Exploiting Program Semantics to Place Data in Hybrid Memory

Wei Wei*†, Dejun Jiang*, Sally A. McKee‡, Jin Xiong*, Mingyu Chen*
*Advanced Computer Systems laboratory, State Key Laboratory of Computer Architecture
Institute of Computing Technology, Chinese Academy of Sciences
†University of Chinese Academy of Scieneces
Beijing, China
‡Chalmers University of Technology
{weiwei01, jiangdejun, xiongjin, cmy}@ict.ac.cn, mckee@chalmers.se

***Abstract*—Large-memory applications like data analytics and graph processing benefit from extended memory hierarchies, and hybrid DRAM/NVM (non-volatile memory) systems represent an attractive means by which to increase capacity at reasonable performance/energy tradeoffs. Compared to DRAM, NVMs generally have longer latencies and higher energies for writes, which makes careful data placement essential for efficient system operation. Data placement strategies that resort to monitoring all data accesses and migrating objects to dynamically adjust data locations incur high monitoring overhead and unnecessary memory copies due to mispredicted migrations. We find that program semantics (specifically, global access characteristics) can effectively guide initial data placement with respect to memory types, which, in turn, makes run-time migration more efficient. We study a combined offline/online placement scheme that uses access profiling information to place objects statically and then selectively monitors run-time behaviors to optimize placements dynamically. We present a software/hardware cooperative framework, 2PP, and evaluate it with respect to state-of-the-art migratory placement, finding that it improves performance by an average of 12.1%. Furthermore, 2PP improves energy efficiency by up to 51.8%, and by an average of 18.4%. It does so by reducing run-time monitoring and migration overheads.**

I. Introduction

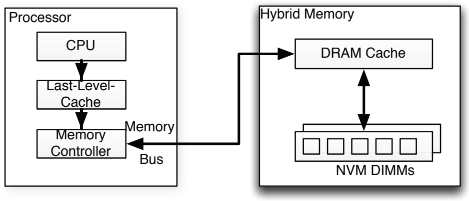
Server main memory currently accounts for up to 40% of energy, which is comparable to or even higher than the processor's contribution [16]. Reducing memory system energy is thus essential for reducing data-center operating costs. DRAM-only memory systems suffer high energy consumption from refresh operations and power leakage. Hybrid memory systems comprising DRAM and non-volatile memories (NVMs) have thus been proposed as an alternative for building large-capacity memories. Compared to DRAM, non-volatile memory technologies such as PCM [21], STT-RAM [9], and ReRAM [12] require no refresh and promise better scalability. Figure 1a and Figure 1b show two typical hybrid memory architectures. The *inclusive* organization [14], [23], [32] in the figure uses DRAM as a cache to hide high NVM access latencies, which lowers total capacity. The *exclusive* organization [34], [24], [15] in Figure 1b maps DRAM and NVM into a single physical address space. Here we focus on exclusive (flat) architectures that try to fully exploit the capacities of both DRAM and NVM.

Writes to nonvolatile memory technologies incur longer latency and higher energy than DRAM [18], [21], which makes good data placement essential for efficient hybrid memory organizations. In other words, write-intensive data should reside in DRAM, not NVM. Current hybrid-memory management techniques [34], [24], [15] let either the memory controller or the OS monitor page-access behaviors to identify "hot" data in NVM and then dynamically migrate it to DRAM (or cold DRAM data to NVM). Like other reactive memory management techniques (dynamic caching being the most pervasive), these approaches rely on past behavior to predict the future, and sometimes these predictions will be wrong. This approach to monitoring and prediction has two weaknesses: storage and computation costs of monitoring page behaviors increase linearly with working set sizes, and mis-identifying pages to migrate generates (costly) unnecessary data movement.

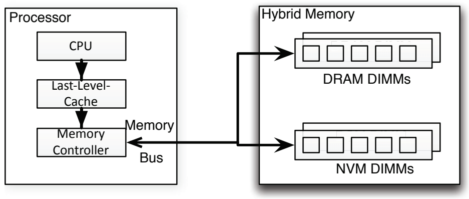
The good news is that for many applications (e.g., data mining and HPC), a consistent read/write behavior emerges to dominate data accesses across the application's lifetime, making runtime variations in access characteristics relatively minor in comparison. Instead of letting the compiler or OS generate a usage-oblivious initial data layout and relying solely on dynamic behaviors to determine hybrid-memory data placement, we take a more holistic view of memory behaviors, striving to generate a good initial placement based on global access characteristics. We find that an initial placement based on major access features can significantly reduce unnecessary memory copies due to mispredicted migrations. This holistic view frees us from the need to monitor all pages, allowing us to maximize the impact of dynamic tuning (and minimize overheads) by focusing on objects that will benefit most.

We make the following contributions:

- We exploit global access characteristics to guide initial data-object placements, which helps avoid the overheads of mispredicted dynamic migrations.
- We propose a better metric — the ratio of read to writes instead of simple write counts — for deciding when to migrate pages.



(a) Inclusive architecture



(b) Exclusive architecture

Figure 1: Hybrid memory organizations

- We show that the majority of accesses are due to heap references and that grouping heap objects according to where they were allocated in the source code allows us to treat all objects within a group similarly.
- We show that profiles we generate offline still work well when applications are run at different scales or with other input sets.
- We demonstrate 2PP, an experimental, hardware/software implementation that manages hybrid memory in two phases: after offline profiling and placement, we leverage profile information to guide selected run-time object monitoring and migration.

We find that our approach is more energy-efficient than systems that focus solely on dynamic page migration between DRAM and NVM. For instance, compared to RaPP [24], 2PP improves performance by an average of 12.1%. Furthermore, 2PP improves energy efficiency by up to 51.8%, and by an average of 18.4%.

II. Reference Behaviors

Most current data placement solutions [36], [24], [33], [15] dynamically migrate pages with high write counts from NVM to DRAM and those with low write counts from DRAM to NVM. These approaches rely on a small data-reference window to predict the best memory type, spurring unnecessary migrations for objects whose reference patterns fluctuate often. In general, migrations generate extra memory accesses that can reduce performance and increase energy. For instance, one recent study shows that migrations increase execution time by 25%, on average [4].

