory allocation scheme, object migrations can further improve application performance by 11 percent on average. Our utility-based object placement model uses Energy-Delay Product as a metric to decide which kinds of memory objects should be placed on. As shown in Fig. 12, OAM without migration can achieve 51 percent reduction of EDP on average, and OAM with migration can further reduce EDP by 10 percent on average, all relative to the page-interleaving policy. This implies that many hot objects are already placed on DRAM during the initial memory allocation. However, the mutable objects can still benefit more from object migration at runtime.

4.4 Comparison of OAM With Page Migration Schemes

To evaluate the efficiency of object-granularity data migration, we compare OAM with two state-of-the-art page migration schemes– CLOCK-DWF [5] and 2PP [16]. CLOCK-DWF is a write recency-aware page replacement algorithm for hybrid memory architectures [5]. 2PP is a memory management system that combines static object placement with dynamic page migration [16]. We also refer to a DRAM-only system as the upper boundary of application performance. We implement CLOCK-DWF and 2PP in a hybrid memory simulator–NVMain integrated with a fast x86-64 simulator–zsim [18]. We note that 2PP exploits an in-house hardware/software coordinated platform-HMTT to collect memory trace of applications [16]. Instead, we use LLVM as a replacement of HMTT for the offline memory profiling.

These systems all use 4 KB page for migration. Because CLOCK-DWF and 2PP all rely on architectural simulators to model the hybrid memory management, it is extremely costly to run programs from the beginning to the end in the simulator. Instead, we only run programs with a fixed number of instructions (4×10^{10}), and then report the performance metric instructions per cycle (IPC). In this way, we can compare all results from simulator (CLOCK-DWF and 2PP) and real systems (OAM and DRAM-only) using the same metric. We evaluate CLOCK-DWF and 2PP with the same configuration described in Section 4.1.

Fig. 13 shows the data migration traffic using CLOCK-DWF, 2PP and OAM, all normalized to the footprints of these workloads. OAM can significantly reduce the migration traffic by 42 and 22 percent on average compared to CLOCK-DWF and 2PP, respectively. For each write operation on NVM pages, CLOCK-DWF would first migrate the NVM pages to DRAM. Thus, CLOCK-DWF always guarantees that every write request is responded by a page in DRAM. This mechanism introduces many page migration operations whose costs may even larger than the gained benefits. Because 2PP exploits an offline profiling scheme to make a better initial placement for objects, it is able to reduce the data traffic of *page* migration by 26 percent on average. However, 2PP still performs data migration at the granularity of pages, which is mostly more coarse-grained than objects. In contrast, OAM only migrates mutable objects when the memory access pattern changes, and thus can further reduce the migration traffic by 22 percent on average.

Fig. 14 shows that OAM can also improve applications' IPC by 9 and 4 percent on average compared to CLOCK-DWF and 2PP, respectively. All results are normalized to the system using only DRAM. We find that the performance gap between OAM and the DRAM-only system is only 5 percent on average. This implies that most data are placed on the right memory medium at a proper time, and the performance overhead of object migration is much less than 5 percent. For CPU-intensive applications such as *namd*, *streamcluster*, and *fluidanimate*, the normalized IPC of those schemes are similar to each other because these applications show very good data locality. For *freqmine*, *canneal*, and other multi-programmed workloads, OAM achieves much higher performance improvement than CLOCK-DWF and 2PP. OAM outperforms CLOCK-DWF and 2PP by up to 22

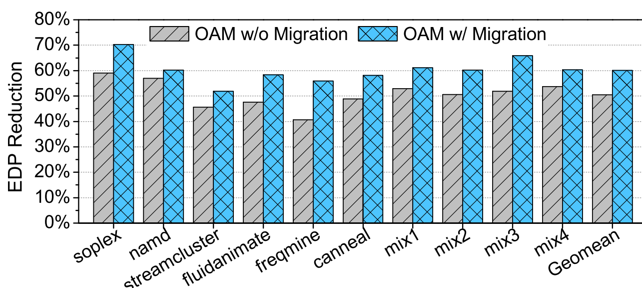

Fig. 12. Reduction of Energy Delay Product (EDP).

