

PRISM: Zooming in Persistent RAM Storage Behavior

Ju-Young Jung and Sangyeun Cho
Computer Science Department
University of Pittsburgh
{juyoung, cho}@cs.pitt.edu

Abstract—It has been foreseen that some of the roles assumed by conventional rotating hard disk drives (HDDs) will migrate to solid-state drives (SSDs) and emerging persistent RAM storages. New persistent RAM storages have critical advantages over HDDs and even SSDs in terms of performance and power. Persistent RAM technologies are flexible enough to be used for both storage and main memory—in future platforms, this flexibility will allow tighter integration of a system’s memory and storage hierarchy. On the other hand, designers are faced with new technical issues to address to fully exploit the benefits of persistent RAM technologies and hide their downsides.

In this paper, we introduce PRISM (PeRsistent RAM Storage Monitor)—our novel infrastructure that enables exploring various design trade-offs of persistent RAM storage. PRISM allows designers to examine a persistent RAM storage’s low-level behavior and evaluate its various architectural organizations while running realistic workloads, as well as storage activities of a contemporary off-the-shelf OS. PRISM builds on kernel source code level instrumentation and the standard Linux device driver mechanism to generate persistent RAM storage traces. Moreover, PRISM includes a storage architecture simulator to faithfully model major persistent RAM storage hardware components. To illustrate how and with what PRISM can help the user, we present a case study that involves running an OLTP (on-line transaction processing) workload. PRISM successfully provides the detailed performance analysis results, while incurring acceptable overheads. Based on our experience, we believe that PRISM is a versatile tool for exploring persistent RAM storage design choices ranging from OS-level resource management policy down to chip-level storage organization.

I. INTRODUCTION

HDDs have been the choice secondary storage for modern computers since they appeared in mid-50s. Unfortunately, they present one of the long-standing system performance bottlenecks due to slow access latency and power burning mechanical operations. More seriously, as the DRAM technology continues to improve its speed and density every year, the performance gap between main memory and secondary storage becomes larger and larger. Fully addressing this problem remains a daunting challenge; researchers have proposed various techniques at multiple design layers including: smart I/O buffering with fast RAM, host buffer caching and OS-level disk scheduling. These optimizations successfully handle only portions of the problem as a first-aid patch to the fundamental limitations of the rotating HDD.

Recently, the storage research community is paying much attention to solid-state memory technologies such as flash memory, PCM (phase-change memory), STT-MRAM (spin-transfer-torque magneto-resistive RAM) and FeRAM (ferro-

electric RAM) as a fast storage medium [1]–[6]. Especially, solid-state drives (SSDs), typically built with NAND flash memory, have been slowly yet increasingly penetrating the market as HDD replacement or complement, and are currently mass-produced [7]–[9]. However, undesirable artifacts of flash memory result in design complexities of SSDs and limit performance. For example, flash memory has a coarse-grain access unit, requires costly erase operation before writing, does not support in-place update, and is subject to write endurance limitations. In contrast, emerging persistent RAMs like PCM and STT-MRAM do not come with many of these restrictions (see Section II for more on this). Researchers are hence actively exploring the potential of using persistent RAM in storage systems and how such storage systems will change the overall system architecture [1], [10], [11].

Modern storage systems are complex, comprised of multiple layers of software and hardware components that interplay. They include the file system, OS block I/O stack, device driver (software components), bus, chips, and devices like HDD and SSD (hardware components). These diverse components are tightly bound up with each other and affect the overall storage system performance. Therefore, it is important to maintain a holistic view throughout the storage system design process and avoid focusing on only a single aspect of the storage system. With adequate measurement and system performance analysis tools, one can design storage software to fully mask/exploit the underlying recording medium’s physical characteristics such as slow seek time and ease of bulk transfer (in the case of HDDs). Likewise, the hardware components can also be customized to utilize the software components’ characteristics such as the I/O scheduling policy and the file system.

Ideally, a storage researcher would run real workloads on a persistent RAM storage model and measure how the workloads exercise the underlying persistent RAM devices to fully understand their interactions. Yet, to the best of our knowledge, there exists no publicly available tool to (1) allow a storage researcher to run large realistic workloads on a persistent RAM storage model, (2) expose low-level persistent RAM storage behavior (without assuming the traditional block I/O interface), and (3) evaluate a variety of design choices in a single integrated framework. The contribution of our work is to design and implement PRISM, a novel infrastructure to explore various design trade-offs of a persistent RAM storage in terms of performance, reliability, and energy. PRISM enables the user to look deeply into persistent RAM storage behavior under real workloads including contemporary off-the-shelf OS

TABLE I
COMPARISON OF PERSISTENT RAM TECHNOLOGIES [4]. * F IS THE FEATURE SIZE OF A GIVEN PROCESS TECHNOLOGY.

	Latency			Program Energy	Allowable Access unit	Endurance	Density*
	read	write	erase				
NAND flash	25us	200us	1.5ms	10 nJ	page/block	$10^4 \sim 10^6$	$4 \sim 5F^2$
PCM	20ns	100ns	N/A	100 pJ	byte	$10^8 \sim 10^9$	$5F^2$
STT-MRAM	10ns	10ns	N/A	0.02 pJ	byte	10^{15}	$4F^2$
FeRAM	75ns	50ns	N/A	2 pJ	byte	10^{13}	$6F^2$

activities. We have implemented PRISM in the Linux 2.6.24 kernel.

In the remainder of this paper, we will first describe the trend of persistent RAM technologies to further motivate our work in Section II. In Section III, we will describe PRISM in detail and its current implementation status. We will then present a case study with experimental results collected with PRISM to highlight its capabilities in Section IV. Lastly, after discussing related work briefly in Section V, Section VI will summarize our current and future work.

II. BACKGROUND

A. Emerging Persistent RAMs

SSDs, primarily built with NAND flash memory today, offer faster access latency, lower energy consumption and stronger shock resistance than HDDs. Various uses of SSDs have been proposed to bridge the performance gap between main memory and secondary storage [12]–[14]. Unfortunately, flash memory has properties that complicate the design of the low-level storage management software (often called flash translation layer or FTL). A flash memory cell cannot be overwritten before it is first erased (“erase-before-write” constraint). Moreover, each memory cell has limited write endurance. After reaching this write endurance limit, further writes to the cell become unreliable. Read, write and erase units are much larger than byte. Most critically, the scalability of the flash memory technology is questionable [1].

New emerging persistent RAMs, such as PCM, STT-MRAM and FeRAM, do not come with some of flash memory’s shortcomings; they are in-place updatable, byte-addressable and typically have higher write endurance. Table I captures the main characteristics of the NAND flash memory, PCM, STT-MRAM and FeRAM technologies. While a persistent RAM’s access interface is similar to that of a conventional RAM, its cell-level operation resorts to fundamentally different physical phenomena such as phase change (PCM), magnetization (STT-MRAM) and polarization (FeRAM).