We choose 10 parallel applications to profile their reference behaviors. These applications are selected from 4 representative application categories (data mining, graph processing, image processing and HPC) which usually require large-capacity memory. Table I shows the application details. In the profiling, we run two problem sizes for each application, which corresponds to two memory capacity configurations 2GB and 8GB separately.[1] We find that global read-write profiles created offline largely reflect dynamic access behaviors. To demonstrate this, we divide application memory traces (arbitrarily) into 1000 subtraces, rank pages referenced in each subtrace according to the number of writes, and then normalize rankings to the total number of pages. Figure 2 shows the comparative "write intensiveness" of a sample page from *libsvm*. The graph shows that this page is rarely written within the subtraces around times 0, 4000, and 6000, but also that in many subtraces this page has the highest write ranking. The result is that the page's global write intensity for the entire execution trace is 0.93. We find that this global metric reflects major access characteristics for most data pages in our applications. This suggests that we may use global read-write profiles to initially place data into appropriate memory types to reduce dynamic migrations.

[1]Unless otherwise specified, we use small problem sizes.

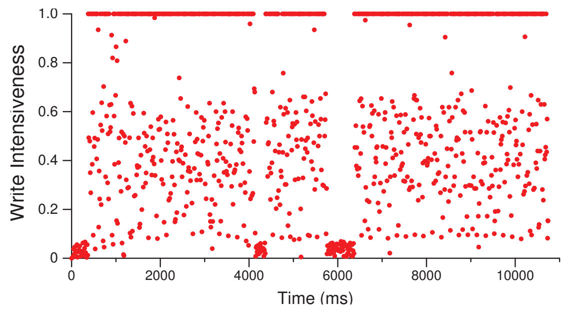


Figure 2: Write intensiveness of a sample page from *libsvm*

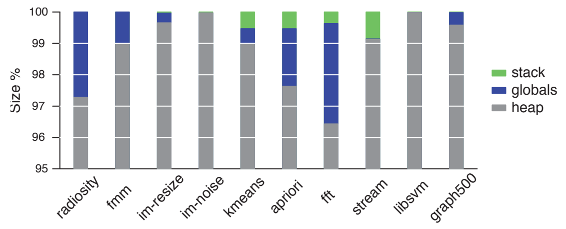


Figure 3: Relative sizes of different object types

A. Profiling Strategy

To implement this approach, we first need to choose the data granularity for profiling. Profiling based solely on physical pages means that different application runs generate different profiles. Instead, we can profile behaviors based on coarser-grained program semantics — analyzing code, global, stack, and heap segments separately. For instance,

Table I: Workload Features

Workload		Problem Size	Working Set [1] Size (MB)	Application Area
SPLASH-2x [30]	radiosity	S: 1.5e-3, Room L: 1.5e-4, Largeroom	877 1442	high performance computing
	fmm	S: 1M Particles, Timestep=5 L: 2M Particles, Timestep=5	474 939	high performance computing
Imagemagick [27] (im)	resize	S: Original Size:37MB resizing 4 times L: Original Size:227MB resizing 4 times	494 4096	image processing
	noise	S: Size of original image:100MB adding 200% noise L: Size of original image:908MB adding 200% noise	366 4812	image processing
NU-MineBench [20]	kmeans	S: 10,000,000 objects 9 features L: 50,000,000 objects 9 features	446 6619	data mining
	apriori	S: input file size: 334MB L: input file size: 834MB	668 1638	data mining
HPCC [19]	fft	S: N=10,000 L: N=40,000	424 6192	high performance computing
	stream	S: N=10,000 L: N=30,000	792 6896	high performance computing
libsvm [5]		S: Classes: 2 data: 30,956 features: 123 L: Classes: 2 data: 19,996 features: 1,355,191	31 1368	data mining
graph500 [1]		S: Scale=21,Edge factor=16 L: Scale=24,Edge factor=16	518 4112	graph processing

[1] The working set sizes are the total memory sizes consumed by applications when they run.

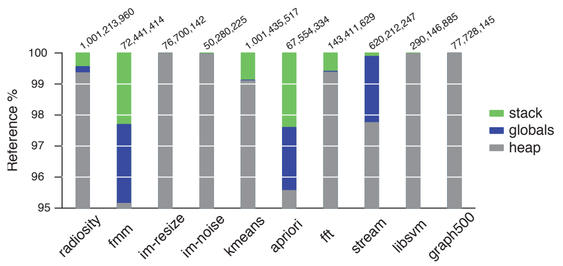


Figure 4: Reference proportions (total counts above bars)

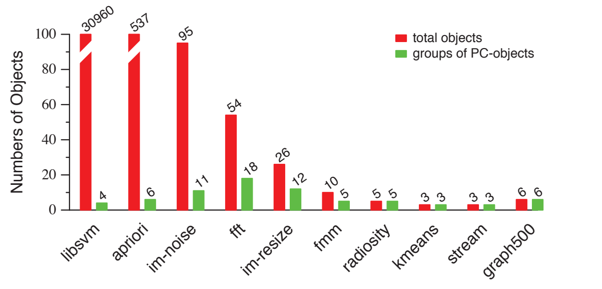


Figure 5: Object distributions

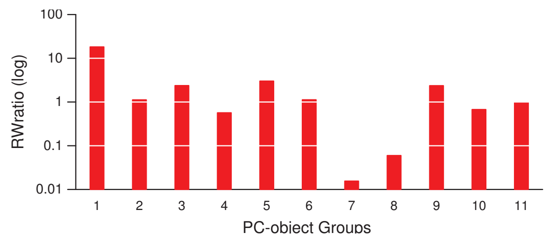


Figure 6: RWratios across *im-noise* PC-object groups

code segments are (almost always) read-only, which makes them good candidates for placement in NVM. Placing global, stack, and heap objects requires further analysis. Figure 3 and Figure 4 show workload access characteristics (object sizes of each segment with respect to total data footprint and the proportions of each type of reference). Figure 3 shows that heap data occupy most of the memory space in all 10 workloads. For example, in *fft*, stack and global data occupy just 3.6% of memory, leaving the heap to consume 96.4%. Likewise, Figure 4 shows that heap data references account for the vast majority of total references: they account for 95.57% (*fmm*) to 99.99% (*graph500*) in our applications. Given that stack and global data occupy relatively little memory, we first focus on optimizing heap object placement: heap references can dramatically affect hybrid memory system efficiency. Profiling pages within objects to see whether their access behaviors differ gives us further information on which to base placement decisions.