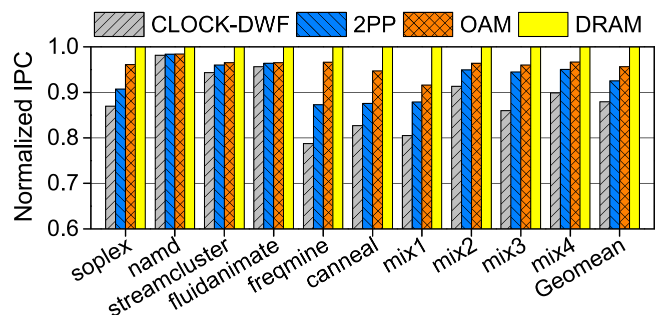

Fig. 14. IPC normalized to the DRAM-only system.

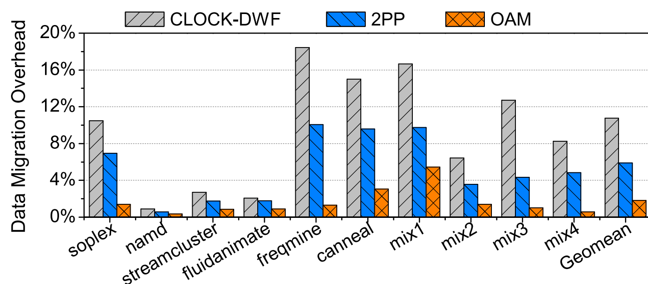

Fig. 15. Normalized data migration overhead.

and 10 percent. Because the footprint of these workloads are very large and more hot data should be migrated to DRAM. Object migration provided by OAM is much more lightweight than the page migrations, and thus introduces less performance overhead than CLOCK-DWF and 2PP.

Fig. 15 shows the "pure" data migration overhead caused by CLOCK-DWF, 2PP and OAM, all normalized to the applications' total execution time. As OAM analyzes the object-level memory access pattern in the offline profiling stage, to make a fair comparison, the runtime cost of page access counting and hot page identification for CLOCK-DWF and 2PP are not included in the results. The overhead includes memory allocation for the new object, on-chip cache flushing and TLB shootdown related to the old object, copying the object data, freeing the old object, and page table updating. As CLOCK-DWF and 2PP introduce more data migration operations at the page granularity, OAM can significantly reduce the data migration overhead by 83 and 69 percent on average compared to CLOCK-DWF and 2PP, respectively.

4.5 Adaptability to Different Datasets and Scales

In the following, we evaluate the adaptability of OAM to different datasets and problem scales using Graph500 and NUMA–STREAM as case studies. Graph500 reflects very poor temporal/spatial data locality and thus can be exploit to evaluate the memory access latency. NUMA-stream benchmark is a multi-threaded program particularly designed for evaluating the memory bandwidth of high performance computers.

At first, we evaluate the adaptability of OAM using different datasets. We run Graph500 with different inputs, using data placement and migration decisions generated by a relatively small dataset. Fig. 16 shows the execution time of Graph500 with different inputs, all relative to the page-interleaving scheme. The input "Graph_n" denotes a graph with 2^n vertices. Particularly, "Graph_26" contains about 64 million vertices and 16 GB raw data. With the increase of dataset sizes, OAM shows more reduction of execution time. Because

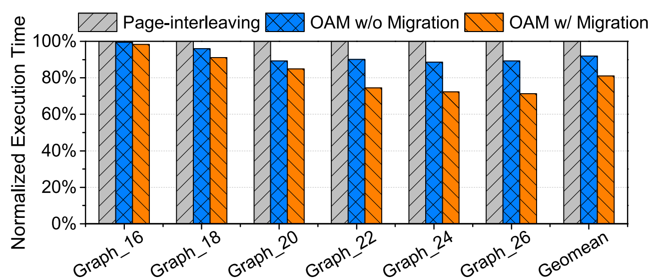

Fig. 16. Execution time of Graph500.

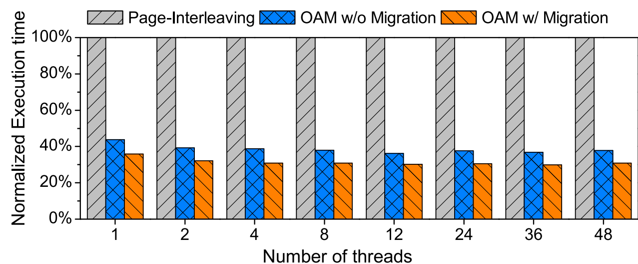

Fig. 17. Execution time of NUMA-stream.

larger inputs can increase memory references exponentially, and more data accesses can benefit from our memory allocation and object migration schemes correspondingly. Compared to the page-interleaving scheme, OAM with migration can improve application performance by up to 29 percent. This implies that our profiling-based OAM scheme is also effective for different datasets and scales.