Because persistent RAMs are byte-addressable, researchers have evaluated their potential as CPU addressable memory. For example, [15], [16] utilize FeRAM as fast intermediate storage to improve the file system performance. However, their work evaluated a file system under an unrealistically miniaturized environment with only 8 to 16 MB of FeRAM. Meanwhile, computer architects are actively evaluating persistent RAMs

as a direct replacement of DRAM in main memory [17]–[19]; however, they do not study persistent RAM’s storage aspect.

We expect that technological advances will continue enhancing the operational characteristics of persistent RAMs as well as dropping their cost per GB. Therefore, a large persistent RAM storage (as alternative or complement to HDD and SSD), directly accessed via the CPU’s memory interface, has the potential to become a reality. With this outlook, we feel that there is an imminent need to explore key design aspects of a persistent RAM storage—from OS-level virtual memory manager (VMM) to file system design to chip-level organization. The fact that there is no publicly available tool for us to address all these aspects strongly motivated us to develop PRISM.

B. Conventional Storage System

Traditionally, the secondary storage has maintained a block device interface regardless of the underlying recording mechanism (e.g., HDD and SSD). As a result, storage systems typically share key datapaths through multiple software layers to reach the storage medium. Fig. 1 depicts the major strata that spans an entire storage system in the case of the Linux OS. A read or write request generated by CPU will travel from top to bottom along the shown stack to handle the request, and then traverse back to the requester with either the desired data or acknowledgment on the completion of the requested operation.

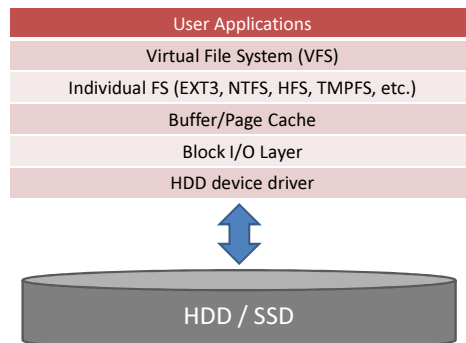


Fig. 1. Conventional storage system stack.

For further exemplification, let us consider how a read operation is handled. Once a user application issues the *read()* system call, the request is first passed to the VFS (Virtual File System) layer, a generic abstraction layer on which diverse

file systems can co-exist. Then the request is transferred to the specific file system where the desired data resides. In an ensuing process, the kernel looks up its page cache that holds in-memory file objects; if the requested data object is not found there, the read request keeps journeying down the stack. In the block I/O layer, an internal I/O request entry is assigned for this read, and then queued up until the request proceeds to the head of the queue and the device driver is ready to handle the request. The device driver will communicate with the hardware disk controller to perform necessary low-level data transfers. The disk controller will eventually notify the device driver of the completion of transfers with the requested data. During data transfer, data will be stored as in-memory file objects and the user application resumes its execution with the desired data after the kernel finishes copying the data from the kernel space to the user space.

Given the large number of layers a request and data have to go through, obviously, streamlining the storage I/O datapaths can bring sizable performance benefits. Recently, some SSDs deviate from this “conventional” stack architecture to bypass a few layers. For example, FusionIO’s ioExtreme [9] makes use of a specialized file system, a simplified OS block I/O layer, and the PCIe (PCI Express) interface. Such design changes are likely to happen for persistent RAM storage systems as well due to their drastically different physical characteristics from rotating HDDs.

C. Designing Persistent RAM Storage

Major architectural and system design changes are anticipated with the introduction of persistent RAM as the main storage medium:

- 1) **File system:** Since the storage medium is uniformly random addressable and does not mandate block-granule operation as a HDD or SSD does, a natural file system design strategy could be an *in-memory file system*. This strategy helps avoid dealing with an unnecessary and often expensive block device interface on accessing a file. To achieve the best performance, a new file system must consider physical characteristics of the target persistent RAM storage medium.
- 2) **OS software stack:** Thanks to desirable properties of a persistent RAM medium, the depth of the OS storage related software stack can be greatly reduced, resulting in a thinner, more efficient software stack and improved overall storage performance. For example, a small data update request does not need to incur block-granule operations (e.g., creating, copying, and transferring a block with DMA) because a persistent RAM can address bytes and support in-place updates. Moreover, a persistent RAM’s uniform fast access latency will help slash a number of I/O queues and complicated I/O scheduling schemes.
- 3) **Wear leveling:** Since some persistent RAMs such as PCM are subject to limited write cycles—although the expected limit is much larger than that of NAND flash

memory—a persistent RAM storage design is still expected to incorporate a wear leveling scheme. However, wear leveling can be merged into the existing OS’s page allocation/reclamation policy. As a result, such a strategy can not only obviate the need for a heavy FTL common in flash memory based storages, but also enable smart wear leveling control with system-wide knowledge of memory-storage interaction.

- 4) **Exploiting parallelism:** A HDD does not by itself expose parallelism when retrieving recorded data.¹ To offset the cost of mechanical head-arm movement, disk scheduling algorithms consider multiple disk access requests simultaneously and cluster them based on their proximity on the platter. On the other hand, with persistent RAM, data access parallelism is revealed by introducing multiple channels to chips/modules and banks (aka planes) within chips. Hence, data mapping and interleaving must be carefully coordinated in the hardware design as well as the OS-level management policy.

All these architectural and system design considerations can dramatically affect the performance of a persistent RAM storage—its access latency, reliability, energy consumption, and storage utilization. Now, the following section will describe PRISM, a novel infrastructure that enables a persistent RAM storage designer to explore these considerations in an integrated manner.

III. PRISM

A. Overall Architecture

To consider all key design aspects together, we take a holistic emulation approach in PRISM design, involving the OS, file system, and storage hardware architecture. The primary design goals of PRISM are: (1) realistic but fast storage system monitoring, (2) ease of functional extension by modular design, (3) flexible user configurability, and (4) automated data analysis. To achieve the goals, PRISM is structured into two major components as Fig. 2 depicts: the front-end tracer that tracks and records various storage activities (Fig. 2(a)) and the back-end (off-line) simulator called *PSDSim* (Persistent RAM Storage Device Simulator) that post-processes a trace based on the user-defined storage configuration (Fig. 2(b)).

Conceptually, a user would trace OS-level file I/O activities with the PRISM probing facility that is deeply embedded inside the kernel. The collected traces are then fed to *PSDSim* configured with the user input. Lastly, the user extracts the desired information (e.g., wearing of cells) and evaluate a particular aspect of the storage design (e.g., wear leveling algorithm) under consideration. Note that the two components can be either coupled together to work on-line or decoupled as needed; we describe them separately in the next subsection to contrast their roles.