We use the number of reads versus writes (*RWratios*) to summarize reference behaviors. Analyzing RWratios of heap objects reveals relationships between programming semantics and memory access behaviors: objects allocated at the same place in the source code tend to exhibit similar reference behaviors. Figure 5 shows the total numbers of objects along with the classes we get when grouping objects by their allocation PCs (Program Counters). Again, taking *libsvm* as an example, there are 30960 heap objects, but when grouping them by allocation PC, we get only four object classes, and we find that all members of a class have similar RWratios (varying by at most 0.01 within any class). **We call such an object class as a group of PC-objects as these objects share the same PC.** Likewise, we find that objects in the same group in *im-noise* vary by at most 0.001, whereas those of objects within different groups vary much more widely. Figure 6 shows details for those objects. We observe similar phenomena across all our applications and workloads.

Examining accesses to the physical pages occupied by

Table II: Percentages of convergent objects

radiosity	fmm	im-resize	im-noise	kmeans
80.0	40.0	41.7	27.3	33.3
apriori	fft	stream	libsvm	graph500
50.0	61.1	33.3	75.0	66.7

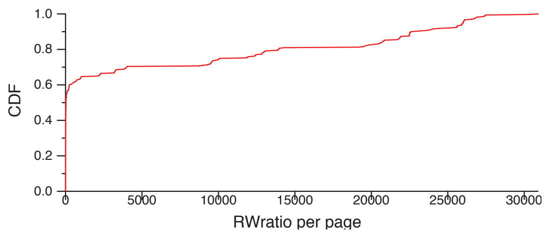


Figure 7: CDF of RWratios in the most divergent *libsvm* object

these PC-object groups lets us further categorize reference behaviors. **Given a group of PC-Objects, if the reference behaviors of all physical pages within this group are similar, we define the group as convergent PC-objects. Otherwise, we define the group as divergent PC-objects.** In our analyses here, we deem objects whose page RWratios vary by 1% or less to be convergent; all other objects are considered divergent.

For example, for *libsvm*, page RWratios from the most convergent object differ by at most 10^{-5}, whereas pages from the most divergent one have RWratios from 1.07 to 30896 (shown in Figure 7). Table II lists the proportion of convergent objects in each workload. Our data suggest that hybrid memory systems will perform efficiently when convergent PC-objects are statically placed according to RWratio. Divergent objects, on the other hand, may benefit from dynamic placement.

B. Changing Inputs and Workload Sizes

A good static placement should hold for different inputs and workload sizes. Four of our applications (im-resize, im-noise, kmeans, and libsvm) have multiple input sets. Table III summarizes the differences in the inputs we use for the four applications. Consider *im-resize* as an example: our two input images are similar in size, but they differ in content and resolution as well as in color range and saturation. Figure 8 shows deviations in RWratios across inputs for *im-resize*; for this application differences are less

Table III: Characteristics of different input sets

Applications	Original Input [1]			New Input [1]		
im-resize or im-noise	contents: people			contents: dog		
	resolution: 28.35×28.35			resolution:72×72		
	color	**mean**	**stdev**	**color**	**mean**	**stdev**
	overall	136.96	88.719	overall	115.08	74.98
kmeans	data features: 9			data features: 18		
libsvm	data features: 123			data features: 10		

[1] Different inputs have similar problem sizes.

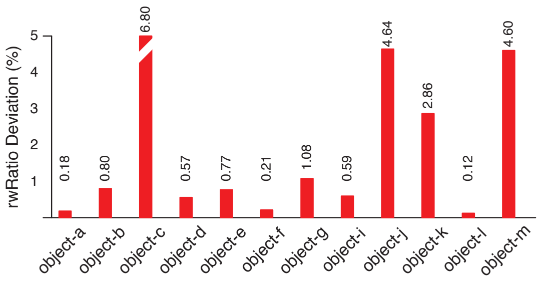


Figure 8: Changing input sets for *im-resize*

than 7% (deviations for the other three applications are even smaller).

Next we run our applications with the large problem sizes listed in Table I. The larger problem sizes are from 1.98 to 14.84 times larger than the small ones, depending on the application. RWratio deviations for objects in *graph500* are less than 5%: the largest deviation within an object group is 4.9%, the next is 2.1%, and the other four are less than 0.01%. Deviations for other applications are even smaller.

III. Placement

Our experimental analyses reveal that:

1) heap objects account for most application data accesses;
2) heap objects allocated from the same PC tend to share major access characteristics;
3) many heap objects sharing a PC are comprised of physical pages that share similar access behaviors, but others exhibit are comprised of pages whose access behaviors differ from the object's dominant access behaviors;
4) dominant heap object behaviors for our applications are stable across different input sets and input problem sizes[2].

These observations motivate two-phased, hybrid-memory placement strategies that generate an initial mapping based on the dominant access characteristics of different PC-object groups and then dynamically adapt those placements for highly referenced pages belonging to divergent PC-objects — pages whose access behaviors indicate that migration will increase efficiency.

Here we study one such strategy that we term *2PP*. In the first (static) phase, 2PP maps code segments to NVM, stack and global data to DRAM. **For heap objects, the basic placing unit is a group of PC-objects. 2PP maps groups of PC-objects to the most appropriate memory type based on their RWratios.** In the second (dynamic) phase, the memory controller (MC) only monitors RWratios of pages comprising divergent heap PC-objects, moving them between memory types to improve performance and/or reduce

[2]Note that we do not target interactive Internet services in this paper, whose reference behaviors differ. Instead, we focus on application categories whose process patterns are stable/fixed.

energy expenditures. 2PP's initial placement is guided by the observations above, but its success still relies on a low-overhead run-time framework. We first describe details of 2PP's static placement strategy and then describe its runtime system.

A. Placement Policy

Any hybrid memory placement strategy must optimize for a given metric. Many recent memory system studies use energy-delay product (ED) [26], [13], [18] to represent energy efficiency (lower is better); we adopt it for ease of comparison and because here we focus on energy efficiency. Note that 2PP can be adapted to use other metrics in order to focus on different system aspects (e.g., energy or performance).

