FlashVM: Virtual Memory Management on Flash

Mohit Saxena and Michael M. Swift
University of Wisconsin-Madison
{msaxena,swift}@cs.wisc.edu

Abstract

With the decreasing price of flash memory, systems will increasingly use solid-state storage for virtual-memory paging rather than disks. *FlashVM* is a system architecture and a core virtual memory subsystem built in the Linux kernel that uses dedicated flash for paging.

FlashVM focuses on three major design goals for memory management on flash: high performance, reduced flash wear out for improved reliability, and efficient garbage collection. FlashVM modifies the paging system along code paths for allocating, reading and writing back pages to optimize for the performance characteristics of flash. It also reduces the number of page writes using zero-page sharing and page sampling that prioritize the eviction of clean pages. In addition, we present the first comprehensive description of the usage of the *discard* command on a real flash device and show two enhancements to provide fast online garbage collection of free VM pages.

Overall, the FlashVM system provides up to 94% reduction in application execution time and is four times more responsive than swapping to disk. Furthermore, it improves reliability by writing up to 93% fewer pages than Linux, and provides a garbage collection mechanism that is up to 10 times faster than Linux with discard support.

1 Introduction

Flash memory is one of the largest changes to storage in recent history. Solid-state disks (SSDs), composed of multiple flash chips, provide the abstraction of a block device to the operating system similar to magnetic disks. This abstraction favors the use of flash as a replacement for disk storage due to its faster access speeds and lower energy consumption [1, 25, 33].

In this paper, we present FlashVM, a system architecture and a core virtual memory subsystem built in the Linux kernel for managing flash-backed virtual memory. FlashVM extends a traditional system organization with dedicated flash for swapping virtual memory pages. Dedicated flash allows FlashVM software to use semantic information, such as the knowledge about free blocks, that is not available within an SSD. Furthermore, dedicating flash to virtual memory is economically attractive, because small quantities can be purchased for a few dollars. In contrast, disks ship only in large sizes at higher initial costs.

The design of FlashVM focuses on three aspects of flash: performance, reliability, and garbage collection. We analyze the existing Linux virtual memory implementation and modify it to account for flash characteristics. FlashVM modifies the paging system on code paths affected by the performance differences between flash and disk: on the read path during a page fault, and on the write path when pages are evicted from memory. On the read path, FlashVM leverages the low seek time on flash to prefetch more useful pages. The Linux VM prefetches eight physically contiguous pages to minimize disk seeks. FlashVM uses stride prefetching to minimize memory pollution with unwanted pages at a negligible cost of seeking on flash. This results in a reduction in the number of page faults and improves the application execution time. On the write path, FlashVM throttles the page write-back rate at a finer granularity than Linux. This allows better congestion control of paging traffic to flash and improved page fault latencies for various application workloads.

The write path also affects the *reliability* of FlashVM. Flash memory suffers from wear out, in that a single block of storage can only be written a finite number of times. FlashVM uses *zero-page sharing* to avoid writing empty pages and uses *page sampling*, which probabilistically skips over dirty pages to prioritize the replacement of clean pages. Both techniques reduce the number of page writes to the flash device, resulting in improved reliability for FlashVM.

The third focus of FlashVM is efficient *garbage collection*, which affects both reliability and performance. Modern SSDs provide a *discard* command (also called *trim*) for the OS to notify the device when blocks no longer contain valid data [30]. We present the first comprehensive description of the semantics, usage and performance characteristics of the discard command on a real SSD. In addition, we propose two different techniques, merged and dummy discards, to optimize the use of the discard command for online garbage collection of free VM page clusters on the swap device. Merged discard batches requests for discarding multiple page clusters in a single discard command. Alternatively, dummy

discards implicitly notify the device about free VM pages by overwriting a logical flash block.

We evaluate the costs and benefits of each of these design techniques for FlashVM for memory-intensive applications representing a variety of computing environments including netbooks, desktops and distributed clusters. We show that FlashVM can benefit a variety of workloads including image manipulation, model checking, transaction processing, and large key-value stores. Our results show that:

- FlashVM provides up to 94% reduction in application execution time and up to 84% savings in memory required to achieve the same performance as swapping to disk. FlashVM also scales with increased degree of multiprogramming and provides up to four times faster response time to revive suspended applications.
- FlashVM provides better flash reliability than Linux by reducing the number of page writes to the swap device. It uses zero-page sharing and dirty page sampling for preferential eviction of clean pages, which result in up to 93% and 14% fewer page writes respectively.
- FlashVM optimizes the performance for garbage collection of free VM pages using merged and dummy discard operations, which are up to 10 times faster than Linux with discard support and only 15% slower than Linux without discard support.

The remainder of the paper is structured as follows. Section 2 describes the target environments and makes a case for FlashVM. Section 3 presents FlashVM design overview and challenges. We describe the design in Sections 4.1, covering performance; 4.2 covering reliability; and 4.3 on efficient garbage collection using the discard command. We evaluate the FlashVM design techniques in Section 5, and finish with related work and conclusions.

2 Motivation

Application working-set sizes have grown many-fold in the last decade, driving the demand for cost-effective mechanisms to improve memory performance. In this section, we motivate the use of flash-backed virtual memory by comparing it to DRAM and disk, and noting the workload environments that benefit the most.

2.1 Why FlashVM?

Fast and cheap flash memory has become ubiquitous. More than 2 *exabytes* of flash memory were manufactured worldwide in 2008. Table 1 compares the price and performance characteristics of NAND flash memory with DRAM and disk. Flash price and performance are between DRAM and disk, and about five times cheaper
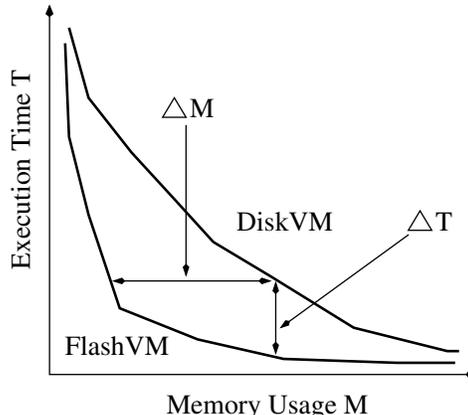

Figure 1: **Cost**/**Benefit Analysis:** *Application execution time plot comparing the performance of disk and flash backed virtual memory with variable main memory sizes. ΔM is the memory savings to achieve the same performance as disk and ΔT is the performance improvement with FlashVM for the same memory size.*

| Device | Read Latencies (μs) | | Write Latencies (μs) | | Price |
	Random	Seq	Random	Seq	$/GB
DRAM	0.05	0.05	0.05	0.05	$15
Flash	100	85	2,000	200-500	$3
Disk	5,000	500	5,000	500	$0.3

Table 1: **Device Attributes:** *Comparing DRAM, NAND flash memory and magnetic disk. Both price and performance for flash lie between DRAM and disk (some values are roughly defined for comparison purposes only).*

than DRAM and an order of magnitude faster than disk. Furthermore, flash power consumption (0.06 W when idle and 0.15–2 W when active) is significantly lower than both DRAM (4–5 W/DIMM) and disk (13–18 W). These features of flash motivate its adoption as second-level memory between DRAM and disk.

Figure 1 illustrates a cost/benefit analysis in the form of two simplified curves (not to scale) showing the execution times for an application in two different systems configured with variable memory sizes, and either disk or flash for swapping. This figure shows two benefits to applications when they must page. First, FlashVM results in faster execution for approximately the same system price without provisioning additional DRAM, as flash is five times cheaper than DRAM. This performance gain is shown in Figure 1 as ΔT along the vertical axis. Second, a FlashVM system can achieve performance similar to swapping to disk with lower main memory requirements. This occurs because page faults are an order of magnitude faster with flash than disk: a program can achieve the same performance with less memory by faulting more frequently to FlashVM. This reduction in memory-resident working set is shown as ΔM along the

horizontal axis.