¹At the system level, parallel data access is enabled by involving multiple HDDs when data are interleaved among them, e.g., RAID-5.

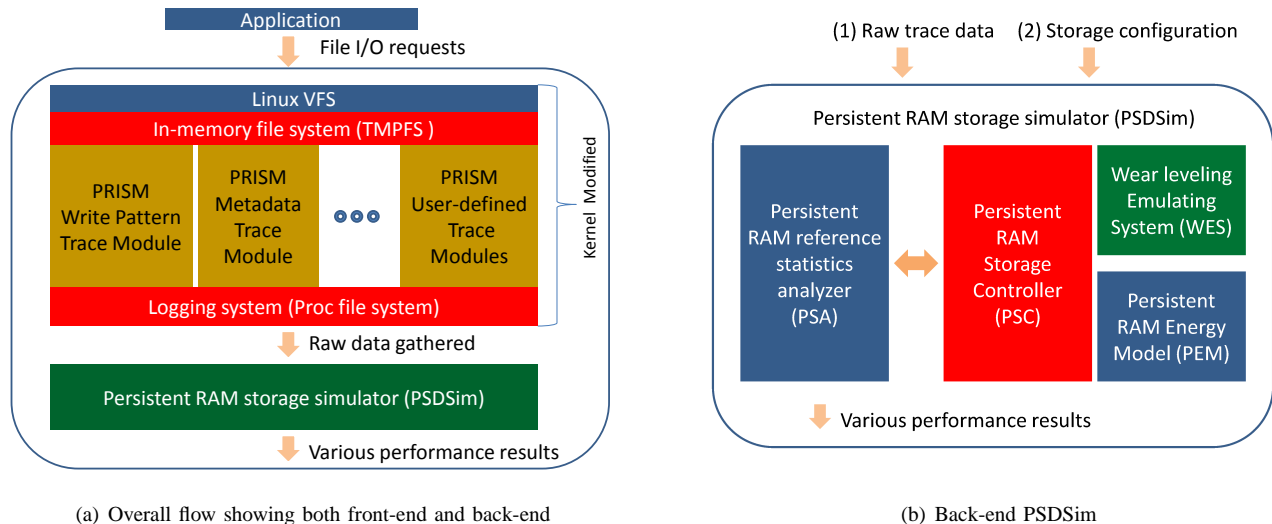


Fig. 2. PRISM architecture.

B. PRISM Components

Front-end tracer. In this work, we have modified Linux 2.6.24, especially the `tmpfs` file system [20] and the generic file system layers to trace file I/O activities. `tmpfs` is a main memory based file system and was originally invented for performance enhancement of short-lived file accesses by avoiding costly disk I/O. Because its main focus is to handle temporary file objects efficiently, it does not assume a persistent medium to store file data.

While its original design objective is different from ours, `tmpfs` offers an attractive framework for use in PRISM for several reasons. First of all, `tmpfs` is an in-memory file system that has already been built in the mainline Linux kernel. Second, it serves our design objectives well in that it “emulates” byte-addressable file data update and is tightly coupled with the kernel’s resource management policies such as page allocation and reclamation. Moreover, `tmpfs` can resize its allocated memory resource dynamically within the capacity set by the user (unlike other popular RAM drives like `ramdisk`). `tmpfs` can even increase its size beyond the system’s main memory capacity, to as much as the main memory plus the system swap space. Additionally, `tmpfs` enables us to observe how well a persistent RAM storage cooperates with the existing virtual memory policies (much tuned to work with a HDD), even before we design a specialized file system for a persistent RAM storage. This will help file system designers by identifying what they have to change and support.

Various “pluggable” tracers within the PRISM front-end can trace specific storage access behavior. Note that a PRISM tracer tracks file I/O activities on the byte-addressable storage model instead of the block storage device model like HDD or SSD. For example, our “write pattern tracer” can gather information on file data being written, such as virtual and physical addresses accessed, write size, and actual write data. Similarly, a “read pattern tracer” can capture information such as accessed addresses and requested data size. A “metadata

tracer” can keep track of the changes in file metadata such as updates in the `inode` structure.

PRISM provides a convenient method with which a user can specify his or her own tracer and how it connects to other kernel modules. Much like a typical Linux device driver, each PRISM tracer is implemented as a kernel module utilizing the `proc` file system interface for interacting with the kernel and logging. Whenever a specific activity happens in the kernel, the call-back function of the corresponding tracer will be invoked to perform a designated tracing task. By modularizing tracers, only necessary tracers will be loaded into the kernel so that PRISM can keep kernel operations lightweight during tracing.

Back-end simulator (PSDSim). Raw file I/O traces that we collect with the PRISM front-end are fed to the back-end architecture simulator. Hardware and device driver designers often ask if a particular storage organization improves the overall system performance, power, and lifespan. PSDSim facilitates exploring persistent RAM storage design trade-offs by allowing the user to define a persistent RAM chip configuration, an interconnection topology, a wear leveling algorithm, and an energy consumption model.

Fig. 2(b) shows the hardware specific back-end modules: Persistent RAM storage controller (PSC), wear leveling emulating system (WES) and persistent RAM storage energy model (PEM). PSC simulates bank conflicts and implements conflict resolution policies (e.g., round-robin for service fairness). It works with WES to lessen biased writes to specific persistent RAM banks with no direct intervention with the OS VMM. PEM enables us to estimate energy consumption of storage access activities.

Finally, to aid users with fast and intuitive data analysis, PSDSim includes a statistics analysis module (PSA). PSA parses and formats a set of traces according to a user specification. It also automates the process of visualization with simple scripting support.

