Synergistic Coupling of SSD and Hard Disk for QoS-aware Virtual Memory

Ke Liu[†] Xuechen Zhang[†‡*] Kei Davis[♮] Song Jiang[†]

[†]ECE Department
Wayne State University
Detroit, MI 48202, USA

[‡] School of Computer Science
Georgia Institute of Technology
Atlanta, GA 30332, USA

[♮] CCS Division
Los Alamos National Laboratory
Los Alamos, NM 87545, USA

***Abstract*—With significant advantages in capacity, power consumption, and price, solid state disk (SSD) has good potential to be employed as an extension of DRAM (memory), such that applications with large working sets could run efficiently on a modestly configured system. While initial results reported in recent works show promising prospects for this use of SSD by incorporating it into the management of virtual memory, frequent writes from write-intensive programs could quickly wear out SSD, making the idea less practical. We propose a scheme, *HybridSwap*, that integrates a hard disk with an SSD for virtual memory management, synergistically achieving the advantages of both. In addition, HybridSwap can constrain performance loss caused by swapping according to user-specified QoS requirements.**

To minimize writes to the SSD without undue performance loss, HybridSwap sequentially swaps a set of pages of virtual memory to the hard disk if they are expected to be read together. Using a history of page access patterns HybridSwap dynamically creates an out-of-memory virtual memory page layout on the swap space spanning the SSD and hard disk such that random reads are served by SSD and sequential reads are asynchronously served by the hard disk with high efficiency. In practice HybridSwap can effectively exploit the aggregate bandwidth of the two devices to accelerate page swapping.

We have implemented HybridSwap in a recent Linux kernel, version 2.6.35.7. Our evaluation with representative benchmarks, such as Memcached for key-value store, and scientific programs from the ALGLIB cross-platform numerical analysis and data processing library, shows that the number of writes to SSD can be reduced by 40% with the system's performance comparable to that with pure SSD swapping, and can satisfy a swapping-related QoS requirement as long as the I/O resource is sufficient.

***Keywords*-SSD; Page Swapping; and Flash Endurance.**

I. Introduction

Given the large performance gap between DRAM (memory) and hard disk, flash memory (flash)—with much lower access latency, and performance much less sensitive to random access, than disk—has been widely employed to accelerate access of data normally stored on disk, or by itself to provide a fast storage system. Another use of flash is as an extension of memory so that working sets of memory-intensive programs can overflow into the additional memory space [24], [4], [16], [19], [27]. This latter application is motivated by the exponential cost of DRAM as a function of density (e.g., ∼\$10/GB for 2GB DIMM, ∼\$200/GB for 8GB DIMM, and hundreds or even thousands of dollars per GB for DIMM over 64GB) and its high power consumption. In contrast, flash is much less expensive, has much greater density, and is much more energy efficient both in terms of energy consumption and needed cooling. A typical approach for inclusion of flash in the memory system is to use it as swap space and map virtual memory onto it [24], [4], [16]. Much effort has been made to address the particular properties of flash for use in this context, such as access at page granularity, the erasure-before-write requirement, and asymmetric read and write performance. The most challenging issue is flash's endurance—the limited number of write-erase cycles each flash cell can endure before failure. A block of MLC (multi-level cell) NAND flash—the less expensive, higher density, and more widely used type—may only be erased and rewritten approximately 5000-10,000 times before it has unacceptably high bit error rates [10], [18]. Though SLC (single-level cell) flash can support a higher erasure limit (around 100,000), increasing flash density can reduce this limit [11].

When flash-based SSD is used for file storage the limited number of write cycles that flash can endure may not be a significant issue because frequently accessed data is usually buffered in memory. However, if SSD is used as a memory extension with memory-intensive applications, the write rate to the SSD can be much higher and SSD longevity a serious concern. For example, for a program writing to its working set on a 64GB SSD of MLC flash at a sustained rate of 128MB/s, the SSD can become unreliable after about two months of the use. Considering the write-amplification effect due to garbage collection at the flash translation layer (FTL), the lifetime could be reduced by a factor of up to 1000 [13]. To improve SSD's lifetime when used as swap space for processes' virtual memory, researchers have tried to reduce write traffic to SSD by using techniques such as minimizing the replacement of dirty pages [24], detecting pages containing all zero bytes to avoid swapping them [24], and managing swap space at the granularity of custom objects [4]. In these prior works, when SSD is proposed to serve as swap space, the conventional host, the hard disk, is excluded from consideration for use for the same purpose. This is in contrast to the scenario where SSD is used as file storage, wherein hard disk is often also used to provide large capacity and to reduce cost.