However, the adoption of flash is fundamentally an economic decision, as performance can also be improved by purchasing additional DRAM or a faster disk. Thus, careful analysis is required to configure the balance of DRAM and flash memory capacities that is optimal for the target environment in terms of both price and performance.

2.2 Where FlashVM?

Both the price/performance gains for FlashVM are strongly dependent on the workload characteristics and the target environment. In this paper, we target FlashVM against the following workloads and environments:

Netbook/Desktop. Netbooks and desktops are usually constrained with cost, the number of DIMM slots for DRAM modules and DRAM power-consumption. In these environments, the large capacity of disks is still desirable. Memory-intensive workloads, such as image manipulation, video encoding, or even opening multiple tabs in a single web browser instance can consume hundreds of megabytes or gigabytes of memory in a few minutes of usage [4], thereby causing the system to page. Furthermore, end users often run multiple programs, leading to competition for memory. FlashVM meets the requirements of such workloads and environments with faster performance that scales with increased multiprogramming.

Distributed Clusters. Data-intensive workloads such as virtualized services, key-value stores and web caches have often resorted to virtual or distributed memory solutions. For example, the popular *memcached* [17] is used to increase the aggregate memory bandwidth. While disk access is too slow to support page faults during request processing, flash access times allow a moderate number of accesses. Fast swapping can also benefit virtual machine deployments, which are often constrained by the main memory capacities available on commonly deployed cheap servers [19]. Virtual machine monitors can host more virtual machines with support for swapping out nearly the entire guest physical memory. In such cluster scenarios, hybrid alternatives similar to FlashVM that incorporate DRAM and large amounts of flash are an attractive means to provide large memory capacities cheaply [6].

3 Design Overview

The FlashVM design, shown in Figure 2, consists of dedicated flash for swapping out virtual memory pages and changes to the Linux virtual memory hierarchy that optimize for the characteristics of flash. We target FlashVM against NAND flash, which has lower prices and better write performance than the alternative, NOR flash. We propose that future systems be built with a small multi-ple of DRAM size as flash that is attached to the motherboard for the express purpose of supporting virtual memory.

Flash Management. Existing solid-state disks (SSD) manage NAND flash memory packages internally for emulating disks [1]. Because flash devices do not support re-writing data in place, SSDs rely on a translation layer to implement block address translation, wear leveling and garbage collection of free blocks. The translation from this layer raises three problems not present with disks: write amplification, low reliability, and aging.

Write amplification occurs when writing a single block causes the SSD to re-write multiple blocks, and leads to expensive read-modify-erase-write cycles (erase latency for a typical 128–512 KB flash block is as high as 2 milliseconds) [1, 26]. SSDs may exhibit *low reliability* because a single block may only be re-written a finite number of times. This limit is around 10,000 and is decreasing with the increase in capacity and density of MLC flash devices. Above this limit, devices may exhibit unacceptably high bit error rates [18]. For a 16 GB SSD written at its full bandwidth of 200 MB/sec, errors may arise in as little as a few weeks. Furthermore, SSDs exhibit *aging* after extensive use because fewer clean blocks are available for writing [24, 26]. This can lead to performance degradation, as the device continuously copies data to clean pages. The FlashVM design leverages semantic information only available within the operating system, such as locality of memory references, page similarity and knowledge about deleted blocks, to address these three problems.

FlashVM Architecture. The FlashVM architecture targets dedicated flash for virtual memory paging. Dedicating flash for virtual memory has two distinct advantages over traditional disk-based swapping. First, dedicated flash is cheaper than traditional disk-based swap devices in price per byte only for small capacities required for virtual memory. A 4 GB MLC NAND flash chip costs less than $6, while the cheapest IDE/SCSI disk of similar size costs no less than $24 [5, 32]. Similarly, the more common SATA/SAS disks do not scale down to capacities smaller than 36 GB, and even then are far more expensive than flash. Furthermore, the premium for managed flash, which includes a translation layer, as compared to raw flash chips is dropping rapidly as SSDs mature. Second, dedicating flash for virtual memory minimizes the interference between the file system I/O and virtual-memory paging traffic. We prototype FlashVM using MLC NAND flash-based solid-state disks connected over a SATA interface.

FlashVM Software. The FlashVM memory manager, shown in Figure 2, is an enhanced version of the memory management subsystem in the Linux 2.6.28 kernel.
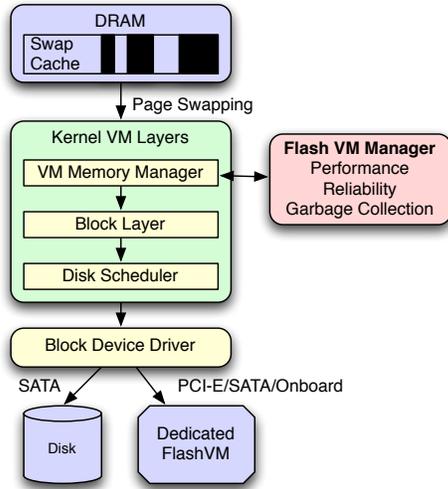
Figure 2: **FlashVM Memory Hierarchy:** *FlashVM manager controls the allocation, read and write-back of pages swapped out from main memory. It hands the pages to the block layer for conversion into block I/O requests, which are submitted to the dedicated flash device.*

Since NAND flash is internally organized as a block device, the FlashVM manager enqueues the evicted pages at the block layer for scheduling. The block layer is responsible for the conversion of pages into block I/O requests submitted to the device driver. At a high-level, FlashVM manages the non-ideal characteristics of flash and exploits its useful attributes. In particular, our design goals are:

- *High performance* by leveraging the unique performance characteristics of flash, such as fast random reads (discussed in Section 4.1).
- *Improved reliability* by reducing the number of page writes to the swap device (discussed in Section 4.2).
- *Efficient garbage collection* of free VM pages by delaying, merging, and virtualizing discard operations (discussed in Section 4.3).

The FlashVM implementation is not a singular addition to the Linux VM. As flash touches on many aspects of performance, FlashVM modifies most components of the Linux virtual memory hierarchy, including the swap-management subsystem, the memory allocator, the page scanner, the page-replacement and prefetching algorithms, the block layer and the SCSI subsystem. In the next section, we identify and describe our changes to each of these subsystems for achieving the different FlashVM design goals.

4   Design and Implementation

This section discusses the FlashVM implementation to improve performance, reliability, and to provide efficient garbage collection.

4.1   FlashVM Performance

Challenges. The virtual-memory systems of most operating systems were developed with the assumption that disks are the only swap device. While disks exhibit a range of performance, their fundamental characteristic is the speed difference between random and sequential access. In contrast, flash devices have a different set of performance characteristics, such as fast random reads, high sequential bandwidths, low access and seek costs, and slower writes than reads. For each code path in the VM hierarchy affected by these differences between flash and disk, we describe our analysis for tuning parameters and our implementation for optimizing performance with flash. We analyze three VM mechanisms: *page pre-cleaning*, *page clustering* and *disk scheduling*, and re-implement *page scanning* and *prefetching* algorithms.

4.1.1   Page Write-Back

Swapping to flash changes the performance of writing back dirty pages. Similar to disk, random writes to flash are costlier than sequential writes. However, random reads are inexpensive, so write-back should optimize for write locality rather than read locality. FlashVM accomplishes this by leveraging the *page pre-cleaning* and *clustering* mechanisms in Linux to reduce page write overheads.