For higher flexibility of architectural exploration, PSDSim

takes the following design requirements into account:

- **Persistent RAM chip configuration:** It has not been fully understood what is the best internal organization for a persistent RAM chip for storage systems. For example, how many banks per chip will be good? What are adequate read and write bandwidth sustained by each bank? How many entries do we need in an on-chip write queue? To answer these questions, the user must have ability to configure the internal hardware organization of a persistent RAM chip.
- **Storage device configuration:** We also want to be able to configure a storage device—a physically or logically separable hardware unit that hosts the storage capacity seen by the user (e.g., persistent RAMs mounted on a DIMM (dual in-line memory module)). The most important parameter here is the total storage capacity and how the capacity is split into different (chip or DIMM) packages. Furthermore, it is desirable to model a storage system having multiple storage devices, e.g., organized in a RAID-like arrangement to exploit device-level parallelism and enable failure recovery. Therefore, PSDSim need the ability to model and evaluate the performance of multiple storage devices connected in diverse interconnect topologies.
- **Management granularity:** Although the persistent RAM storage envisioned in this work is byte-addressable, the storage may utilize a larger unit than a single byte to efficiently implement certain management policies such as wear leveling and garbage collection. To this end, PSDSim should support configuration of “page size,” which is the minimum unit with which the system management software keeps track of wearing information and reclaims resource.
- **Wear leveling strategy:** Since some persistent RAMs have limited write cycles, PSDSim must provide a framework to compare different wear leveling strategies. Although there have been many wear leveling strategies developed for SSDs and some of them may be relatively easily adapted to the the persistent RAM storage, PSDSim should provide high flexibility for the user to explore new wear leveling ideas specifically targeted to the persistent RAM storage.
- **Energy model:** To provide fast and accurate energy estimation, PSDSim should include a configurable energy model to compute energy consumption of a given persistent RAM storage. The energy model should be sufficiently detailed and should separately report energy due to chip-level structures like address decoder, memory cell array and sense amplifiers, as well as more system-level structures like interconnects and controllers.

C. PRISM Developmental Stage

As of this writing our PRISM front-end tracers are fully functional and are capable of tracing both data and metadata accesses into a named `proc` file. They can monitor storage activities within the OS kernel at the system software level

as well as at the file system level. Although our current implementation leaves room for further improvement in terms of tracing overheads, our experiment shows that the overheads are very moderate (Section IV-C). Trace modules are small in object size. The heaviest tracer module is about 134 KB.

For the PRISM back-end simulator, we have implemented and validated the most important functionalities of a storage device. PSDSim can currently take and parse raw traces passed from the front-end tracers and run architectural simulations according to a user-defined storage configuration. In our current implementation, the timing-accurate PSC can handle read and write requests to the modeled persistent RAM arrays and the WES can realize a perfect wear leveling scheme based on a per-page write counting mechanism similar to [34]. In addition, the PEM supports a simple energy consumption model based on the cell access energy information (e.g., Table I). Another model we have adapted in the PEM uses published device characteristics, e.g., [35]. Our near-term future work includes: Testing PSDSim with a multiple storage device configuration by additionally implementing elaborate interconnect models and supporting a storage system level energy consumption model.

IV. CASE STUDY WITH PRISM

In this section, we present a case study to illustrate what a user can do with PRISM. Before describing our experiment further, let us make two assumptions. First, we assume a persistent RAM storage managed directly by the Linux VMM without a dedicated storage management software layer such as FTL in SSD. Our storage models mimic a single address space memory and storage system [1]. Second, we are interested in observing write patterns (rather than read patterns) to the persistent RAM medium for enhancing write endurance. We choose this scenario because wear leveling issues may still remain an important problem in the future for the persistent RAM storage.

A. Experimental Setup

For experiments, we employ an on-line transaction processing benchmark (TPC-C) [21] that has a sufficiently large number of file update operations to exercise and wear individual storage cells meaningfully. TPC-C is a representative benchmark to evaluate the transaction processing performance of modern database systems. The benchmark first creates database schemas that result in nine separate tables and then populates each table having different cardinality according to the TPC-C standard requisites. Once all the tables are prepared, the benchmark starts processing queries requested by a pre-determined number of users. This simulates a typical on-line transaction processing situation that involves multiple customers (e.g., Amazon e-market). For this study, we specifically make use of an implementation by Llanos [22], which includes an open-source PostgreSQL 8.1.4 as database engine.

During measurement we run a 2-hours test with a single warehouse and 10 terminals. Also, we set the 20-minute ramp-up period before starting the actual measurements and

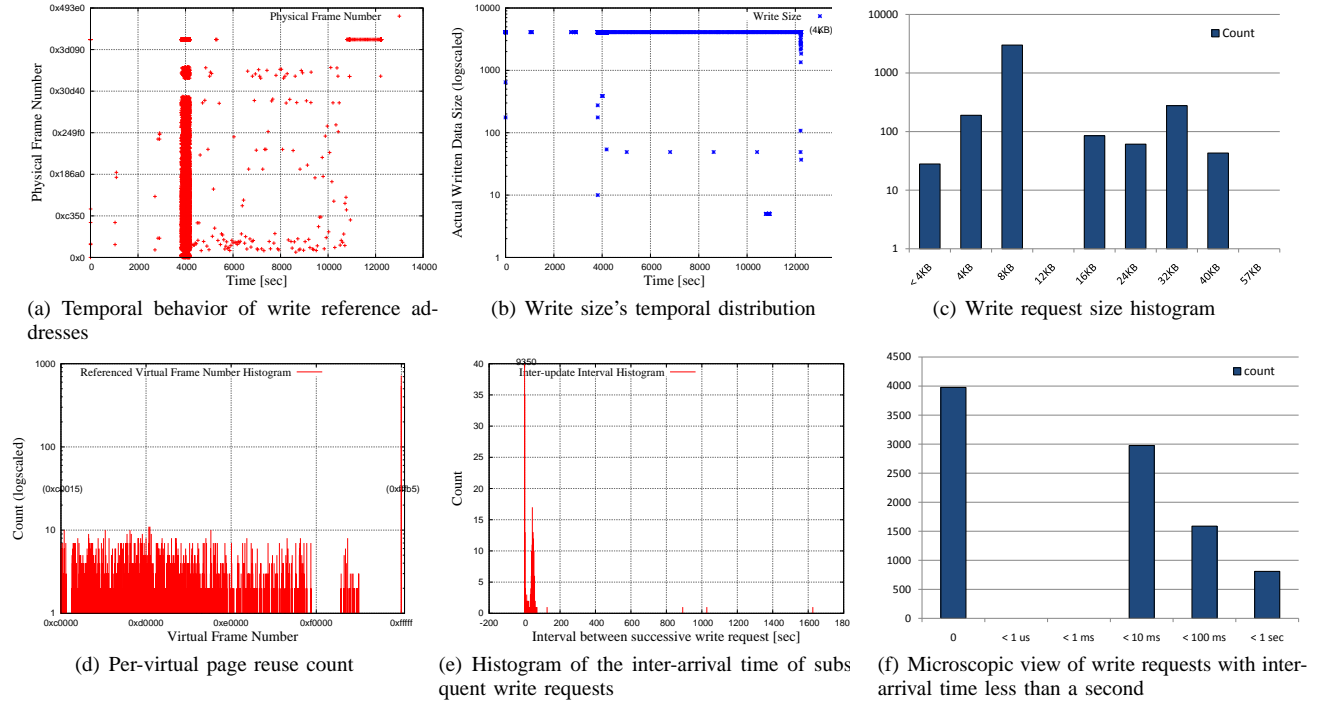


Fig. 3. TPC-C write access patterns for file data on persistent RAM storage.

do not perform hourly “vacuum” operations, which remove performance-sensitive residual information from the database. Table II summarizes our experimental environment.