We claim that it is feasible to distribute write traffic to the swap space between SSD and hard disk to obtain SSD-like performance but with greatly reduced wear on the SSD, and therefore that hard disk can augment SSD to provide a

*Xuechen Zhang conducted this work at Wayne State University.

highly-performing and reliable memory extension. Though the hard disk's access latency is greater than SSD's by orders of magnitude, its peak throughput, which is achieved with long sequential access, can be comparable to that of SSD. As long as there are memory pages that are sequentially swapped out and back in, the hard disk can play a role in the implementation of such a system, in particular exploiting its significant advantages in cost and longevity. Furthermore, using pre-swapping-out and pre-swapping-in techniques, access of swap space can be served asynchronously with the occurrence of page faults, and by having the swap space span both SSD and disk it is possible to access them in parallel to increase the bandwidth of the swapping system. Using disk for page swapping is a particularly economical choice when a disk is already present. We note that it is less likely for disk to be involved in file-data access when applications are actively computing on their prepared working sets and are actively page swapping. While endurance remains an issue in anticipated storage-class memory (SCM) [8], our approach to keeping the hard disk relevant in the effort towards reliable, large, and affordable memory provides a low-cost and effective supplement to existing solutions [7], [20], [28].

We propose a page-swapping management scheme, *HybridSwap*, to reduce SSD writes in its use as a memory extension. In this scheme we track page access patterns to identify pages of strong spatial locality to form page sequences and accordingly determine the destinations of pages to be swapped out.

In summary we make the following contributions.

- We introduce the hard disk into the use of SSD for memory extension. We show that representative memory-intensive applications have substantial sequential access that warrants the use of disk to significantly reduce SSD writes without significant performance loss compared to SSD-only swapping.
- We develop an efficient algorithm to record memory access history, and to identify page access sequences and evaluate their locality. Swapping destinations are accordingly determined for the sequences to ensure that both high disk throughput and low SSD latency are exploited while high latency is avoided or hidden.
- We build a QoS-assurance mechanism into HybridSwap so that it can bound the performance penalty due to swapping. Specifically, it allows users to specify a bound on the program stall time due to page faults as a percentage of the program's total run time.
- We implement the hybrid swapping system in the Linux kernel and conduct extensive evaluation using representative benchmarks including key-value store for in-memory caching, image processing, and scientific computations. The results show that HybridSwap can reduce SSD swapping writes by 40% with performance comparable to that of using an SSD-only solution.

II. Related Work

There are numerous works in the literature concerned with various aspects of SSD. We briefly review the most closely related efforts in reducing write cycles in the use of SSD for virtual memory, and in the integration of SSD and hard disk for storage systems.

In the FlashVM system SSD is used as a dedicated page-swapping space for its greater cost-effectiveness than adding DRAM [24]. To reduce writes to SSD, FlashVM does not swap pages if they are all zeros. It also prioritizes the replacement of clean pages over dirty pages. In another work using SSD as swap space, SSDAlloc, data for swapping is managed at object granularity rather than at page granularity, where objects may be much smaller then the page size [4]. Objects are defined by programmers via the *ssd_alloc()* API for dynamically allocating memory. SSDAlloc also attempts to track the access patterns of objects to exploit temporal locality in the replacement of the objects. Mogul et al. propose to reduce writes to non-volatile memory (NVM), such as flash used with DRAM in a hybrid memory, by estimating time-to-next-write (TTNW) for pages managed by the operating system, and placing pages with large TTNW on the NVM [19]. Ko et al., recognizing that current OS swapping strategies are designed for hard disk and can cause excessive block copies and erasures on SSD if it is used as swap space, modify the swapping strategies of Linux by swapping out pages in a log structure and swapping in pages in a block-aligned manner [16].