Pre-cleaning.   Page pre-cleaning is the act of eagerly swapping out dirty pages before new pages are needed. The Linux page-out daemon *kswapd* runs periodically to write out 32 pages from the list of inactive pages. The higher write bandwidth of flash allows FlashVM write pages more aggressively, and without competing file system traffic, use more I/O bandwidth.

Thus, we investigate writing more pages at a time to achieve sequential write performance on flash. For disks, pre-cleaning more pages interferes with high-priority reads to service a fault. However, with flash, the lower access latency and higher bandwidths enable more aggressive pre-cleaning without affecting the latency for handling a page-fault.

Clustering. The Linux clustering mechanism assigns locations in the swap device to pages as they are written out. To avoid random writes, Linux allocates *clusters*, which are contiguous ranges of *page slots*. When a cluster has been filled, Linux scans for a free cluster from the start of the swap space. This reduces seek overheads on disk by consolidating paging traffic near the beginning of the swap space.

We analyze FlashVM performance for a variety of cluster sizes from 8 KB to 4096 KB. In addition, for clusters at least the size of an erase block, we align clusters
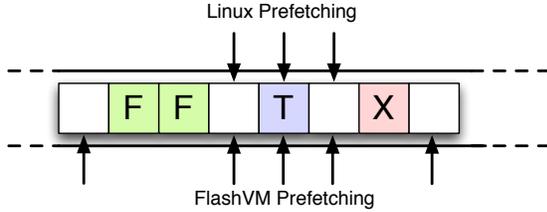
Figure 3: **Virtual Memory Prefetching**: *Linux reads-around an aligned block of pages consisting of the target page and delimited by free or bad blocks to minimize disk seeks. FlashVM skips the free and bad blocks by seeking to the next allocated page and reads all valid pages. (T, F and X represent target, free and bad blocks on disk respectively; unfilled boxes represent the allocated pages).*

with erase-block boundaries to ensure minimum amount of data must be erased.

4.1.2 Page Scanning

A virtual memory system must ensure that the rate at which it selects pages for eviction matches the write bandwidth of the swap device. Pages are selected by two code paths: memory reclaim during page allocation; and the page-out daemon that scan the inactive page list for victim pages. The Linux VM subsystem balances the rate of scanning with the rate of write-back to match the bandwidth of the swap device. If the scanning rate is too high, Linux throttles page write-backs by waiting for up to 20–100 milliseconds or until a write completes. This timeout, appropriate for disk, is more than two orders of magnitude greater than flash access latencies.

FlashVM controls write throttling at a much finer granularity of a system *jiffy* (one clock tick). Since multiple page writes in a full erase block on flash take up to two milliseconds, FlashVM times-out for about one millisecond on our system. These timeouts do not execute frequently, but have a large impact on the average page fault latency [29]. This enables FlashVM to maintain higher utilization of paging bandwidth and speeds up the code path for write-back when reclaiming memory.

4.1.3 Prefetching on Page Fault

Operating systems prefetch pages after a page fault to benefit from the sequential read bandwidth of the device [12]. The existing Linux prefetch mechanism reads in up to 8 pages contiguous on disk around the target page. Prefetching is limited by the presence of free or bad page slots that represent bad blocks on disk. As shown in Figure 3, these page slots delimit the start or the end of the prefetched pages. On disk, this approach avoids the extra cost of seeking around free and bad pages, but often leads to fetching fewer than 8 pages.

FlashVM leverages fast random reads on flash with

two different prefetching mechanisms. First, FlashVM seeks over the free/bad pages when prefetching to retrieve a full set of valid pages. Thus, the fast random access of flash medium enables FlashVM to bring in more pages with spatial locality than native Linux.

Fast random access on flash also allows prefetching of more distant pages with temporal locality, such as stride prefetching. FlashVM records the offsets between the current target page address and the last two faulting addresses. Using these two offsets, FlashVM computes the strides for the next two pages expected to be referenced in the future. Compared to prefetching adjacent pages, stride prefetching reduces memory pollution by reading the pages that are more likely to be referenced soon.

We implement stride prefetching to work in conjunction with contiguous prefetching: FlashVM first reads pages contiguous to the target page and then prefetches stride pages. We find that fetching too many stride pages increases the average page fault latency, so we limit the stride to two pages. These two prefetching schemes result in a reduction in the number of page faults and improve the total execution time for paging.

4.1.4 Disk Scheduling

The Linux VM subsystem submits page read and write requests to the block-layer I/O scheduler. The choice of the I/O scheduler affects scalability with multiprogrammed workloads, as the scheduler selects the order in which requests from different processes are sent to the swap device.

Existing Linux I/O schedulers optimize performance by (i) merging adjacent requests, (ii) reordering requests to minimize seeks and to prioritize requests, and (iii) delaying requests to allow a process to submit new requests. Work-conserving schedulers, such as the NOOP and deadline schedulers in Linux, submit pending requests to the device as soon as the prior request completes. In contrast, non-work-conserving schedulers may delay requests for up to 2–3 ms to wait for new requests with better locality or to distribute I/O bandwidth fairly between processes [9]. However, these schedulers optimize for the performance characteristics of disks, where seek is the dominant cost of I/O. We therefore analyze the impact of different I/O schedulers on FlashVM.

The Linux VM system tends to batch multiple read requests on a page fault for prefetching, and multiple write requests for clustering evicted pages. Thus, paging traffic is more regular than file system workloads in general. Further, delaying requests for locality can lead to lower device utilization on flash, where random access is only a small component of the page transfer cost. Thus, we analyze the performance impact of work conservation when scheduling paging traffic for FlashVM.

4.2 FlashVM Reliability

Challenges. As flash geometry shrinks and MLC technology packs more bits into each memory cell, the prices of flash devices have dropped significantly. Unfortunately, so has the *erasure limit* per flash block. A single flash block can typically undergo between 10,000 and 100,000 erase cycles, before it can no longer reliably store data. Modern SSDs and flash devices use internal wear-leveling to spread writes across all flash blocks. However, the bit error rates of these devices can still become unacceptably high once the erasure limit is reached [18]. As the virtual memory paging traffic may stress flash storage, FlashVM specially manages page writes to improve device reliability. It exploits the information available in the OS about the state and content of a page by employing *page sampling* and *page sharing* respectively. FlashVM aims to reduce the number of page writes and prolong the lifetime of the flash device dedicated for swapping.

4.2.1 Page Sampling

Linux reclaims free memory by evicting inactive pages in a least-recently-used order. Clean pages are simply added to the free list, while reclaiming dirty pages requires writing them back to the swap device.

FlashVM modifies the Linux page replacement algorithm by prioritizing the reclaim of younger *clean* pages over older *dirty* pages. While scanning for pages to reclaim, FlashVM skips dirty pages with a probability dependent on the rate of pre-cleaning. This policy increases the number of clean pages that are reclaimed during each scan, and thus reduces the overall number of writes to the flash device.

The optimal rate for sampling dirty pages is strongly related to the memory reference pattern of the application. For applications with read-mostly page references, FlashVM can find more clean pages to reclaim. However, for applications that frequently modify many pages, skipping dirty pages for write-back leads to more frequent page faults, because younger clean pages must be evicted.

FlashVM addresses workload variations with adaptive page sampling: the probability of skipping a dirty page also depends on the write rate of the application. FlashVM predicts the average write rate by maintaining a moving average of the time interval t_n for writing n dirty pages. When the application writes to few pages and t_n is large, FlashVM more aggressively skips dirty pages. For applications that frequently modify many pages, FlashVM reduces the page sampling probability unless n pages have been swapped out. The balance between the rate of page sampling and page writes is adapted to provide a smooth tradeoff between device lifetime and application performance.