TABLE II
EXPERIMENTAL PLATFORM.

Components	Specification
CPU	Intel Core Duo T2300
Core Clock Speed	1.66GHz
L2 Cache	2MB
Main Memory	1GB SDRAM
OS	Linux Kernel 2.6.24.7
File System	Tmpfs 10GB capacity
Workload	TPC-C two hour measurement

B. Main Results

File data update pattern. Fig. 3 and 4 depict the file data and metadata update patterns observed during measurement. Fig. 3(a) shows that TPC-C issues temporally dense file write operations for updating database tables at around 4,000 second. We also observe from Fig. 3(b) that a large portion of file write operations are for 4 KB size even if the most common write request size was 8 KB, as shown from Fig. 3(c). This is due to an artifact of the Linux kernel: It first cuts up large file data into 4 KB pages and then performs page-size writes. The largest write request from our TPC-C trace was 57 KB, which was completed by 14 separate 4 KB write operations (plus a smaller write).

The result also shows that there are many smaller writes than a page size, which would typically incur a full block update in a block device like HDD or SSD (see the first column in Fig. 3(c)). Note that the result so far is for file data writes rather than metadata update. As we will discuss shortly, metadata update is frequently on relatively small data structures. For example, the `inode` structure size of the popular `ext3` file system is only 128 bytes.

From Fig. 3(a) and Fig. 3(d), we find that VMM allocates a number of pages from the low-memory (following the Linux jargon) below the 893 MB line, while there are highly clustered accesses to pages allocated from the high-memory around the 1 GB line to handle high page demanding situations. Since we perform this experiment on a 32-bit machine (see Table II), the Linux kernel is able to address a maximum of 1 GB address space. Because the small number of high-memory pages are repeatedly reused on every page shortage situation, such references form a “spike” in the high-memory region while other pages have a much smaller reuse count of less than 10. Fig. 3(d) manifests the per-page reuse pattern of the TPC-C benchmark.

Fig. 3(e) and Fig. 3(f) plot the inter-arrival time between successive write requests. Notably, (near) zero inter-arrival time is dominant and timing gaps are centered around at less than 100 seconds. We found that (near) zero inter-arrival time occurs frequently because: (1) our platform has dual out-of-order processor cores and (2) the most common request size is 8 KB (greater than the 4 KB page size), thereby creating many 4 KB requests that are issued nearly simultaneously. Meanwhile, write requests having an inter-arrival time of less

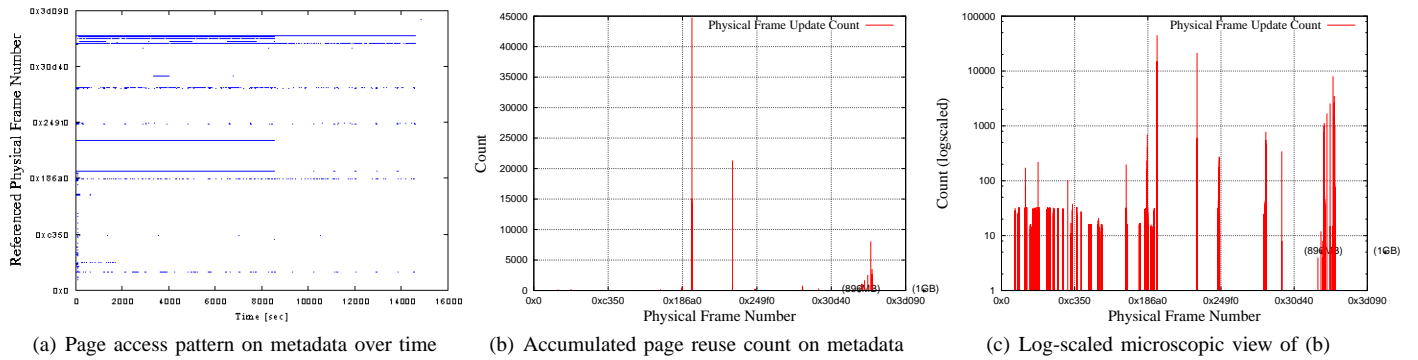


Fig. 4. TPC-C write access patterns for file metadata (inode's `i_atime`) on persistent RAM storage.