We calculate ED as in Equation 1. Here RE_1 (WE_1) and RE_0 (WE_0) represent the read (write) energies for a given data object/page in DRAM and NVM, respectively. Likewise, RD_1 (WD_1) and RD_0 (WD_0) represent the corresponding read (write) delays. R_N and W_N represent the corresponding counts. Our 2PP implementation strives to lower ED values (Algorithm 1).

$$ED_i = (RE_i \times R_N + WE_i \times W_N) \times (RD_i \times R_N + WD_i \times W_N)$$
$$i = \begin{cases} 1 \text{ means the memory type is DRAM} \\ 0 \text{ means the memory type is NVM} \end{cases} \tag{1}$$

Algorithm 1

```
1: for each PC-object do
2:     if $ED_{DRAM} \geq ED_{NVM}$ then
3:         place in NVM
4:     else
5:         place in DRAM
6:     end if
7: end for
```

$$\frac{ED_i}{(W_N)^2} = (RE_i \times RWratio + WE_i \times 1) \times (RD_i \times RWratio + WD_i \times 1) \tag{2}$$

We normalize all accesses to W_N and then calculate normalized ED (Equation 2) for both DRAM and NVM for each heap object class, trying to place objects in whichever memory type has lower ED (or in NVM if they are equal). The final RE_i, WE_i, RD_i, and WD_i values are constant for a given hybrid memory system. ED thus depends only on the RWratios. Replacing the original ED with the transformed ED value gives us Algorithm 2, which leads to Algorithm 3. The threshold value for Algorithm 2 is constant for a given memory system. If an object prefers to be allocated in DRAM but there is no space available (**e.g., space contention in multiprogramming environment**), we simply allocate it in NVM but mark it as divergent, so that we migrate it to DRAM at run time (and vice versa). This addresses capacity problems.

Algorithm 2

```
1: for each PC-object do
2:     if $RWratio \geq threshold$ then
3:         place in NVM
4:     else
5:         place in DRAM
6:     end if
7: end for
```

Algorithm 3 Placement Policy

```
1: Offline:
2: for each PC-object do
3:     if $RWratio \geq threshold$ then
4:         allocate in NVM
5:     else
6:         allocate in DRAM
7:     end if
8: end for
9: Online:
10: for each page in each divergent object do
11:     if in NVM with a $RWratio < threshold$ then
12:         migrate to DRAM
13:     elseif in DRAM with a $RWratio \geq threshold$
14:         migrate to NVM
15:     end if
16: end for
```

B. Run-Time Implementation

Figure 9 shows an overview of 2PP. Our experimental framework includes three components: a profiler, a software allocator, and a memory controller. We use an offline profiler based on HMTT [6], [2] to gather PC-object information (RWratios and whether the objects are convergent or divergent), and generate an inital placement accordingly. Placement files specify each object's PC value, its preferred initial memory type, and whether it is convergent or divergent.

The 2PP software allocator loads the placement file and assigns space in the appropriate memory type for each PC-object. This involves both a custom library and the kernel. Figure 10 shows how the 2PP-aware library adds three components. First, the file parser interprets the initial placement file at launch time and generates a mapping (hash) table specifying the heap object PCs, what memory types they prefer, and whether they are convergent or divergent. Second, the object identifier traces the call stack to retrieve the PC for each allocation request and uses the retrieved PCs to index the mapping table to obtain an object's appropriate memory type. Third, the hybrid memory allocator places

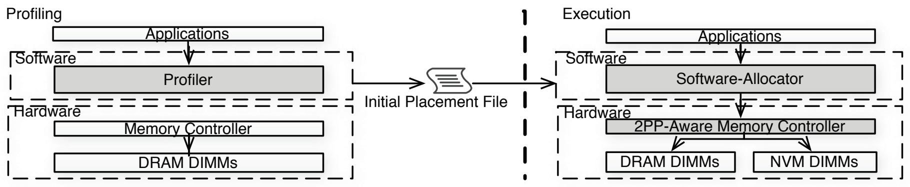


Figure 9: 2PP framework overview

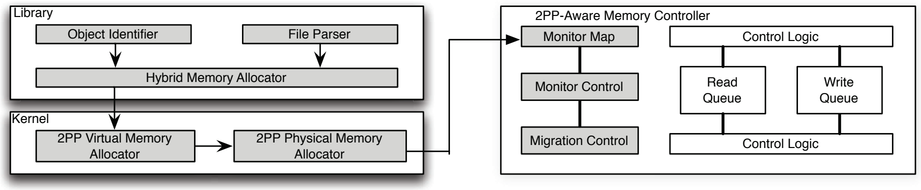


Figure 10: 2PP-aware software and hardware

each object. To implement this functionality, we add a system call (called *nvm_mmap*, similar to *mmap*) by which the OS allocates NVM memory spaces. Note that the kernel must now handle allocation and paging for the different hybrid memory spaces (both virtual and physical).

2PP is designed for exclusive architectures employing unified address spaces. The kernel adds a *VM_NVM* flag to the NVM virtual memory space, and when the 2PP runtime library requests regions via *nvm_mmap*, the kernel marks these regions as *VM_NVM*. The kernel uses the *NVM_ZONE* flag to identify NVM physical memory pages. At boot time, the kernel initializes memory zones, associating *NVM_ZONE* virtual addresses with the NVM physical address space.

The software allocator accesses the placement file to identify divergent PC-objects for which placement should be adjusted at run time. If objects are divergent, the allocator passes the information to the kernel. The library implements a system call (*divergent_alloc*) to tell the OS to mark the virtual spaces corresponding to a divergent object's allocation. The library also implements a corresponding call (*divergent_free*) to inform the OS when a divergent object's virtual space has been de-allocated. The kernel tracks divergent object allocations within the page table. The page table thus serves as the communication medium among the 2PP library, the kernel, and the memory controller.

The 2PP-aware memory controller includes three components. The monitor map marks pages belonging to divergent PC-objects. The monitor control unit tracks RWratios of divergent pages to identify those to migrate. It includes two counters to record read and write references for each divergent page (tracking up to a maximum of 25% of all pages). It periodically calculates RWratios of divergent pages to determine desired migrations. Since there are few divergent pages, counter storage overhead is low. The monitor map uses a bit for each page to mark whether it belongs to a divergent PC-object (according to the OS page table). The associated storage overhead is also low: the size of this map is 256KB for a system with 8GB of PCM and 4KB pages (8GB/4KB$\times$ 1 bit/8 = 256KB). The adjustment control unit then executes migrations by exchanging data between DRAM and NVM pages when MC is free.

IV. Evaluation

We first profile our applications to generate placement files as described above. We then run the applications on a real server machine (see Table V). **Since the applications are either parallel or multi-threaded, they use all 8 cores of the machine when running.** The server OS kernel implements the 2PP software allocator in order to emulate a hybrid memory system that uses the 2PP initial placements. We collect traces and replay them in a hybrid memory simulator to obtain both performance and energy results. We use 2GB and 8GB memory spaces for our small and large scale inputs, respectively.