4.2.2 Page Sharing

The Linux VM system writes back pages evicted from the LRU inactive list without any knowledge of page content. This may result in writing many pages to the flash device that share the same content. Detecting identical or similar pages may require heavyweight techniques like explicitly tracking changes to each and every page by using transparent page sharing [3] or content-based page sharing by maintaining hash signatures for all pages [8]. These techniques reduce the memory-resident footprint and are orthogonal to the problem of reducing the number of page write-backs.

We implement a limited form of content-based sharing in FlashVM by detecting the swap-out of *zero pages* (pages that contain only zero bytes). Zero pages form a significant fraction of the memory-footprint of some application workloads [8]. FlashVM intercepts paging requests for all zero pages. A swap-out request sets a zero flag in the corresponding page slot in the swap map, and skips submitting a block I/O request. Similarly, a swap-in request verifies the zero flag, which if found set, allocates a zero page in the address space of the application. This extremely lightweight page sharing mechanism saves both the memory allocated for zero pages in the main-memory swap cache and the number of page write-backs to the flash device.

4.3 FlashVM Garbage Collection

Challenges. Flash devices cannot overwrite data in place. Instead, they must first erase a large flash block (128–512 KB), a slow operation, and then write to pages within the erased block. Lack of sufficient pre-erased blocks may result in copying multiple flash blocks for a single page write to: (i) replenish the pool of clean blocks, and (ii) ensure uniform wear across all blocks. Therefore, flash performance and overhead of wear management are strongly dependent on the number of clean blocks available within the flash device. For example, high-end enterprise SSDs can suffer up to 85% drop in write performance after extensive use [24, 26]. As a result, efficient garbage collection of clean blocks is necessary for flash devices, analogous to the problem of segment cleaning for log-structured file systems [28].

For virtual memory, sustained paging can quickly *age* the dedicated flash device by filling up all free blocks. When FlashVM overwrites a block, the device can reclaim the storage previously occupied by that block. However, only the VM system has knowledge about empty (free) page clusters. These clusters consist of page slots in the swap map belonging to terminated processes, dirty pages that have been read into the memory and all blocks on the swap device after a reboot. Thus, a flash device that implements internal garbage collection or wear-leveling may unnecessarily copy stale data,
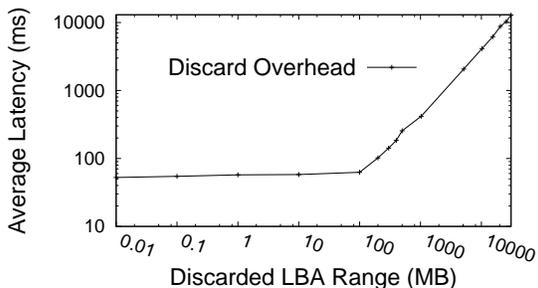
Figure 4: **Discard Overhead**: *Impact of the number of blocks discarded on the average latency of a single discard command. Both x-axis and y-axis are log-scale.*

reducing performance and reliability when not informed about invalid pages.

FlashVM addresses this problem by explicitly notifying the flash device of such pages by using the *discard* command (also called *trim* introduced in the recent SATA SSDs [30]). The discard command has the following semantics:

discard(*dev*, *rangelist*[(sector, nsectors), ...])

where *rangelist* is the list of logical block address ranges to be discarded on the flash device *dev*. Each block range is represented as a pair of the starting sector address and the number of following sectors.

Free blocks can be discarded offline by first flushing all in-flight read/write requests to the flash device, and then wiping the requested logical address space. However, offline discard typically offers very coarse-grain functionality, for example in the form of periodic disk scrubbing or disk format operations [20]. Therefore, FlashVM employs online cleaning that discards a smaller range of free flash page clusters at runtime.

Linux implements rudimentary support for online cleaning in recent kernel versions starting in 2.6.28. When it finds 1 MB of free contiguous pages, it submits a single discard request for the corresponding page cluster. However, Linux does not fully support discard yet: the block layer breaks discard requests into smaller requests of no more than 256 sectors, while the ATA disk driver ignores them. Thus, the existing Linux VM is not able to actually discard the free page clusters. FlashVM instead bypasses the block and ATA driver layers and sends discard commands directly to the flash device through the SCSI layer [14, 16]. Thus, FlashVM has the ability to discard any number of sectors in a single command.

We next present an experimental analysis of the cost of discard on current flash devices. Based on these results,

which show that discard is expensive, we describe two different techniques that FlashVM uses to improve the performance of garbage collection: *merged discard* and *dummy discard*.

4.3.1 Discard Cost

We measure the latency of discard operations on the OCZ-Vertex SSD, which uses the Indilinx flash controller used by many SSD manufacturers. Figure 4 shows the overheads for discard commands issued over block address ranges of different sizes. Based on Figure 4, we infer the cost of a single discard command for cleaning B flash blocks in one or more address ranges, each having an average utilization u of valid (not previously cleaned) pages:

$$cost_M = \left\{ \begin{array}{ll} c_o & \text{if } B \leq B_o \\ c_o + m \cdot u \cdot (B - B_o) & \text{otherwise} \end{array} \right.$$

In this equation, c_o is the fixed cost of discarding up to B_o blocks and m is the marginal cost of discarding each additional block. *Interestingly, the fixed cost of a single discard command is 55 milliseconds!* We speculate that this overhead occurs because the SSD controller performs multiple block erase operations on different flash channels when actually servicing a discard command [15]. The use of an on-board RAM buffer conceals the linear increase only up to a range of B_o blocks lying between 10–100 megabytes.

The cost of discard is exacerbated by the effect of command queuing: the ATA specification defines the discard commands as *untagged*, requiring that every discard be followed by an I/O barrier that stalls the request queue while it is being serviced. Thus, the long latency of discards requires that FlashVM optimize the use of the command, as the overhead incurred may outweigh the performance and reliability benefits of discarding free blocks.

4.3.2 Merged Discard

The first optimization technique that FlashVM uses is *merged discard*. Linux limits the size of each discard command sent to the device to 128 KB. However, as shown in Figure 4, discards get cheaper per byte as range sizes increase. Therefore, FlashVM opportunistically discards larger block address ranges. Rather than discard pages on every scan of the swap map, FlashVM defers the operation and batches discards from multiple scans. It discards the largest possible range list of free pages up to a size of 100 megabytes in a single command.

This approach has three major benefits. First, delaying discards reduces the overhead of scanning the swap map. Second, merging discard requests amortizes the fixed discard cost c_o over multiple block address ranges.

Third, FlashVM merges requests for fragmented and non-contiguous block ranges. In contrast, the I/O scheduler only merges contiguous read/write requests.

4.3.3 Dummy Discard

Discard is only useful when it creates free blocks that can later be used for writes. Overwriting a block *also* causes the SSD to discard the old block contents, but without paying the high fixed costs of the discard command. Furthermore, overwriting a free block removes some of the benefit of discarding to maintain a pool of empty blocks. Therefore, FlashVM implements *dummy discard* to avoid a discard operation when unnecessary.

Dummy discard elides a discard operation if the block is likely to be overwritten soon. This operation implicitly informs the device that the old block is no longer valid and can be asynchronously garbage collected without incurring the fixed cost of a discard command. As FlashVM only writes back page clusters that are integral multiples of erase-block units, no data from partial blocks needs to be relocated. Thus, the cost for a dummy discard is effectively zero.