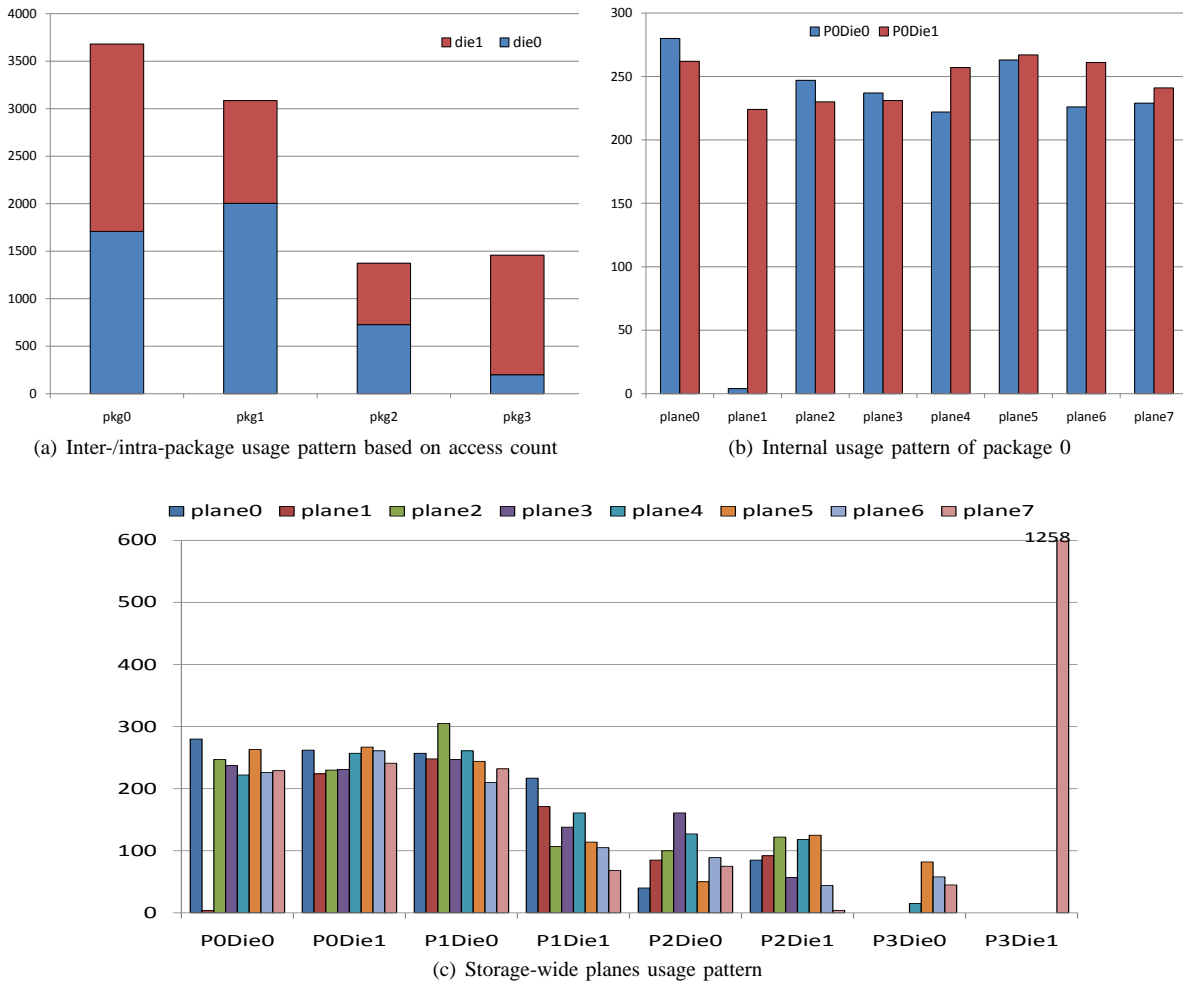


Fig. 5. Hardware architecture usage pattern on file data update.

than 1 second have a bimodal distribution with peaks near the ends of the 1 second span missing successive write requests distant of nanosecond or microsecond. Storage designers may make use of this characteristic to determine the size and depth of request buffers on a storage controller.

File metadata update pattern. Fig. 4(a)–(c) show the access pattern for metadata—inode's `i_atime` field. Metadata updates are particularly interesting to us because memory cells

storing metadata are prone to wear out faster than cells for file data. While the file data are updated with explicit file write operation, metadata are often updated with both file write and read operation in order to keep the file information up-to-date. For example, file reference count should be updated whenever a process newly opens the file to read. Otherwise, the OS may close the file prematurely when the reference count is (wrongly) zero. Since the storage space to store

metadata is typically smaller than that to hold file data, the number of pages touched for metadata updates is also small, as shown in Fig. 4(a). However, when a storage designer has to consider the wear leveling issue of storage cells, the number of pages touched by metadata is likely to increase because pages holding metadata may move around all pages within storage device. This leaves designers a question of how the migration of metadata pages should be done to achieve longer lifetime of memory cells.

Hardware resource utilization. The spikes in Fig. 4(b) and 4(c) show that there are unbalanced page updates for metadata; these pages must be handled carefully to keep them from rapidly reaching the write endurance limit. To investigate this situation in detail, we repeated experiments after organizing our persistent RAM storage in relatively small 1 GB capacity consisting of four packages. Each package holds two dies, each having 8 planes.

Fig. 5 presents the results, from which we make several interesting observations. First, there are unbalances at different levels. Fig. 5(a) shows that references are headed to different packages in different amounts: Low-numbered packages received more references. Furthermore, two dies in each package are accessed differently in frequency. For example, die 1 of package 3 is predominantly accessed, leaving its sibling die 0 far behind. Fig. 5(b) shows that there is additional unequal access pattern within a single die. Especially, plane 1 in die 0 was updated rarely while all other planes show a relatively balanced usage. Lastly, Fig. 5(c) plots the per-plane update frequency across all packages and dies. We again find similarly unbalanced usages. Quantifying all these unbalanced usage of memory cells (in different planes) helps the user understand how data must be interleaved and how wear leveling must be done given a hardware organization.

Wear leveling effect. To manifest the potential write endurance problem of a persistent RAM storage more clearly, we reorganize our hardware in this experiment and decrease the single plane capacity to 2 MB (i.e., finer-grain interleaving of data). Note that we keep the overall storage capacity unchanged.

As a result, Fig. 6(a) displays the heavily skewed uses of the available planes; lower-numbered planes are still used much more frequently than higher-numbered ones. To elude this undesirable situation, we apply a simple round-robin (RR) wear leveling policy assuming a flexible data to plane mapping capability. With RR, when a page’s update count reaches a pre-defined threshold, a “clean” page frame with a small update count is identified and the page is migrated to the selected page frame. If no clean page frame is found, WES (explained in Section III-B) globally increments the write limit by one conservatively. Algorithm 1 lists in pseudo-code the above sketched algorithm. This simple RR policy may not achieve the best wear leveling performance, however, is selected to highlight PRISM’s functionality. After applying the RR policy, significantly more balanced plane usage is achieved, as shown in Fig. 6(b).

Algorithm 1 Example write endurance management in WES

```

// initialize current_threshold and counters (ucount)
current_threshold ← init_val
for k = 1 to NUM_PAGES do
    page k :: ucount ← 0
end for
// perform write endurance check
if page i :: ucount > current_threshold then
    if there exists a page having ucount ≤ current_threshold
    then
        page j = getCleanPageID()
        page j :: ucount ++
    else
        page i :: ucount ← 1
        current_threshold += 1
    end if
else
    page i :: ucount ++
end if

```

C. PRISM Overheads

Lastly, we looked into how PRISM affects a program’s I/O performance by running the IOzone benchmark [23] with 200 MB file write operations. We do not measure overheads due to the back-end PSDSim because it can be easily offloaded in real experiments.

Our measurement showed that file writes with full-tracing PRISM have a modest slowdown of $\sim 3.5\times$ compared with unmodified `tmpfs`. While our current implementation remains room for improvement, this slowdown is acceptable for experiments involving large workloads. When compared with HDD, PRISM is an order of magnitude faster ($\sim 11.5\times$), even when sequential file write operations are performed (which is favorable to HDD).

V. RELATED WORK

There have been considerable efforts to develop tools to analyze storage behaviors and perform architectural exploration. In what follows, we will summarize most notable previously developed tools that are closely related to our work in two categories: kernel tracers and storage-architecture simulators.

Kernel tracers. One of the better-known tracers used today in storage system studies is `blktrace` [25]. It keeps track of I/O requests to a block device by recording the logical block address, type, and size of each request. While this tool provides detailed block I/O request information, it cannot trace storage activities not going through the conventional block I/O layer as in the case of a persistent RAM storage on the system’s memory bus. `Diskmon` [26] is a similar utility for the NTFS file system of the Windows OS.