Our hybrid memory simulator, which is based on DRAMSim2 [25], integrates NVM as another memory channel. For ease of comparison with prior work, we use PCM as an example NVM in our evaluations. Table IV shows features of the two memory types that we model. Since PCM is expected to have higher density than DRAM, we configure the first quarter of the memory space to be DRAM and the remaining to be PCM. The simulator thus distinguishes DRAM or NVM requests by address. **We also simulate the stall due to migration in our simulator. Once a page access comes in a mid-migration, it will be stalled until the migration finishes. We take Equation 2 into Algorithm 1 to calculate the threshold.** For the system modeled here, this threshold is 14.13. Objects with RWratios less than or equal to this value prefer DRAM, whereas the

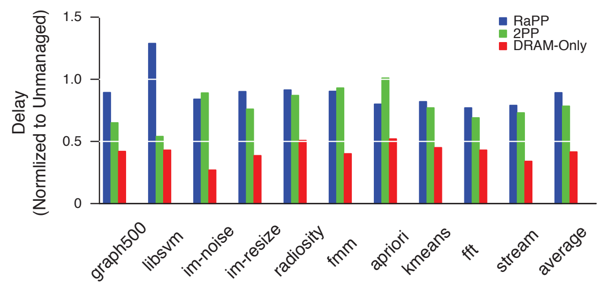


Figure 11: Delay

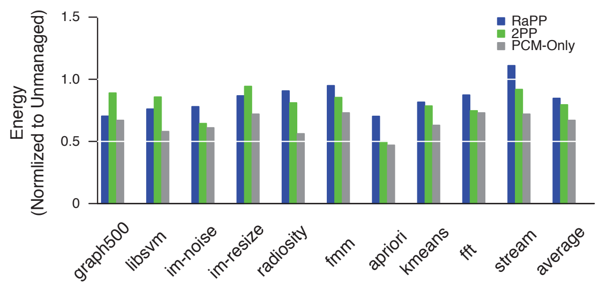


Figure 12: Energy

Table IV: Features of our hybrid memory simulator

Features		Values [35]	
		DRAM	PCM
Device	Device width	16	16
	Rank size (MB)	512	512
Latency (ns)	Burst latency	9.33	6.47
	Read latency	20.04	36.28
	Write latency	20.04	105.27
Energy (nJ) per 64B	Burst energy	12.06	3.77
	Array read energy	12.17	10.68
	Array write energy	14.48	20.75
Others	Row buffer policy	closed	closed
	Transaction queue	32	32
	Command queue	32	32

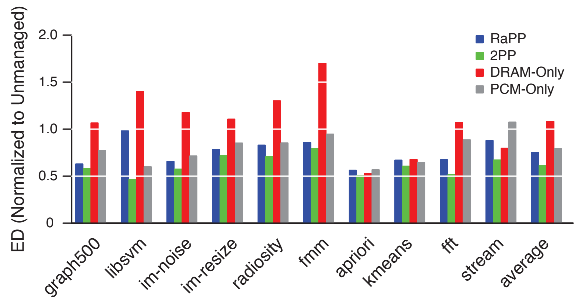


Figure 13: Energy efficiency for smaller inputs

Table V: Features of our hardware emulation platform

Features	Values
Processors	Intel Xeon E5504
CPU cores	8
L1 cache (per core)	32 KB (4-way)
L2 cache (per core)	265 KB (8-way)
L3 cache (shared)	4 MB (16-way)
Memory capacity	2 GB (8 GB)
Kernel version	linux 3.2.16

others prefer PCM.

We use the current Linux memory management as our "Unmanaged" baseline. By default, Linux allocates memory with no knowledge of the underlying memory types. For comparison, we implement RaPP [24], which uses write counts as a metric to guide dynamically migrating pages between DRAM and PCM without specifying an initial placement.

For the evaluation, we focus on energy efficiency of memory system. Thus, we use request latency and energy consumption of the memory system as the metrics instead of application wall-time, which is affected by several other factors, such as CPU calculation and IO operations.

A. *Performance and Energy*

We first compare energy efficiency of Unmanaged, RaPP, and 2PP. All results are normalized to Unmanaged. Figure 11 shows average memory request latency for the 10 workloads under RaPP and 2PP. We show average request latencies in a DRAM-only system as a lower bound. Compared with Unmanaged, RaPP and 2PP reduce latency by an average of 10.8% and 21.6%, respectively. Unmanaged is unaware of the underlying heterogeneous memory types, and its blind data placement suffers from higher access latency. 2PP outperforms RaPP by 12.1%, on average, because it reduces unnecessary migrations.

Figure 12 shows average energy consumption of memory accesses for the memory system for all workloads. We show results for a PCM-only memory system as a lower bound. Compared with Unmanaged, RaPP and 2PP reduce energy consumption an average of by 15.8% and 21.8%, respectively.

We then combine latency and energy to evaluate energy efficiency. Figure 13 shows ED values for our 10 applications running the small-scale workloads. (**The ED value is calculated as the average result of total latency times total energy over all operations**) We show results for DRAM-only and PCM-only memories for comparison. Although DRAM-only memory provides low latency, it inevitably suffers from high energy consumption, decreasing its energy efficiency. For example, compared to Unmanaged, the DRAM-only memory increases ED by an average of 8%. On the other hand, placing all data in PCM reduces energy consumption, but hurts application performance, which also lowers energy efficiency. In contrast, RaPP and 2PP increase energy efficiency by an average of 24.9% and 38.7%, respectively.

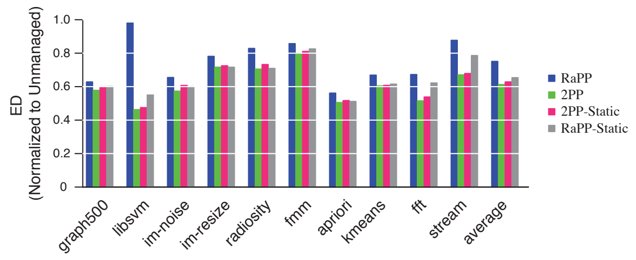


Figure 14: Energy efficiency of different mechanisms

B. Static versus Dynamic Approaches

As shown in Figure 13, 2PP outperforms RaPP by up to 51.8%, and by an average of 18.4%. This is because 2PP uses both static placement and dynamic migration. We next analyze the effectiveness of combining static placement and dynamic migration. 2PP first statically places memory objects according to their profiling results. In contrast, RaPP relies on the OS to place memory objects when they are allocated. Figure 14 shows ED results for static placement alone compared to when it is combined with dynamic migration. By placing objects according to global read-write behaviors (and avoiding all migration), 2PP-Static delivers 16.3% better energy efficiency than RaPP.