Second, we study whether OAM is scalable for different problem of scales. Fig. 17 shows the normalized execution time of multi-threaded NUMA-stream with increasing threads. For NUMA-stream, more threads imply larger memory bandwidth consumption. However, all threads in NUMA-stream with different inputs show relatively stable performance. This implies that the OAM scheme is also effective when the problem scale changes.

4.6 Sensitivity to Different NVM Performance Features

Recently, Intel has announced the only commercially available NVM device–Intel Optane DC Persistent Memory Module. Its read latency is about 169-305ns, about 2-3 times higher than that of DRAM (81 ns). Surprisingly, the write latency of Optane DC is measured as 94 ns compared to 86 ns for DRAM [31]. Its read and write bandwidths are about 2.4-7.6 GB/s and 0.5-2.3 GB/s per DIMM, respectively. In this section, we explore the sensitivity of OAM to real NVM devices by setting the NVM read/write latencies and bandwidth according to the performance features of Intel Optane DIMMs.

Fig. 18 shows how the different setting of NVM access latencies affect the application performance. Since the write latency of Intel Optane memory is almost similar to that of DRAM, OAM would not migrate write-intensive objects to fast DRAM according to our utility models. As a result, the migration traffic is mainly attributed to read-intensive objects in the "Optane + DRAM" hybrid memory system. As shown in Fig. 19, the migration traffic is significantly reduced by using Intel Optane DIMMs as a replacement of PCM. However,

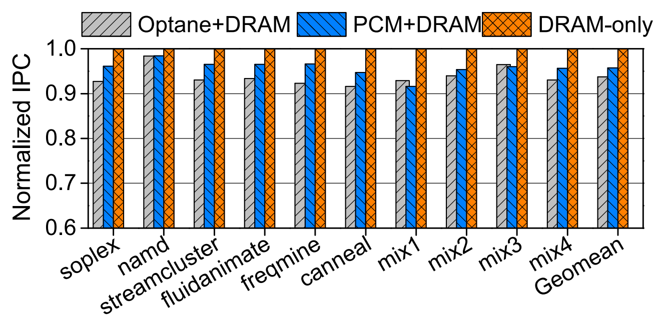

Fig. 18. OAM sensitive to different NVM features.

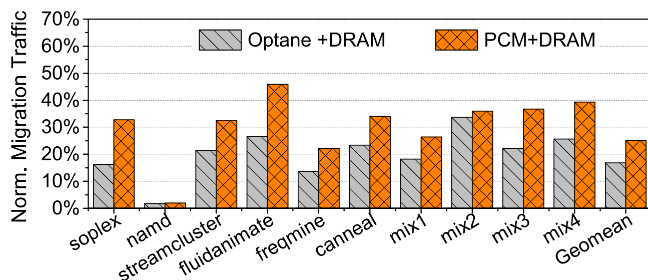

Fig. 19. Data migration traffic of OAM using different NVM devices, all normalized to the footprint of applications.

OAM achieves similar application performance for the two memory systems, because it can adapt to different performance characteristics of NVM devices to identify hot objects and place them on the proper memory medium.

5 Related Work

We summarize the related work in two categories.

5.1 Memory Allocation in Hybrid Memory Systems

There have been a number of studies on memory allocation for hybrid memory systems. Youngwoo *et al.* [32] propose a page allocator for hybrid memories based on access patterns of programs' memory segments. It simply assumes all data in heap and stack segments are frequently accessed (hot), and thus allocates pages in the heap and stack segments on DRAM, while data in other segments is allocated on NVM. WAlloc [33] is a wear-aware memory allocator for reducing write operations on NVM. HEAPO [34], and Makalu [35] all provide programming interfaces for NVM allocation with data consistency guarantee. All these studies are orthogonal to OAM. They only provide programming interfaces for NVM, and leave decision making of data placement to programmers. OAM not only offers programming interfaces for hybrid memory systems, but also provides a profiling tool and an utility model to direct object memory allocation/migration.