Unlike merged discards, dummy discards do not make new clean blocks available. Rather, they avoid an ineffective discard, and can therefore only replace a fraction of all discard operations. FlashVM must therefore decide when to use each of the two techniques. Ideally, the number of pages FlashVM discards using each operation depends on the available number of clean blocks, the ratio of their costs and the rate of allocating free page clusters. Upcoming and high-end enterprise SSDs expose the number of clean blocks available within the device [24]. In the absence of such functionality, FlashVM predicts the rate of allocation by estimating the expected time interval t_s between two successive scans for finding a free page cluster. When the system scans frequently, recently freed blocks are overwritten soon, so FlashVM avoids the extra cost of discarding the old contents. When scans occur rarely, discarded clusters remain free for an extended period and benefit garbage collection. Thus, when t_s is small, FlashVM uses dummy discards, and otherwise applies merged discards to a free page cluster in the swap map.

4.4 Summary

FlashVM architecture improves performance, reliability and garbage collection overheads for paging to dedicated flash. Some of the techniques incorporated in FlashVM, such as zero-page sharing, also benefit disk-backed virtual memory. However, the benefit of sharing is more prominent for flash, as it provides both improved performance and reliability.

While FlashVM is designed for managed flash, much of its design is applicable to unmanaged flash as well. In

Device	Sequential (MB/s)		Random 4K-IO/s		Latency
	Read	Write	Read	Write	ms
Seagate Disk	80	68	120-300/s		4-5
IBM SSD	69	20	7K/s	66/s	0.2
OCZ SSD	230	80	5.5K/s	4.6K/s	0.2
Intel SSD	250	70	35K/s	3.3K/s	0.1

Table 2: **Device Characteristics:** *First-generation IBM SSD is comparable to disk in read bandwidth but excels for random reads. Second-generation OCZ-Vertex and Intel SSDs provide both faster read/write bandwidths and IOPS. Write performance asymmetry is more prominent in first-generation SSDs.*

such a system, FlashVM would take more control over garbage collection. With information about the state of pages, it could more effectively clean free pages without an expensive discard operation. Finally, this design avoids the cost of storing persistent mappings of logical block addresses to physical flash locations, as virtual memory is inherently volatile.

5 Evaluation

The implementation of FlashVM entails two components: changes to the virtual memory implementation in Linux and dedicated flash for swapping. We implement FlashVM by modifying the memory management subsystem and the block layer in the x86-64 Linux 2.6.28 kernel. We focus our evaluation on three key questions surrounding these components:

- How much does the FlashVM architecture of *dedicated flash* for virtual memory improve performance compared to traditional disk-based swapping?
- Does FlashVM software design improve *performance*, *reliability* via write-endurance and *garbage collection* for virtual memory management on flash?
- Is FlashVM a *cost-effective* approach to improving system price/performance for different real-world application workloads?

We first describe our experimental setup and methodology and then present our evaluation to answer these three questions in Section 5.2, 5.3 and 5.4 respectively. We answer the first question by investigating the benefits of dedicating flash for paging in Section 5.2. In Section 5.3 and 5.4, we isolate the impact of FlashVM software design by comparing against the native Linux VM implementation.

5.1 Methodology

System and Devices. We run all tests on a 2.5 GHz Intel Core 2 Quad system configured with 4 GB DDR2

DRAM and 3 MB L2 cache per core, although we reduce the amount of memory available to the OS for our tests, as and when mentioned. We compare four storage devices: an IBM first generation SSD, a trim-capable OCZ-Vertex SSD, an Intel X-25M second generation SSD, and a Seagate Barracuda 7200 RPM disk, all using native command queuing. Device characteristics are shown in Table 2.

Application Workloads. We evaluate FlashVM performance with four memory-intensive application workloads with varying working set sizes:

1. *ImageMagick* 6.3.7, resizing a large JPEG image by 500%,
2. *Spin* 5.2 [31], an LTL model checker for testing mutual exclusion and race conditions with a depth of 10 million states,
3. *pseudo-SpecJBB*, a modified SpecJBB 2005 benchmark to measure execution time for 16 concurrent data warehouses with 1 GB JVM heap size using Sun JDK 1.6.0,
4. *memcached* 1.4.1 [17], a high-performance object caching server bulk-storing or looking-up 1 million random 1 KB key-value pairs.

All workloads have a virtual memory footprint large enough to trigger paging and reach steady state for our analysis. For all our experiments, we report results averaged over five different runs. While we tested with all SSDs, we mostly present results for the second generation OCZ-Vertex and Intel SSDs for brevity.

5.2 Dedicated Flash

We evaluate the benefit of dedicating flash to virtual memory by: (i) measuring the costs of sharing storage with the file system, which arise from scheduling competing I/O traffic, and (ii) comparing the scalability of virtual memory with traditional disk-based swapping.

5.2.1 Read/Write Interference

With disk, the major cost of interference is the seeks between competing workloads. With an SSD, however, seek cost is low and the cost of interference arises from interleaving reads and writes from the file and VM systems. Although this cost occurs with disks as well, it is dominated by the overhead of seeking. We first evaluate the performance loss from interleaving, and then measure the actual amount of interleaving with FlashVM.

We use a synthetic benchmark that reads or writes a sequence of five contiguous blocks. Figure 5(a) shows I/O performance as we interleave reads and writes for disk, IBM SSD and Intel SSD. For disk, I/O performance drops from its sequential read bandwidth of 80 MB/s to 8 MB/s when the fraction of interleaved writes reaches 60% because the drive head has to be repositioned be-

tween read and write requests. On flash, I/O performance also degrades as the fraction of writes increase: IBM and Intel SSDs performance drops by 10x and 7x respectively when 60 percent of requests are writes. Thus, interleaving can severely reduce system performance.

These results demonstrate the potential improvement from dedicated flash, because, unlike the file system, the VM system avoids interleaved read and write requests. To measure this ability, we traced the block I/O requests enqueued at the block layer by the VM subsystem using Linux *blktrace*. Page read and write requests are governed by prefetching and page-out operations, which batch up multiple read/write requests together. On analyzing the average length of read request streams interleaved with write requests for ImageMagick and Spin, we found that FlashVM submits long strings of read and write requests. The average length of read streams ranges between 138–169 I/O requests, and write streams are between 170–230 requests. Thus, the FlashVM system architecture benefits from dedicating flash without interleaved reads and writes from the file system.

5.2.2 Scaling Virtual Memory

Unlike flash, dedicating a disk for swapping does not scale with multiple applications contending for memory. This scalability manifests in two scenarios: increased throughput as the number of threads or programs increases, and decreased interference between programs competing for memory.

Multiprogramming. On a dedicated disk, competing programs degenerate into random page-fault I/O and high seek overheads. Figure 5(b) compares the paging throughput on different devices as we run multiple instances of ImageMagick. Performance, measured by the rate of page faults served per second, degrades for both disk and the IBM SSD with as few as 3 program instances, leading to a CPU utilization of 2–3%. For the IBM SSD, performance falls largely due to an increase in the random write traffic, which severely degrades its performance.