Although it was not originally designed for storage system tracing, `pagemap` [27] is another tracer to note. It traces Linux’s virtual memory activities on a per-process basis, and allows a user-space program to post-process the statistics of in-kernel memory usages. However, its tracing capability is

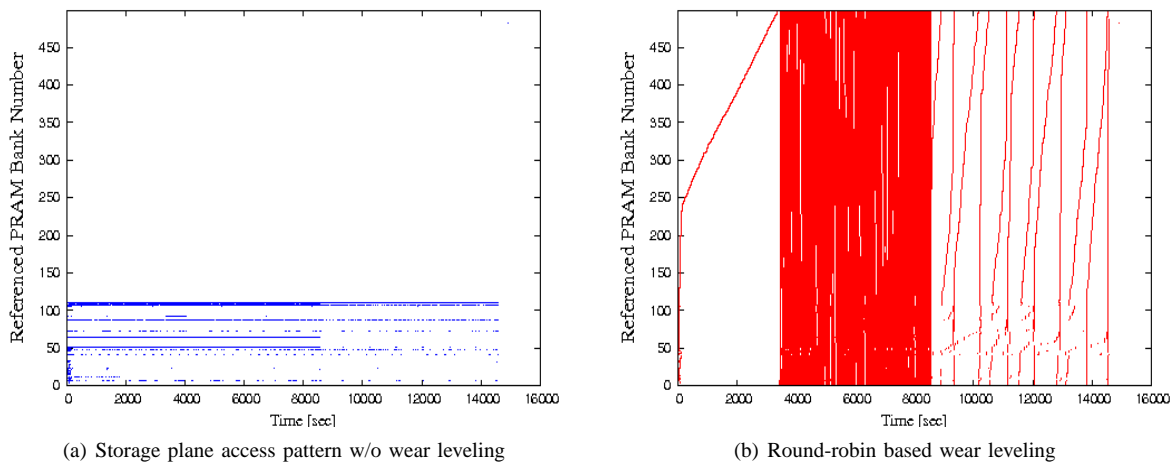


Fig. 6. Wear leveling effect on metadata update.

limited only to physical frame number (PFN) mapped to each virtual page, swap information, PFN-based page mapping count, and a small set of page status such as page locking. `pagemap` mandates users to trace only predefined limited information rather than giving users freedom to determine what information they can trace.

Desnoyers et al. developed a more generic framework for Linux kernel tracing called `LTTng` [28]. It features lightweight kernel tracing and graphical data analysis utilities. The tool targets to provide a general-purpose trace interface for a wide user base. Unfortunately, we find the tool not immediately usable for specific storage research tasks because it requires heavy customization. `PRISM` is a more intuitive tool for storage systems research.

Storage architecture simulators. `DiskSim` [29] simulates a variety of HDD models and disk arrays, and has been in widespread use. It is essentially an event-driven simulator that models major architectural components in a disk system having a block I/O interface like SCSI. `DiskSim` can be used as a standalone simulator when traces have been prepared, or can be used on-line in connection with a separate full-system simulator. We note however that `DiskSim` was not originally developed to study persistent RAM storage or SSDs, which have quite different physical characteristics from rotating HDD.

Thanks to the growing popularity and importance of SSDs, there are a handful of SSD simulators developed recently. Agrawal et al. [13] describes a SSD model that can be plugged into `DiskSim`. It offers essential configurability to specify and simulate a generic SSD model. Kim et al. [30] describes a standalone SSD simulator called `FlashSim`. Their focus is to explore various FTL schemes at a behavioral level assuming the traditional block I/O interface. Lee et al. [31] reports another relatively simple SSD simulator called `CPS-SIM`. `CPS-SIM` tries to determine an optimal hardware organization in terms of the number of buses and flash chips as well as the interconnection. However, their work does not model important I/O queuing effect or realistic workloads. `DiskSim`

and the above SSD simulators are fundamentally a block I/O storage simulator and are not easily re-targeted to model a main memory based persistent RAM storage.

Recently, Dong et al. [32] developed a simulator to study PCM-based main memory and cache. Their main contribution is to provide a system-level tool beyond the device-level research of PCM by automating the process of finding an optimal PCM array organization, much like `CACTI` [33]. While they evaluated the performance of main memory and processor cache built with PCM, they didn't deal with the potential of a persistent RAM based storage; the simulator deficits the ability to trace OS kernel activities and does not model and evaluate wear leveling schemes.

In comparison with the above tools, `PRISM` features the full functionalities necessary to study a persistent RAM storage. It can generate byte-addressable storage trace using real workloads and simulate a configurable persistent RAM storage model. With fast trace collection and simulation speed and convenient data collection and processing utilities, it enables the user to study specific aspects of a storage system as well as the impact of variations in the hardware organization.

VI. CONCLUSIONS AND FUTURE WORK

In this paper, we introduced `PRISM`, a novel infrastructure to explore various design trade-offs of emerging persistent RAM storage. While researchers are requested to evaluate persistent RAM storages, there is no publicly available tool that can run realistic workloads while exposing low, memory-level behavior of the modeled storage. `PRISM` fills this gap.

`PRISM` is different from existing I/O tracers and storage hardware simulators; existing tools focus primarily on block I/O traffic and do not provide facilities to track low-level activities in a storage system that is tightly coupled with the system's main memory. Moreover, `PRISM` incorporates both front-end tracing and back-end simulation as well as data post-processing utilities in a single integrated framework to maximize its usability.

To demonstrate the usefulness of `PRISM`, we presented a case study involving an OLTP workload with a goal to

enhance the write endurance of a persistent RAM storage. Our experimental results show that PRISM offers excellent observability and configurability during evaluating various persistent RAM storage organizations.

As future work, we will continue to elaborate PRISM (e.g., timing [24]) and extensively explore the design space of persistent RAM storage systems.

ACKNOWLEDGEMENT

This work was supported in part by the US National Science Foundation (NSF) under grants CNS-1012070 and CCF-0952273.