Some studies propose offline profiling schemes to guide the initial data placement in hybrid memory systems [15], [36]. Hassan *et al.* [15] exploit an offline profiling tool to direct object memory allocation on hybrid memories. This work mainly targets to embedded systems and applications. Dulloor *et al.* develop an offline profiling tool to classify memory access patterns into sequential, random and pointer chasing, and propose a data placement runtime called X-Mem [36] to map objects to different data structures, which are placed on separate memory regions. Wu *et al.* leverage online profiling and performance models to characterizes objects' access patterns, and develop a runtime system called *Unimem* [4] to place data objects on hybrid memories. However, Unimem is only applicable to MPI-based HPC applications that can be easily decomposed into different phases according to MPI operations. In contrast, OAM is applicable for all general-purpose applications written in C++. Tahoe [37] characterizes memory access patterns of task-parallel programs based on machine learning and analytical models, and make the best data placement decisions in hybrid memory systems. The object placement schemes proposed by these studies are only based on objects' global access characteristics. They do not consider the dynamic change of objects' access patterns in fine-grained execution phases at runtime.

2PP [16] is a software/hardware cooperative framework for object placement in hybrid memory systems. 2PP also exploits offline profiling to direct the initial object placement. However, the runtime data migration at the page granularity is relatively coarse-grained than objects, and still relies on hardware extension for page-level access monitoring. Moreover, 2PP only considers the global memory read-write ratios of applications and the available DRAM capacity for the initial data placement. If an object should be preferably allocated in the DRAM but there is not enough memory space available at runtime, it is still initially placed in the NVM but marked as divergent so that it can be migrated to DRAM when there is free DRAM available at runtime.

5.2 Data Migration in Hybrid Memory Systems

Page Access Counting. Page migration relies on page access monitoring to identify the hot (cold) pages. A large body of work relies on hardware extensions for page access counting. Gaurav *et al.* [6] propose a small cache in the memory controller to record write counts of selected pages, and migrate pages whose write counts exceed a given threshold from NVM to DRAM. Luiz *et al.* [8] propose hardware-assistant Rank-based Page Placement (RaPP) to rank pages according to page access frequency and write intensity, and migrates top-ranked pages to fast DRAM. Meswani *et al.* [38] propose a hardware counter in TLB which assists OS to identify hot pages for migration in hybrid memory architectures. These studies all require to modify/extend the hardware for page access monitoring.

Page Migration Algorithms. Soyoon *et al.* [5] argue that the frequency of memory writes is better than the temporal locality in predicting future memory writes, and propose a page replacement algorithm called CLOCK-DWF. Reza *et al.* [9] consider both memory writes and reads to model the benefit of page migration, and select the victim pages using two Least Recently Used (LRU) queues for DRAM and NVM individually. There are also some other page replacement algorithms based on LRU, such as LRU-WPAM [39], MHR-LRU [7]. Yoon *et al.* [40] propose to place data that frequently misses in row buffers on DRAM to speed up data accesses. Yang *et al.* [10] take page access frequency, row buffer locality and memory level parallelism into consideration, and propose a page-utility based performance model to direct page migrations in hybrid memory systems. Khouzani *et al.* [11] take both program segments and DRAM conflict into consideration to allocate/migrate pages in hybrid memory systems. A key limitation of the above page migration approaches is that they all rely on significant hardware modification to monitor memory access statistics. These studies also ignore application-level semantics and migrate pages according to temporal memory access patterns, which may result in unnecessary page migrations. In contrast, our proposal avoids any hardware and OS level modifications, and provides global optimization on data placement at the object granularity.

There is only a few work of page migration implemented in OSes. Modern OSes such as Linux only support some page migration primitives in Non-Uniform Memory Access (NUMA) architectures [41]. Zhang *et al.* [42] exploit an OS-level page migration scheme to improve NVM write performance

and endurance. Memos [43] introduces an OS-level page access profiling module and page migration engine to optimize data placement in hybrid memory systems. Memif [44] is a protected OS service for asynchronous, DMA-accelerated page migration in hybrid memory systems. HeteroOS [45] provides an OS-managed hybrid memory migration solution in virtualization environments. These studies all need to overhaul OS kernel mechanisms to support page migration, and the software overhead is usually rather high. In contrast, OAM provides an offline profiling tool to analyze objects' memory access patterns, and a static code instrumentation tool to modify the application source code, without any intervention of OSes and hardware. OAM also avoids the performance overhead of memory monitoring at runtime.