In contrast, we find improvement in the effective utilization of paging bandwidth on the Intel SSD with an increase in concurrency. At 5 instances, paging traffic almost saturates the device bandwidth: for each page fault FlashVM prefetches an additional 7 pages, so it reads 96 MB/s to service 3,000 page faults per second. In addition, it writes back a proportional but lower number of pages. Above 5 instances of ImageMagick, the page fault service rate drops because of increased congestion for paging traffic: CPU utilization falls from 54% with 5 concurrent programs to 44% for 8 programs, and write traffic nears the bandwidth of the device. Nevertheless, these results demonstrate that performance scales significantly as multiprogramming increases on flash when
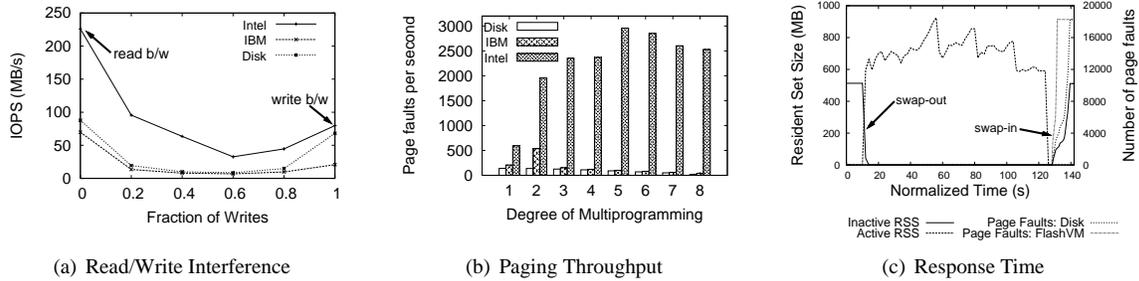
Figure 5: **Dedicated Flash:** *Impact of dedicating flash for VM on performance for read/write interference with file system traffic, paging throughput for multiprogrammed workloads and response time for increased memory contention.*

compared to disk. We find similar increase in paging throughput on dedicated flash for multithreaded applications like memcached. FlashVM performance and device utilization increase when more threads generate more simultaneous requests. This is much the same argument that exists for hardware multithreading to increase parallelism in the memory system.

Response Time. The second scalability benefit of dedicated flash is faster response time for demand paging when multiple applications contend for memory. Figure 5(c) depicts the phenomena frequently observed on desktops when switching to inactive applications. We model this situation with two processes each having working-set sizes of 512 MB and 1.5 GB, that contend for memory on a system configured with 1 GB of DRAM. The curves show the resident set sizes (the amount of physical memory in use by each process) and the aggregate number of page faults in the system over a time interval of 140 seconds. The first process is active for the first 10 seconds and is then swapped out by the Linux VM to accommodate the other process. When 120 seconds have elapsed, the second process terminates and the first process resumes activity.

Demand paging the first process back into memory incurs over 16,000 page faults. With disk, this takes 11.5 seconds and effectively prevents all other applications from accessing the disk. In contrast, resuming the first process takes only 3.5 seconds on flash because of substantially lower flash access latency. Thus, performance degrades much more acceptably with dedicated flash than traditional disk-based swapping, leading to better scalability as the number of processes increase.

5.3 FlashVM Software Evaluation

FlashVM enhances the native Linux virtual memory system for improved performance, reliability and garbage collection. We first analyze our optimizations to the existing VM mechanisms required for improving flash performance, followed by our enhancements for wear management and garbage collection of free blocks.

5.3.1 Performance Analysis

We analyze the performance of different codes paths that impact the paging performance of FlashVM.

Page pre-cleaning. Figure 6(a) shows the performance of FlashVM for ImageMagick, Spin and memcached as we vary the number of pages selected for write-back (pre-cleaned) on each page fault. Performance is poor when only two pages are written back because the VM system frequently scans the inactive list to reclaim pages. However, we find that performance does not improve when pre-cleaning more than 32 pages, because the overhead of scanning is effectively amortized at that point.

Page clustering. Figure 6(b) shows FlashVM performance as we vary the cluster size, the number of pages allocated contiguously on the swap device, while keeping pre-cleaning constant at 32 pages. When only two pages are allocated contiguously (cluster size is two), overhead increases because the VM system wastes time finding free space. Large cluster sizes lead to more sequential I/O, as pages are allocated sequentially within a cluster. However, our results show that above 32 page clusters, performance again stabilizes. This occurs because 32 pages, or 128 KB, is the size of a flash erase block and is enough to obtain the major benefits of sequential writes on flash. We tune FlashVM with these optimal values for pre-cleaning and cluster sizes for all our further experiments.

Congestion control. We evaluate the performance of FlashVM congestion control by comparing it against native Linux on single- and multi-threaded workloads. We separately measured the performance of changing the congestion timeout for the page allocator and for both the page allocator and the *kswapd* page-out daemon for ImageMagick. With the native Linux congestion control timeout tuned to disk access latencies, the system idles even when there is no congestion.

For single-threaded programs, reducing the timeout for the page allocator from 20ms to 1ms improved performance by 6%, and changing the timeout for *kswapd*

Read-Ahead	Native		Stride	
(# of pages)	PF	Time	PF	Time
2	139K	103.2	118K / 15%	88.5 / 14%
4	84K	96.3	70K / 17%	85.7 / 11%
8	56K	91.5	44K / 21%	85.1 / 7%
16	43K	89.0	28K / 35%	83.5 / 6%

Table 3: **VM Prefetching:** *Impact of native Linux and FlashVM prefetching on the number of page faults and application execution time for ImageMagick. (PF is number of hard page faults in thousands, Time is elapsed time in seconds, and percentage reduction and speedup are shown for the number of page faults and application execution time respectively.)*

in addition leads to a 17% performance improvement. For multithreaded workloads, performance improved 4% for page allocation and 6% for both page allocation and the *kswapd*. With multiple threads, the VM system is less likely to idle inappropriately, leading to lower benefits from a reduced congestion timeout. Nevertheless, FlashVM configures lower timeouts, which better match the latency for page access on flash.

Prefetching. Along the page-fault path, FlashVM prefetches more aggressively than Linux by reading more pages around the faulting address and fetching pages at a stride offset. Table 3 shows the benefit of these two optimizations for ImageMagick. The table lists the number of page faults and performance as we vary the number of pages read-ahead for FlashVM prefetching against native Linux prefetching, both on the Intel SSD.

We find that FlashVM outperforms Linux for all values of read-ahead. The reduction in page faults improves from 15% for two pages to 35% for 16 pages, because of an increase in the difference between the number of pages read for native Linux and FlashVM. However, the speedup decreases because performance is lost to random access that results in increased latency per page fault. More sophisticated application-directed prefetching can provide additional benefits by exploiting a more accurate knowledge of the memory reference patterns and the low seek costs on flash.

Disk Scheduling. FlashVM depends on the block layer disk schedulers for merging or re-ordering I/O requests for efficient I/O to flash. Linux has four standard schedulers, which we compare in Figure 6(c). For each scheduler, we execute 4 program instances concurrently and report the completion time of the last program. We scale the working set of the program instances to ensure relevant comparison on each individual device, so the results are not comparable across devices.

On disk, the NOOP scheduler, which only merges adjacent requests before submitting them to the block device driver in FIFO order, performs worst, because it results in long seeks between requests from different pro-

cesses. The deadline scheduler, which prioritizes synchronous page faults over asynchronous writes, performs best. The other two schedulers, CFQ and anticipatory, insert delays to minimize seek overheads, and have intermediate performance.

In contrast, for both flash devices the NOOP scheduler outperforms all other schedulers, outperforming CFQ and anticipatory scheduling by as much as 35% and the deadline scheduler by 10%. This occurs because there is no benefit to localizing seeks on an SSD. We find that average page access latency measured for disk increases linearly from 1 to 6 ms with increasing seek distance. In contrast, for both SSDs, seek time is constant and less than 0.2 ms even for seek distances up to several gigabytes. So, the best schedule for SSDs is to merge adjacent requests and queue up as many requests as possible to obtain the maximum bandwidth. We find that disabling delaying of requests in the anticipatory scheduler results in a 22% performance improvement, but it is still worse than NOOP. Thus, non work-conserving schedulers are not effective when swapping to flash, and scheduling as a whole is less necessary. For the remaining tests, we use the NOOP scheduler.

5.3.2 Wear Management

FlashVM reduces wear-out of flash blocks by write reduction using dirty page sampling and zero-page sharing.