On the other hand, 2PP also employs dynamic migration for divergent objects, but it triggers data migration based on RWratio threshold instead of write counts. The RWratio threshold is calculated once per hybrid memory configuration, whereas a write-count base threshold must be tuned to the workload (necessitating retuning when workloads change). To evaluate 2PP's dynamic migration, we implement RaPP with 2PP's static placement (which we call RaPP-Static). Figure 14 shows that 2PP (with both initial placement and dynamic migration) outperforms RaPP-Static by 6.3%, on average. For *im-resize*, *radiosity*, and *apriori*, RaPP-Static delivers similar energy efficiencies to 2PP, but for *libsvm*, *fft*, and *stream*, RaPP-Static performs worse. This suggests that RWratios are a better metric on which to base migration decisions.

We also measure the number of migrations. Figure 15 shows migration reductions in 2PP compared to RaPP. 2PP only monitors pages in divergent PC-objects and migrates pages when write intensities change. Thus, 2PP triggers fewer migrations, reducing them by 79% on average.

C. Different Inputs and Scales

We next evaluate the adaptivity of 2PP under different data scales and inputs. We run our 10 workloads with the large problem sizes, as shown in Table I, using the placements generated under small scales. Figure 17 shows results. As in Figure 13, 2PP exhibit better energy efficiencies than RaPP and reduce ED by 20.9%, on average. (which are close to the reductions at small scales).

We then change data inputs for the applications with different input sets and apply the placement policies generated under the initial inputs. Table III shows application inputs. Note that working set sizes remain unchanged. Figure 18 shows ED results. 2PP still achieve better energy efficiency than RaPP. For example, for *im-resize*, RaPP reduces ED by 19.2% less than it did for the original inputs. In contrast, 2PP further reduces ED by 21.8% more than it did with the old inputs. We observe similar results for *im-resize* and *libsvm*. For *kmeans*, although RaPP and 2PP achieve fewer ED reductions under the new inputs, 2PP continue to outperform RaPP. This suggests that placement policies generated offline can achieve energy efficiency when application data scales or data inputs change.

D. Runtime Overhead

2PP introduces extra memory accesses when an application allocates and frees objects because it reads the mapping table on each allocation. On each free operation, 2PP informs the memory controller to reset the corresponding page bits in the monitor map. Figure 16 shows these additional accesses as a percentage of total memory accesses. For *im-noise*, 2PP incurs the largest overhead, introducing 7.1% more accesses compared to running without 2PP. On average, 2PP causes 3.7% more accesses. **We consider both memory allocate and free operations are not on the critical path. For example, when manipulating a data structure such as linked list and B+ tree, inserting/deleting nodes result in memory allocation/deallocation. However, a number of reads/updates usually follow the insertion of a node, which contributes lots of memory accesses. Thus, we consider these runtime overheads to be reasonable.**

V. Related Work

Some efforts have tried to characterize application characteristics for NVM-based memory. Bivens et al. [3] illustrate the range of bandwidth, latency, endurance, and reliability needed for NVM to be used in main memories. Li et al. [17] develop a binary instrumentation tool to study the potential impact of NVM for HPC applications. The tool monitors access patterns of objects in stack, heap, and global data. Their results reveal many opportunities for using NVM in HPC workloads.

Several recent research efforts study integrating NVMs with DRAM. We classify these systems into two categories according to how they exploit non-volatility: extended memories and fast persistent memories.

A. Hybrid Memory Systems

Including NVMs in main memory is a good way to build large-capacity, low-power memory systems. Since NVMs have higher write latencies than DRAM, hybrid memory systems including both DRAM and NVM are a sensible

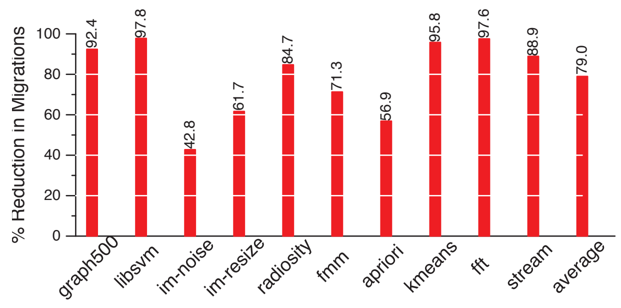


Figure 15: 2PP migrations compared to RaPP

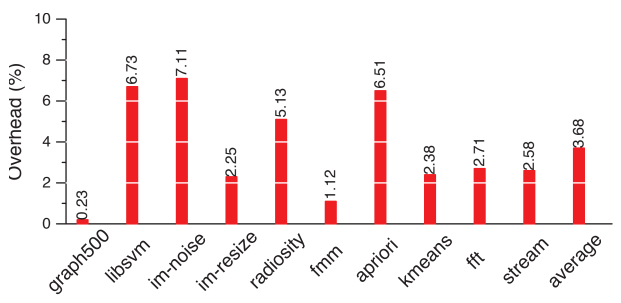


Figure 16: 2PP runtime overhead

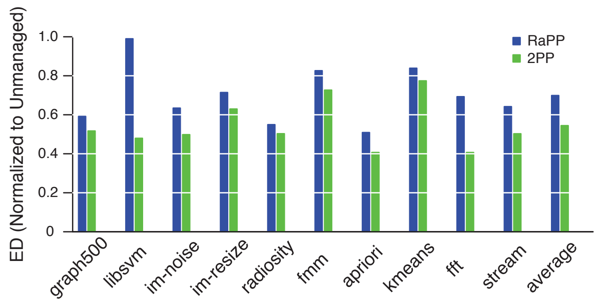


Figure 17: Energy efficiency under large scales

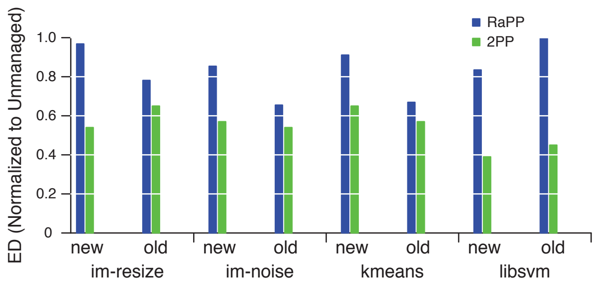


Figure 18: Energy efficiency under different inputs

choice. There are mainly two types of hybrid memory systems, *inclusive* organization and *exclusive* organization

Inclusive organizations use DRAM as a cache. Dube et al. [11] study relationships between cache performance and memory system configuration for inclusive systems. Lee et al. [14] propose a Threshold-Based Pre-Invalidation (TBPI) cache technique to increase memory bus utilization and mask PCM's poor write performance. TBPI uses an eager write-back scheme to minimize stall time on misses to their write-only DRAM cache. Zhou et al. [35] avoid redundant writes to significantly reduce write energy and extend PCM lifetime. Qureshi et al. [22] design a simple and effective wear-leveling mechanism that randomizes the address space to move lines to spatially proximal locations.