6 Conclusion

To mitigate the overhead of online page access monitoring and the cost of page migration in hybrid memory systems, we propose object-level memory allocation and migration mechanisms. OAM exploits offline profiling techniques to collect application memory access statistics, and classify the objects as stable and variable objects according to their memory access patterns. We propose an utility model that takes into account both memory access performance and energy consumption to direct the object placement on NVM or DRAM. For stable objects, the model-directed initial memory allocation is good enough. For variable objects, as their memory access patterns may change at different execution phases, OAM identifies the change of different phases and migrates objects to the other kind of memory accordingly. Experimental results show that OAM can significantly reduce data migration overhead by 83 and 69 percent compared to the state-of-the-art page migration schemes CLOCK-DWF and 2PP, respectively, while delivering higher application performance.

Acknowledgments

This work was supported by the National Key Research and Development Program of China under Grant No. 2017YFB1001603, and National Natural Science Foundation of China under Grant No.61672251, 61732010, and 61825202, 61929103.

References

[1] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecting phase change memory as a scalable dram alternative," in *Proc. 36th Annu. Int. Symp. Comput. Archit.*, 2009, pp. 2–13.

[2] M. Arjomand, M. T. Kandemir, A. Sivasubramaniam, and C. R. Das, "Boosting access parallelism to PCM-based main memory," in *Proc. 43rd Int. Symp. Comput. Archit.*, 2016, pp. 695–706.

[3] L. Zhang, B. Neely, D. Franklin, D. Strukov, Y. Xie, and F. T. Chong, "Mellow writes: Extending lifetime in resistive memories through selective slow write backs," in *Proc. 43rd Int. Symp. Comput. Archit.*, 2016, pp. 519–531.

[4] K. Wu, Y. Huang, and D. Li, "Unimem: Runtime data managementon non-volatile memory-based heterogeneous main memory," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2017, pp. 58:1–58:14.

[5] S. Lee, H. Bahn, and S. H. Noh, "CLOCK-DWF: A write-history-aware page replacement algorithm for hybrid PCM and DRAM memory architectures," *IEEE Trans. Comput.*, vol. 63, no. 9, pp. 2187–2200, Sep. 2014.

[6] G. Dhiman, R. Ayoub, and T. Rosing, "PDRAM: A hybrid pram and dram main memory system," in *Proc. 46th Annu. Des. Autom. Conf.*, 2009, pp. 664–469.

[7] K. Chen, P. Jin, and L. Yue, "A novel page replacement algorithm for the hybrid memory architecture involving PCM and DRAM," in *Proc. 11th IFIP Int. Conf. Netw. Parallel Comput.*, 2014, pp. 108–119.

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

[9] R. Salkhordeh and H. Asadi, "An operating system level data migration scheme in hybrid DRAM-NVM memory architecture," in *Proc. Des. Autom. Test Eur. Conf. Exhib.*, 2016, pp. 936–941.

[10] Y. Li, S. Ghose, J. Choi, J. Sun, H. Wang, and O. Mutlu, "Utility-based hybrid memory management," in *Proc. IEEE Int. Conf. Cluster Comput.*, 2017, pp. 152–165.

[11] H. A. Khouzani, F. S. Hosseini, and C. Yang, "Segment and conflict aware page allocation and migration in DRAM-PCM hybrid main memory," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 36, no. 9, pp. 1458–1470, Sep. 2017.

[12] J. Gandhi, A. Basu, M. D. Hill, and M. M. Swift, "BadgerTrap: A tool to instrument x86–64 TLB misses," *SIGARCH Comput. Archit. News*, vol. 42, no. 2, pp. 20–23, Sep. 2014.

[13] N. Agarwal and T. F. Wenisch, "Thermostat: Application-transparent page management for two-tiered main memory," in *Proc. 22nd Int. Conf. Archit. Support Program. Lang. Operating Syst.*, 2017, pp. 631–644.