REFERENCES

- [1] R. F. Freitas and W. W. Wilcke, "storage-class memory: The next storage system technology," *IBM Journal of Research and Development*, 52(4):439–447, 2008.
- [2] S. Raoux, G. W. Burr, M. J. Breitwisch, C. T. Rettner, Y.-C. Chen, R. M. Shelby, M. Salinga, D. Krebs, S.-H. Chen, H.-L. Lung, and C. H. Lam, "Phase-Change Random Access Memory: A Scalable Technology," *IBM Journal of Research and Development*, 52(4):465–479, 2008.
- [3] A. D. Smith and Y. Huai, "STT-RAM—A New Spin on Universal Memory," *Future Fab Intl.*, Issue 23, 2007.
- [4] M. H. Kryder and C. S. Kim, "After Hard Drives—What Comes Next?," *IEEE Transactions on Magnetics*, 45(10):3406–3413, 2009.
- [5] X. Dong, X. Wu, G. Sun, Y. Xie, H. Li, and Y. Chen, "Circum and microarchitecture evaluation of 3D stacking magnetic RAM (MRAM) as a universal memory replacement," *Proceedings of the 45th Design Automation Conference (DAC)*, pp 554–559, 2009.
- [6] H. Shiga et al., "A 1.6 GB/s DDR2 128 Mb chain FeRAM with scalable octal bitline and sensing schemes," *IEEE Int'l Solid-State Circuits Conference (ISSCC)*, Session 27, 2009.
- [7] Samsung Electronics, SS805 product specification, http://www.samsung.com/global/business/semiconductor/products/flash/ssd/2008/down/ss805_25_inch.pdf.
- [8] Mtron, Solid state drive msd-sata 3035 product specification, http://mtron.net/Upload_Data/Spec/ASiC/MOBI/SATA/MSD-SATA3035_rev0.4.pdf.
- [9] FusionIO Corporation, ioExtreme Datasheet, <http://www.fusionio.com/products/ioxtreme>.
- [10] J. Condem, E. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee, "Better I/O through byte-addressable, persistent memory," *Proceedings of the 22nd Symposium on Operating Systems Principles (SOSP'09)*, pp 133–146, 2009.
- [11] J. C. Mogul, E. Argollo, M. Shah, and P. Faraboschi, "Operating System Support for NVM+DRAM Hybrid Main Memory," *Proceedings of the 12th Workshop on Hot Topics in Operating Systems (HotOS XII)*, pp 14–14, 2009.
- [12] M. Wu and W. Zwaenepoel, "eNVy: a non-volatile, main memory storage system," *Proceedings of the 6th international conference on Architectural support for programming languages and operating systems (ASPLOS)* pp 86–97, 1994.
- [13] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy, "Design Tradeoffs for SSD Performance," *Proceedings of the 2008 USENIX Technical Conference (USENIX'08)*, pp 57–70, 2008.
- [14] M. Saxena and M. M. Swift, "FlashVM: Revisiting the Virtual Memory Hierarchy," *Proceedings of the 12th Workshop on Hot Topics in Operating Systems (HotOS XII)*, pp 13–13, 2009.
- [15] J. Jung, Y. Won, E. Kim, H. Shin, and B. Jeon, "FRASH: Exploiting Storage Class Memory in Hybrid File System for Hierarchical Storage," *ACM Transactions on Storage (TOS)*, 6(1), 2010.
- [16] I. H. Do, H. Lee, Y. Moon, E. Kim, J. Choi, D. Lee, and S. Noh, "Impact of NVRAM Write Cache for File System Metadata on I/O Performance in Embedded Systems," *Proceedings of the 2009 ACM Symposium on Applied Computing (SAC)*, pp 1658–1663, 2009.
- [17] M. K. Qureshi, V. Srinivasan, and J. Rivers, "Scalable High-Performance Main Memory System Using Phase-Change Memory Technology," *Proceedings of the 36th Intl Symposium on Computer Architecture (ISCA)*, pp 24–33, 2009.
- [18] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecting phase change memory as a scalable DRAM alternative," *Proceedings of the 36th Intl Symposium on Computer Architecture (ISCA)*, pp 2–13, 2009.
- [19] P. Zhou, B. Zhao, J. Yang, and Y. Zhang, "A Durable and Energy Efficient Main Memory Using Phase Change Memory Technology," *Proceedings of the 36th Intl Symposium on Computer Architecture (ISCA)*, pp 14–23, 2009.
- [20] Linux tmpfs, <http://lxr.linux.no/#linux+v2.6.24.7/Documentation/filesystems/tmpfs.txt>.
- [21] Transaction Processing Performance Council, TPC Benchmark C, Standard Specification, http://www.tpc.org/tpcc/spec/tpcc_current.pdf.
- [22] D. R. Llanos, "TPCC-UVa: an open-source TPC-C implementation for global performance measurement of computer systems," *ACM SIGMOD Record*, 35(4):6–15, 2006.
- [23] IOzone, IOzone Filesystem Benchmark, <http://www.iozone.org>.
- [24] J. L. Griffin, J. Schindler, S. Schlosser, J. S. Bucy, and G. R. Ganger, "Timing-accurate Storage Emulation," *Proceedings of the Conference on File and Storage Technologies (FAST)*, pp 75–88, 2002.
- [25] A. D. Brunelle, "Block I/O Layer Tracing: blktrace," *Gelato-Itanium Conference and Expo (gelato-ICE)*, 2006.
- [26] M. Russinovich, "DiskMon for Windows v2.01," <http://technet.microsoft.com/en-us/sysinternals/bb896646.aspx>.
- [27] Linux pagemap, <http://lxr.linux.no/#linux+v2.6.35/Documentation/vm/pagemap.txt>.
- [28] M. Desnoyers and M. R. Dagenais, "The LTTng tracer: A low impact performance and behavior monitor for GNU/Linux," *Proceedings of Ottawa Linux Symposium (OLS)*, vol. 1, pp 209–224, 2006.
- [29] J. S. Bucy, J. Schindler, S. Schlosser, and G. R. Ganger, "The DiskSim simulation environment version 4.0 reference manual," *Technical Report CMU-PDL-08-101*, Carnegie Mellon University, May 2008.
- [30] Y. Kim, B. Tauras, A. Gupta, and B. Urgaonkar, "FlashSim: A Simulator for NAND Flash-based Solid-State Drives," *International Conference on Advances in System Simulation (SIMUL)*, pp 125–131, 2009.
- [31] J. Lee, E. Byun, H. Park, J. Choi, D. Lee, and S. Noh, "CPS-SIM: Configurable and Accurate Clock Precision Solid State Drive Simulator," *Proceedings of Symposium on Applied Computing (SAC)*, pp 318–325, 2009.
- [32] X. Dong, N. Jouppi, and Y. Xie, "PCRAMsim: System-Level Performance, Energy, and Area Modeling for Phase-Change RAM," *Proceedings of International Conference on Computer Aided Design (ICCAD)*, pp 269–275, 2009.
- [33] CACTI, An integrated cache and memory access time, cycle time, area, leakage, and dynamic power model <http://www.hpl.hp.com/research/cacti/>.
- [34] L. P.-. Chang, "Hybrid Solid-State Disks: Combining Heterogeneous NAND Flash in Large SSDs," *Proceedings of Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp 428–433, 2008.
- [35] K.-J. Lee, B.-H. Cho, W.-Y. Cho, S.-B. Kang, B.-G. Choi, et al., "A 90 nm 1.8 V 512 Mb Diode-Switch PRAM With 266 MB/s Read Throughput," *IEEE Journal of Solid-State Circuits*, 43(1):150–162, 2008.