Exclusive organizations use both DRAM and NVM as main memory, which better utilizes the capacities of the two technologies. Exclusive-memory proposals generally focus on finding a good placement policy, as we do here. Choi et al. [7] implement an evaluation framework, OPAMP, to gather memory traces, DRAM/PCM specifications, and memory system configurations, which they then use to evaluate hybrid memory performance under two page allocations. Like us, they find that better performance can be achieved by profiling to generate an initial placement instead of dynamically migrating pages.

Most hybrid memory proposals dynamically migrate data into appropriate memory types. Ramos et al. [24] use Multi-Queues [36] to identify hot PCM pages to migrate to DRAM. Similarly, Zhang et al. [34] use Multi-Queues to rank pages, leveraging the OS to dynamically migrate physical pages. Yoon et al. [33] place data according to row buffer locality. Data causing frequent row buffer misses are placed in DRAM to obtain faster access. Other data are placed in PCM to save energy.

A few studies also focus on data placement in other heterogeneous memory hierarchies. For example, Dong et al. [10] focus on memory systems consisting of two partitions built from on-package, low-latency devices and high-expandability off-package devices. They use an LRU algorithm to migrate data between the partitions. Sudan et al. [28] focus on a novel, tiered memory hierarchy composed of a hot and a cold tier (both DRAM), migrating frequently accessed data to the hot tier.

B. Persistent Memory Systems

NVM can replace slow disk devices to build a fast persistent memory. Coburn et al. [8] and Volos et al. [29] provide memory programming interfaces to allow applications to store their persistent data. Both propose transactional memory systems: that of Coburn et al [8] is object-based (for user-defined objects), whereas that of Volos et al. [29] is word-based (and thus it is possible to transactionally update any granularity of data kept in persistent memory). Yang et al. [31] propose an NV-Tree, a consistent and cache-optimized B+Tree variant that not only guarantees persistency but reduces the cost of maintaining consistency.

VI. Conclusions

Hybrid memories include DRAM and NVMs to extend main memory capacity for today's large-memory applications. However, the longer write latencies and higher write energies in NVMs require careful data placement to guarantee energy efficiency. Current approaches use migration to dynamically adjust data locations according to runtime access behaviors of all physical pages. In this paper, we take another perspective to address the data placement problem. We exploit global access characteristics of heap objects to guide initial data placement. We show that for many applications, heap objects allocated at the same place in the code share similar access behaviors. We further categorize these PC-objects into convergent objects whose pages share a dominant read-write behavior and divergent objects whose pages are accessed differently. For the former, the dominant access behaviors can guide initial object placement. For the latter, dynamic placement can adapt to runtime access behaviors for individual pages. We implement 2PP, a software/hardware cooperative framework to implement both strategies, and evaluate it against a state-of-the-art migratory system that monitors all pages.We show that 2PP improves performance by an average of 12.1%. Furthermore, 2PP improves energy efficiency by up to 51.8%, and by an average of 18.4%.

VII. Acknowledgements

This work is supported by National High-Technology Research and Development Program of China under grant No. SS2015AA010502, National Basic Research Program of China under grant No. 2011CB302502, National Science Foundation of China under No. 61379042 and No. 61221062, Strategic Priority Research Program of the Chinese Academy of Sciences under grant No. XDA06010401 and Huawei Research Program No.YBCB2011030. We would like to thank the anonymous reviews for their detailed and helpful comments.

References

[1] The Graph 500 list. http://www.graph500.org/, August 2014.

[2] Yungang Bao, Mingyu Chen, Yuan Ruan, Li Liu, Jianping Fan, Qingbo Yuan, Bo Song, and Jianwei Xu. HMTT: a platform independent full-system memory trace monitoring system. In *Proc. SIGMETRICS*, pages 229–240, June 2008.

[3] A. Bivens, P. Dube, M. Franceschini, J. Karidis, L. Lastras, and M. Tsao. Architectural design for next generation heterogeneous memory systems. In *IEEE International Memory Workshop*, pages 1–4, May 2010.

[4] Santiago Bock, Bruce R. Childers, Rami Melhem, and Daniel Mossé. Concurrent page migration for mobile systems with OS-managed hybrid memory. In *Proc. Computing Frontiers*, pages 31:1–31:10, May 2014.

[5] Chih-Chung Chang and Chih-Jen Lin. LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*, 2:1–27, 2011.

[6] Licheng Chen, Zehan Cui, Yungang Bao, Mingyu Chen, Yongbing Huang, and Guangming Tan. A lightweight hybrid hardware/software approach for object-relative memory profiling. In *Proc. International Symposium Performance Analysis of Systems and Software*, pages 46–57, April 2012.

[7] Jong Hun Choi, SeongMin Kim, Chulmin Kim, Ki Woomg Park, and Kyu Ho Park. OPAMP: Evaluation framework for optimal page allocation of hybrid main memory architecture. In *Proc. Internal Conference on Parallel and Distributed Systems*, pages 620–627, December 2012.

[8] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. NV-Heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. In *Proc. Architectural Support for Programming Languages and Operating Systems*, pages 105–118, March, 2011. ACM.

[9] B Dieny, R Sousa, S Bandiera, M Souza, S Auffret, B Rodmacq, JP Nozieres, J Herault, E Gapihan, IL Prejbeanu, C. Ducruet, C. Portemont, K. Mackay, and B Cambou. Extended scalability and functionalities of MRAM based on thermally assisted writing. In *Proc. Electron Devices Meeting*, pages 1–3, December 2011.

[10] Xiangyu Dong, Yuan Xie, Naveen Muralimanohar, and Norman P. Jouppi. Simple but effective heterogeneous main memory with on-chip memory controller support. In *Proc. International Conference on High Performance Computing, Networking, Storage and Analysis (Supercomputing)*, pages 1–11, November 2010.

[11] P. Dube, M. Tsao, D. Poff, Li Zhang, and A. Bivens. Program behavior characterization in large memory systems. In *Proc. International Symposium Performance Analysis of Systems and Software*, pages 113–114, March 2010.