[14] B. Pham, J. Veselý, G. H. Loh, and A. Bhattacharjee, "Large pages and lightweight memory management in virtualized environments: Can you have it both ways?" in *Proc. 48th Int. Symp. Microarchit.*, 2015, pp. 1–12.

[15] A. Hassan, H. Vandierendonck, and D. S. Nikolopoulos, "Software-managed energy-efficient hybrid DRAM/NVM main memory," in *Proc. 12th ACM Int. Conf. Comput. Frontiers*, 2015, pp. 23:1–23:8.

[16] W. Wei, D. Jiang, S. A. McKee, J. Xiong, and M. Chen, "Exploiting program semantics to place data in hybrid memory," in *Proc. Int. Conf. Parallel Archit. Compilation*, 2015, pp. 163–173.

[17] 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.*, 2009, pp. 24–33.

[18] H. Liu *et al.*, "Hardware/software cooperative caching for hybrid DRAM/NVM memory architectures," in *Proc. Int. Conf. Supercomputing*, 2017, pp. 26:1–26:10.

[19] LLVM, 2017. [Online]. Available: https://llvm.org/

[20] SPECCPU2006, 2006. [Online]. Available: https://www.spec.org/cpu2006

[21] PBBS, 2014. [Online]. Available: http://www.cs.cmu.edu/ pbbs/

[22] J. Chang, W. H. Lee, and W. Srisa-an,"A study of the allocation behavior of C++ programs," *J. Syst. Softw.*, vol. 57, no. 2, pp. 107–118, 2001.

[23] J.-S. Kim and Y. Hsu, "Memory system behavior of Java programs: Methodology and analysis," *SIGMETRICS Perform. Eval. Rev.*, vol. 28, no. 1, pp. 264–274, Jun. 2000.

[24] N. Beckmann and D. Sanchez, "Modeling cache performance beyond LRU," in *Proc. Int. Symp. High Perform. Comput. Archit.*, 2016, pp. 481–493.

[25] B. Li, L. Wang, and H. Leung, "Profiling selected paths with loops," *Sci. China Inf. Sci.*, vol. 57, no. 7, pp. 1–15, Jul. 2014.

[26] HME, 2017. [Online]. Available: https://github.com/CGCL-codes/HME

[27] Z. Duan, H. Liu, X. Liao, and H. Jin, "HME: A lightweight emulator for hybrid memory," in *Proc. IEEE Conf. Des. Autom. Test Europe*, 2017, pp. 1375–1380.

[28] PARSEC3.0, 2011. [Online]. Available: http://parsec.cs.princeton.edu/publications.htm

[29] GRAPH500, 2017. [Online]. Available: http://graph500.org/

[30] STREAM, 2015. [Online]. Available: http://www.cs.virginia.edu/stream/

[31] J. Izraelevitz *et al.*, "Basic performance measurements of the Intel optane DC persistent memory module," *CoRR*, vol. abs/1903.05714, 2019. [Online]. Available: http://arxiv.org/abs/1903.05714

[32] Y. Park, S. K. Park, and K. H. Park, "Linux kernel support to exploit phase change memory," in *Proc. Linux Symp.*, 2010, pp. 217–224.

[33] S. Yu, N. Xiao, M. Deng, F. Liu, and W. Chen, "Redesign the memory allocator for non-volatile main memory," *ACM J. Emerg. Technol. Comput. Syst.*, vol. 13, no. 3, pp. 49:1–49:26, Apr. 2017.

[34] T. Hwang, J. Jung, and Y. Won, "HEAPO: Heap-based persistent object store," *ACM Trans. Storage*, vol. 11, no. 1, pp. 3:1–3:21, Dec. 2014.

[35] K. Bhandari, D. R. Chakrabarti, and H.-J. Boehm, "Makalu: Fast recoverable allocation of non-volatile memory," in *Proc. ACM SIGPLAN Int. Conf. Object-Oriented Program. Syst. Lang. Appl.*, 2016, pp. 677–694.