Page Sampling. For ImageMagick, uniformly skipping 1 in 100 dirty pages for write back results in up to 12% reduction in writes but a 5% increase in page faults and a 7% increase in the execution time. In contrast, skipping dirty pages aggressively only when the program has a lower write rate better prioritizes the eviction of clean pages. For the same workload, adaptively skipping 1 in 20 dirty pages results in a 14% write reduction without any increase in application execution time. Thus, adaptive page sampling better reduces page writes with less affect on application performance.

Page Sharing. The number of zero pages swapped out from the inactive LRU list to the flash device is dependent on the memory-footprint of the whole system. Memcached clients bulk-store random keys, leading to few empty pages and only 1% savings in the number of page writes with zero-page sharing. In contrast, both ImageMagick and Spin result in substantial savings. ImageMagick shows up to 15% write reduction and Spin swaps up to 93% of zero pages. We find that Spin preallocates a large amount of memory and zeroes it down before the actual model verification phase begins. Zero-page sharing improves both the application execution time as well as prolongs the device lifetime by reducing the number of page writes.
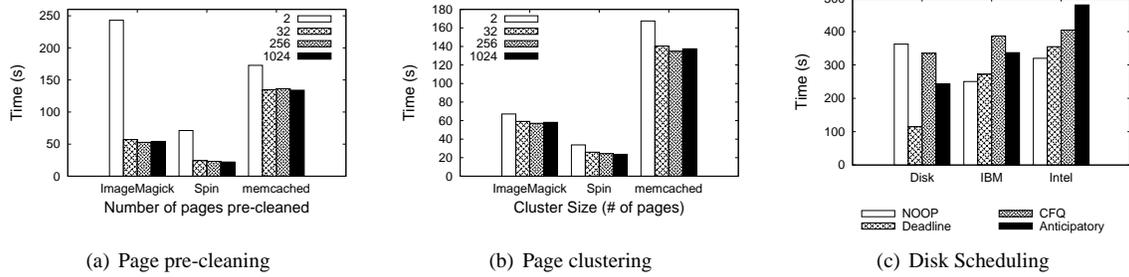
(a) Page pre-cleaning	(b) Page clustering	(c) Disk Scheduling

Figure 6: **Performance Analysis:** *Impact of page pre-cleaning, page clustering and disk scheduling on FlashVM performance for different application workloads.*

5.3.3 Garbage Collection

FlashVM uses *merged* and *dummy* discards to optimize garbage collection of free VM pages on flash. We compare the performance of garbage collection for FlashVM against native Linux VM on an SSD. Because Linux cannot currently execute discards, we instead collect block-level I/O traces of paging traffic for different applications. The block layer breaks down the VM discard I/O requests into 128 KB discard commands, and we emulate FlashVM by merging multiple discard requests or replacing them with equivalent dummy discard operations as described in Section 4.3. Finally, we replay the processed traces on an aged trim-capable OCZ-Vertex SSD and record the total trace execution time.
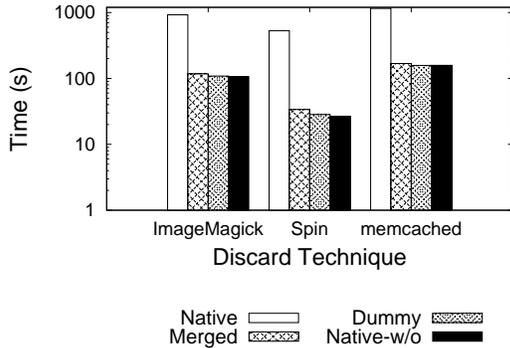

Figure 7: **Garbage Collection Performance:** *Impact of merged and dummy discards on application performance for FlashVM. y-axis is log-scale for application execution time.*

Figure 7 compares the performance of four different systems: FlashVM with merged discards over 100 MB ranges, FlashVM with dummy discards, native Linux VM with discard support and baseline Linux VM without discard support. Linux with discard is 12 times slower than the baseline system, indicating the high cost for inefficient use of the discard command. In contrast,

Workload	DiskVM		FlashVM	
			Const Mem	Const Runtime
	Runtime	Mem	Rel. Runtime	Rel. Memory
ImageMagick	207	814	31%	51%
Spin	209	795	11%	16%
SpecJBB	275	710	6%	19%
memcached-store	396	706	18%	60%
memcached-lookup	257	837	23%	50%

Table 4: **Cost/Benefit Analysis:** *FlashVM analysis for different memory-intensive application workloads. Systems compared are DiskVM with disk-backed VM, a FlashVM system with the same memory (Const Mem) and one with the same performance but less memory (Const Runtime). FlashVM results show the execution time and memory usage, both relative to DiskVM. Application execution time Runtime is in seconds; memory usage Mem is in megabytes.*

FlashVM with merged discard, which also has the reliability benefits of Linux with discard, is only 15 *percent* slower than baseline. With the addition of adaptive dummy discards, which reduces the rate of discards when page clusters are rapidly allocated, performance is 11% slower than baseline. In all cases, the slowdown is due to the long latency of discard operations, which have little direct performance benefit. These results demonstrate that naive use of discard greatly degrades performance, while FlashVM's merged and dummy discard achieve similar reliability benefits at performance near native speeds.

5.4 FlashVM Application Performance

Adoption of FlashVM is fundamentally an economic decision: a FlashVM system can perform better than a DiskVM system even when it is provisioned with more expensive DRAM. Therefore, we evaluate the performance gains and memory savings when replacing disk with flash for paging. Our results reflect estimates for absolute memory savings in megabytes.

Table 4 presents the performance and memory usage of five application workloads on three systems:

1. *DiskVM* with 1 GB memory and a dedicated disk for swapping;
2. *FlashVM - Const Mem* with the same DRAM size as DiskVM, but improved performance;
3. *FlashVM - Const Runtime* with reduced DRAM size, but same performance as DiskVM.

Our analysis in Table 4 represents configurations that correspond to the three data points shown in Figure 1. System configurations for workloads with high locality or unused memory do not page and show no benefit from FlashVM. Similarly, those with no locality or extreme memory requirements lie on the far left in Figure 1 and perform so poorly as to be unusable. Such data points are not useful for analyzing virtual memory performance. The column in Table 4 titled DiskVM shows the execution time and memory usage of the five workloads on a system swapping to disk. Under FlashVM - Const Mem and FlashVM - Const Runtime, we show the percentage reduction in the execution time and memory usage respectively, both when compared to DiskVM. The reduction in memory usage corresponds to the potential price savings by swapping to flash rather than disk for achieving similar performance.

For all applications, a FlashVM system outperforms a system configured with the same amount of DRAM and disk-backed VM (FlashVM - Const Mem against DiskVM). FlashVM's reduction in execution time varies from 69% for ImageMagick to 94% for the modified SpecJBB, a 3-16x speedup. On average, FlashVM reduces run time by 82% over DiskVM. Similarly, we find that there is a potential 60% reduction in the amount of DRAM required on the FlashVM system to achieve similar performance as DiskVM (FlashVM - Const Runtime against DiskVM). This benefit comes directly from the lower access latency and higher bandwidth of flash, and results in both price and power savings for the FlashVM system.

Overall, we find that applications with poor locality have higher memory savings because the memory saved does not substantially increase their page fault rate. In contrast, applications with good locality see proportionally more page faults from each lost memory page. Furthermore, applications also benefit differently depending on their access patterns. For example, when storing objects, *memcached* server performance improves 5x on a FlashVM system with the same memory size, but only 4.3x for a lookup workload. The memory savings differ similarly.