[12] A. Kawahara, R. Azuma, Y. Ikeda, K. Kawai, Y. Katoh, K. Tanabe, T. Nakamura, Y. Sumimoto, N. Yamada, N. Nakai, S. Sakamoto, Y. Hayakawa, K. Tsuji, S. Yoneda, A. Himeno, K. Origasa, K. Shimakawa, T. Takagi, T. Mikawa, and K. Aono. An 8Mb multi-layered cross-point ReRAM macro with 443MB/s write throughput. In *Proc. Solid-State Circuits Conference Digest of Technical Papers*, pages 432–434, February 2012.

[13] Dongsoo Lee and K. Roy. Energy-delay optimization of the STT MRAM write operation under process variations. *IEEE Transactions on Nano-technology*, 13(4):714–723, July 2014.

[14] Hyung Gyu Lee, Seungcheol Baek, C. Nicopoulos, and Jongman Kim. An energy-and performance-aware DRAM cache architecture for hybrid DRAM/PCM main memory systems. In *Proc. IEEE International Computer Design Conference*, pages 381–387, February 2011.

[15] Soyoon Lee, Hyokyung Bahn, and S. H. Noh. Characterizing memory write references for efficient management of hybrid PCM and DRAM memory. In *Proc. Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pages 168–175, July 2011.

[16] C. Lefurgy, K. Rajamani, F. Rawson, W. Felter, M. Kistler, and T.W. Keller. Energy management for commercial servers. *IEEE Transactions on Computers*, 36(12):39–48, December 2003.

[17] Dong Li, Jeffrey S. Vetter, Gabriel Marin, Collin McCurdy, Cristian Cira, Zhuo Liu, and Weikuan Yu. Identifying opportunities for byte-addressable non-volatile memory in extreme-scale scientific applications. In *Proc. International Parallel and Distributed Processing Symposium*, pages 945–956, May 2012.

[18] Jianhua Li, C.J. Xue, and Yinlong Xu. STT-RAM based energy-efficiency hybrid cache for CMPs. In *Proc. VLSI and System-on-Chip*, pages 31–36, October 2011.

[19] Piotr R Luszczek, David H Bailey, Jack J Dongarra, Jeremy Kepner, Robert F Lucas, Rolf Rabenseifner, and Daisuke Takahashi. The HPC Challenge (HPCC) benchmark suite. In *Proc. Conference on Supercomputing*. ACM, October 2006.

[20] R. Narayanan, B. Ozisikyilmaz, J. Zambreno, G. Memik, and A Choudhary. MineBench: A benchmark suite for data mining workloads. In *Proc. IEEE International Symposium on Workload Characterization*, pages 182–188, October 2006.

[21] T. Nirschl, J. B. Phipp, T. D. Happ, G. W. Burr, B. Rajendran, M.-H. Lee, A. Schrott, M. Yang, M. Breitwisch, C.-F. Chen, E. Joseph, M. Lamorey, R. Cheek, S.-H. Chen, S. Zaidi, S. Raoux, Y. C. Chen, Y. Zhu, R. Bergmann, H.-L. Lung, and C. Lam. Write strategies for 2 and 4-bit multi-level phase-change memory. In *Proc. Electron Devices Meeting*, pages 461–464, December 2007.

[22] Moinuddin K. Qureshi, John Karidis, Michele Franceschini, Vijayalakshmi Srinivasan, Luis Lastras, and Bulent Abali. Enhancing lifetime and security of PCM-based main memory with start-gap wear leveling. In *Proc. International Symposium on Microarchitecture*, pages 14–23, December 2009.

[23] Moinuddin K. Qureshi, Vijayalakshmi Srinivasan, and Jude A. Rivers. Scalable high performance main memory system using phase-change memory technology. In *Proc. International Symposium on Computer Architecture*, June 2009.

[24] Luiz E. Ramos, Eugene Gorbatov, and Ricardo Bianchini. Page placement in hybrid memory systems. In *Proc. International Conference on Supercomputing*, pages 85–95, May 2011.

[25] P. Rosenfeld, E. Cooper-Balis, and B. Jacob. DRAMSim2: A cycle accurate memory system simulator. *IEEE Computer Architecture Letters*, 10(1):16–19, 2011.

[26] C.W. Smullen, V. Mohan, A Nigam, S. Gurumurthi, and M.R. Stan. Relaxing non-volatility for fast and energy-efficient STT-RAM caches. In *Proc. High Performance Computer Architecture*, pages 50–61, February 2011.

[27] M. Still. *The Definitive Guide to ImageMagick*. Apress, 2005.

[28] K. Sudan, K. Rajamani, Wei Huang, and J.B. Carter. Tiered memory: An iso-power memory architecture to address the memory power wall. *IEEE Transactions on Computers*, 61(12):1697–1710, December 2012.

[29] Haris Volos, Andres Jaan Tack, and Michael M. Swift. Mnemosyne lightweight persistent memory. In *Proc. Architectural Support for Programming Languages and Operating Systems*, pages 91–104. ACM, March 2011.

[30] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 programs: characterization and methodological considerations. In *Proc. International Symposium on Computer Architecture*, pages 24–36. ACM, June 1995.

[31] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. NV-Tree: Reducing consistency cost for nvm-based single level systems. In *Proc. File and Storage Technologies*, pages 167–181, February 2015.

[32] Dong Ye, A. Pavuluri, C.A. Waldspurger, B. Tsang, B. Rychlik, and S. Woo. Prototyping a hybrid main memory using a virtual machine monitor. In *Proc. International Conference on Computer Design*, pages 272–279, October 2008.

[33] Hanbin Yoon, Justin Meza, Rachata Ausavarungnirun, Rachael Harding, and Onur Mutlu. Row buffer locality-aware data placementin hybrid memories. Technical Report No. 2011-005, Dept. of Electrical and Computer Engineering, Carnegie Mellon University, September 2011.

[34] Wangyuan Zhang and Tao Li. Exploring phase change memory and 3D die-stacking for power/thermal friendly, fast and durable memory architectures. In *Proc. Parallel Architectures and Compilation Techniques*, pages 101–112, September 2009.

[35] Ping Zhou, Bo Zhao, Jun Yang, and Youtao Zhang. A durable and energy efficient main memory using phase change memory technology. In *Proc. International Symposium on Computer Architecture*, pages 14–23, June 2009.

[36] Yuanyuan Zhou, James Philbin, and Kai Li. The Multi-Queue replacement algorithm for second level buffer caches. In *Proc. USENIX Annual Technical Conference*, pages 91–104, June 2001.