[36] S. R. Dulloor *et al.*, "Data tiering in heterogeneous memory systems," in *Proc. 11th Eur. Conf. Comput. Syst.*, 2016, pp. 15:1–15:16.
[37] 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.*, 2018, pp. 31:1–31:13.
[38] M. R. Meswani, S. Blagodurov, D. Roberts, J. Slice, M. Ignatowski, and G. H. Loh, "Heterogeneous memory architectures: A HW/SW approach for mixing die-stacked and off-package memories," in *Proc. IEEE 21st Int. Symp. High Perform. Comput. Archit.*, 2015, pp. 126–136.
[39] H. Seok, Y. Park, K.-W. Park, and K. H. Park, "Efficient page caching algorithm with prediction and migration for a hybrid main memory," *ACM SIGAPP Appl. Comput. Rev.*, vol. 11, no. 4, pp. 38–48, Dec. 2011.
[40] H. Yoon, "Row buffer locality aware caching policies for hybrid memories," in *Proc. IEEE 30th Int. Conf. Comput. Des.*, 2012, pp. 337–344.
[41] C. Lameter, "Local and remote memory: Memory in a Linux/NUMA system," in *Proc. Linux Symp.*, 2006, pp. 1–25.
[42] W. Zhang and T. Li, "Exploring phase change memory and 3D die-stacking for power/thermal friendly, fast and durable memory architectures," in *Proc. 18th Int. Conf. Parallel Archit.s Compilation Techn.*, 2009, pp. 101–112.
[43] L. Liu, H. Yang, Y. Li, M. Xie, L. Li, and C. Wu, "Memos: A full hierarchy hybrid memory management framework," in *Proc. IEEE 34th Int. Conf. Comput. Des.*, 2016, pp. 368–371.
[44] F. X. Lin and X. Liu, "Memif: Towards programming heterogeneous memory asynchronously," in *Proc. 21st Int. Conf. Archit. Support Program. Lang. Operating Syst.*, 2016, pp. 369–383.
[45] S. Kannan, A. Gavrilovska, V. Gupta, and K. Schwan, "HeteroOS: OS design for heterogeneous memory management in datacenter," in *Proc. 44th Annu. Int. Symp. Comput. Archit.*, 2017, pp. 521–534.

Haikun Liu (Member, IEEE) received the PhD degree from the Huazhong University of Science and Technology (HUST), China. He was the recipient of Outstanding Doctoral Dissertation Award in Hubei province, China. He is currently an associate professor with the School of Computer Science and Technology, HUST. His current research interests include in-memory computing, virtualization technologies, cloud computing, and distributed systems.

Renshan Liu received the master's degree in computer science from the Huazhong University of Science and Technology (HUST). His research interest includes hybrid memory systems.

Xiaofei Liao (Member, IEEE) received the PhD degree in computer science and engineering from the Huazhong University of Science and Technology (HUST), China, in 2005. He is currently a professor with the School of Computer Science and Engineering, HUST. His research interests include the areas of system virtualization, system software, and cloud computing.

Hai Jin (Fellow, IEEE) received the PhD degree in computer engineering from the Huazhong University of Science and Technology, China, in 1994. He is a Cheung Kung Scholars chair professor of computer science and engineering at the Huazhong University of Science and Technology (HUST), China. In 1996, he was awarded a German Academic Exchange Service fellowship to visit the Technical University of Chemnitz, Germany. He worked at The University of Hong Kong, Hong Kong between 1998 and 2000, and as a visiting scholar with the University of Southern California, Los Angeles, California between 1999 and 2000. He was awarded Excellent Youth Award from the National Science Foundation of China in 2001. He is the chief scientist of ChinaGrid, and the chief scientists of National 973 Basic Research Program Project of Virtualization Technology of Computing System, and Cloud Security. He is a fellow of CCF, and a member of the ACM. He has co-authored 22 books and published more than 800 research papers. His research interests include computer architecture, cloud computing, big data processing, and network security.

Bingsheng He (Member, IEEE) received the bachelor's degree in computer science from Shanghai Jiao Tong University, Shanghai, China, and the PhD degree in computer science from the Hong Kong University of Science and Technology, Hong Kong. He is currently an associate professor with the School of Computer, National University of Singapore, Singapore. His research interests include high performance computing, cloud computing, and database systems. He has been awarded with the IBM PhD fellowship (2007–2008) and with NVIDIA Academic Partnership (2010–2011).

Yu Zhang (Member, IEEE) received the PhD degree in computer science from the Huazhong University of Science and Technology (HUST), Wuhan, China, in 2016. He is currently a postdoctor with the School of Computer Science, HUST. His research interests include big data processing, runtime system, computer architecture, and software.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/csdl.