6   Related Work

The FlashVM design draws on past work investigating the use of solid-state memory for storage. We categorize this work into the following four classes:

Persistent Storage. Flash has most commonly been proposed as a storage system to replace disks. eNVy presented a storage system that placed flash on the memory bus with a special controller equipped with a battery-backed SRAM buffer [33]. File systems, such as YAFFS and JFFS2 [27], manage flash to hide block erase latencies and perform wear-leveling to handle bad blocks. More recently, TxFlash exposes a novel transactional interface to use flash memory by exploiting its copy-on-write nature [25]. These systems all treat flash as persistent storage, similar to a file system. In contrast, FlashVM largely ignores the non-volatile aspect of flash and instead focuses on the design of a high-performance, reliable and scalable virtual memory.

Hybrid Systems. Guided by the price and performance of flash, hybrid systems propose flash as a second-level cache between memory and disk. FlashCache uses flash as secondary file/buffer cache to provide a larger caching tier than DRAM [10]. Windows and Solaris can use USB flash drives and solid-state disks as read-optimized disk caches managed by the file system [2, 7]. All these systems treat flash as a cache of the contents on a disk and mainly exploit its performance benefits. In contrast, FlashVM treats flash as a backing store for evicted pages, accelerates both read and write operations, and provides mechanisms for improving flash reliability and efficiency of garbage collection by using the semantic information about paging only available within the OS.

Non-volatile Memory. NAND flash is the only memory technology after DRAM that has become cheap and ubiquitous in the last few decades. Other non-volatile storage class memory technologies like phase-change memory (PCM) and magneto-resistive memory (MRAM) are expected to come at par with DRAM prices by 2015 [21]. Recent proposals have advocated the use of PCM as a first-level memory placed on the memory bus alongside DRAM [11, 19]. In contrast, FlashVM adopts cheap NAND flash and incorporates it as swap space rather than memory directly addressable by user-mode programs.

Virtual Memory. Past proposals on using flash as virtual memory focused on new page-replacement schemes [23] or providing compiler-assisted, energy-efficient swap space for embedded systems [13, 22]. In contrast, FlashVM seeks more OS control for memory management on flash, while addressing three major problems for paging to dedicated flash. Further, we present the first description of the usage of the discard command on a real flash device and provide mechanisms to optimize the performance of garbage collection.

7   Conclusions

FlashVM adapts the Linux virtual memory system for the performance, reliability, and garbage collection char-

acteristics of flash storage. In examining Linux, we find many dependencies on the performance characteristics of disks, as in the case of prefetching only adjacent pages. While the assumptions about disk performance are not made explicit, they permeate the design, particularly regarding batching of requests to reduce seek latencies and to amortize the cost of I/O. As new storage technologies with yet different performance characteristics and challenges become available, such as memristors and phase-change memory, it will be important to revisit both operating system and application designs.

Acknowledgments

References

[1] AGRAWAL, N., PRABHAKARAN, V., WOBBER, T., DAVIS, J., MANASSE, M., AND PANIGRAHY, R. Design tradeoffs for ssd performance. In *USENIX* (2008).

[2] ARCHER, T. MSDN Blog: Microsoft Ready-Boost. http://blogs.msdn.com/tomarcher/archive/2006/06/02/615199.aspx.

[3] BUGNION, E., DEVINE, S., AND ROSENBLUM, M. Disco: Running commodity operating systems on scalable multiprocessors. In *SOSP* (1997).

[4] DOTNETPERLS.COM. Memory benchmark for Web Browsers., December 2009. http://dotnetperls.com/chrome-memory.

[5] DRAMEXCHANGE.COM. Mlc nand flash price quote, Dec. 2009. http://dramexchange.com/#flash.

[6] GEAR6. Scalable hybrid memcached solutions. http://www.gear6.com.

[7] GREGG, B. Sun Blog: Solaris L2ARC Cache. http://blogs.sun.com/brendan/entry/test.

[8] GUPTA, D., LEE, S., VRABLE, M., SAVAGE, S., SNOEREN, A. C., VARGHESE, G., VOELKER, G. M., AND VAHDAT, A. Difference engine: Harnessing memory redundancy in virtual machines. In *OSDI* (2008).

[9] IYER, S., AND DRUSCHEL, P. Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous IO. In *SOSP* (2001).

[10] KGIL, T., AND MUDGE, T. N. Flashcache: A nand flash memory file cache for low power web servers. In *CASES* (2006).

[11] LEE, B. C., IPEK, E., MUTLU, O., AND BURGER, D. Architecting phase change memory as a scalable DRAM alternative. In *ISCA* (2009).

[12] LEVY, H. M., AND LIPMAN, P. H. Virtual memory management in the VAX/VMS operating system. *Computer 15*, 3 (1982), 35–41.

[13] LI, H.-L., YANG, C.-L., AND TSENG, H.-W. Energy-aware flash memory management in virtual memory system. In *IEEE Transactions on VLSI systems* (2008).

[14] LINUX-DOCUMENTATION. SCSI Generic Driver Interface. http://tldp.org/HOWTO/SCSI-Generic-HOWTO.

[15] LORD, M. Author, Linux IDE Subsystem, Personal Communication. December, 2009.

[16] LORD, M. hdparm 9.27: get/set hard disk parameters. http://linux.die.net/man/8/hdparm.

[17] MEMCACHED. High-performance Memory Object Cache. http://www.danga.com/memcached.

[18] MIELKE, N., MARQUART, T., KESSENICH, J. N. W., BELGAL, H., SCHARES, E., TRIVEDI, F., GOODNESS, E., NEVILL, E., AND L.R. Bit error rate in nand flash memories. In *IEEE IRPS* (2008).

[19] MOGUL, J. C., ARGOLLO, E., SHAH, M., AND FARABOSCHI, P. Operating system support for nvm+dram hybrid main memory. In *HotOS* (2009).

[20] MSDN BLOG. Trim Support for Windows 7. http://blogs.msdn.com/e7/archive/2009/05/05/support-and-q-a-for-solid-state-drives-and.aspx.

[21] OBJECTIVE-ANALYSIS.COM. White Paper: PCM becomes a reality., Aug. 2009. http://www.objective-analysis.com/uploads/2009-08-03_Objective_Analysis_PCM_White_Paper.pdf.

[22] PARK, C., LIM, J., KWON, K., LEE, J., AND MIN, S. L. Compiler-assisted demand paging for embedded systems with flash memory. In *EMSOFT* (2004).

[23] PARK, S., JUNG, D., KANG, J., KIM, J., AND LEE, J. CFLRU: A replacement algorithm for flash memory. In *CASES* (2006).

[24] POLTE, M., SIMSA, J., AND GIBSON, G. Enabling enterprise solid state disks performance. In *WISH* (2009).

[25] PRABHAKARAN, V., RODEHEFFER, T., AND ZHOU, L. Transactional flash. In *OSDI* (2008).

[26] RAJIMWALE, A., PRABHAKARAN, V., AND DAVIS, J. D. Block management in solid-state devices. In *USENIX* (2009).

[27] REDHAT INC. JFFS2: The Journalling Flash File System, version 2, 2003. http://sources.redhat.com/jffs2.

[28] ROSENBLUM, M., AND OUSTERHOUT, J. K. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems 10*, 1 (1992).

[29] SAXENA, M., AND SWIFT, M. M. FlashVM: Revisiting the Virtual Memory Hierarchy. In *HotOS-XII* (2009).

[30] SHU, F., AND OBR, N. Data Set Management Commands Proposal for ATA8-ACS2, 2007. http://t13.org/Documents/UploadedDocuments/docs2008/e07154r6-Data_Set_Management_Proposal_for_ATA-ACS2.doc.

[31] SPIN. LTL Model Checker, Bell Labs. http://spinroot.com.

[32] STREETPRICES.COM. Disk drive price quote, Dec. 2009. http://www.streetprices.com.

[33] WU, M., AND ZWAENEPOEL, W. eNVy: A non-volatile, main memory storage system. In *ASPLOS-VI* (1994).