Because SSD is still used as a swap device in addition to hard disk in our proposed scheme, the optimizations of the use of SSD in the mentioned works are complementary or supplemental to our effort to improve SSD lifetime. In some of the works the Linux virtual memory prefetching strategy is tuned to match SSD's characteristics to improve swapping efficiency, such as allowing non-sequential prefetching and realigning prefetching scope with SSD block boundaries [24], [16]. In constrast, HybridSwap coordinates prefetching over SSD and disk to hide disk access latency.

As an accelerator for hard disk, SSD has been used either as buffer cache between main (DRAM) memory and the hard disk and exploits workloads' locality for data caching [26], [22], or used in parallel with the hard disk to form a hybrid storage device such that frequently accessed data is stored on the SSD [6], [21], [29]. A major effort in these works for optimized performance and improved SSD lifetime is in dynamic identification of randomly-read blocks and caching them on, or migrating them to, the SSD. In principle HybridSwap has a similar goal of directing sequential page access to the disk. However, unlike accessing file data in a storage system, HybridSwap manages the swapping of virtual memory pages and has different opportunities and challenges. First, the placement of swapped-out pages on the swap space is determined by the swapping system rather than by the file system. For continuously swapped-out pages HybridSwap can usually manage to sequentially write to the disk. There is an opportunity to improve the efficiency of reading from the disk

in resolving page faults by placing the pages on the disk in an order consistent with their anticipated future read (page fault) sequence. To this end HybridSwap predicts future read sequences at the time of swapping out pages. Second, because data in a file system is structured, information is available to assist in the prediction of access patterns: for example, metadata and small files are more likely to be randomly accessed. Virtual memory pages lack such information and their access patterns can be expensive to detect. Third, swap space can be more frequently accessed than files because it is mapped into process address space, and data migration strategies used in works concerned with hybrid disks adapting to changing access patterns are usually not efficient in this scenario. To meet these challenges HybridSwap incorporates new and effective methods for tracking access patterns and the laying out of swap pages in the swap space.

In the domain of file I/O there are two recent works in which the hard disk is explicitly used to reduce writes to the SSD. To allow disk to be a write cache for access to the SSD, Soundararajan et al. analyzed file I/O traces in the desktop and server environments and found that there is significant write locality—most writes are on a small percentage of file blocks and writes to a block are concentrated in a short time window. Based on this observation they cache these blocks on the disk during its write period and migrate them to the SSD when the blocks start to be read [9]. In contrast, swap pages do not have such locality because a page's access following a swap-out, if ever, must be a read (swap-in). Therefore HybridSwap must take into account the efficiency of reading pages from the disk. The second work, I-CASH, makes the assumption that writes to a file block usually do not significantly change its contents, i.e., that the difference in content (the delta) is usually small [23]. By storing the deltas on the hard disk the SSD will mostly serve reads, and writes to the disk are made more efficient as deltas are compacted into disk blocks. However, for virtual memory access there is insufficient evidence to support the assumption that the deltas are consistently small. In addition, the on-line computation required for producing the deltas and recovering the original pages could heavily burden the CPU. In contrast, HybridSwap achieves high read efficiency from the disk by forming sequential read patterns to exploit high disk throughput.

To provision QoS assurance for programs running in virtual memory, a common practice is to prevent the memory regions of performance-sensitive programs from swapping, either enforced by the kernel [5] or facilitated with application-level paging techniques [12]. In the HPC environment swapping (or virtual memory) is often disabled (or not supported) to ensure predictable and efficient execution of parallel programs on a large cluster [27]. While HybridSwap advocates the use of hard disk along with SSD as a swapping device, its integrated QoS assurance mechanism provides a safety measure to prevent excessive performance loss in the effort to reduce wear on flash.

III. Design and Implementation

When a combination of SSD and hard disk is used to host swap space as part of virtual memory, the performance goal is to efficiently resolve page faults, i.e., to read pages from the swap space quickly. While HybridSwap is proposed to achieve a reduction in writes to SSD, an equally important goal is high efficiency for reading swapped pages. To achieve both goals HybridSwap is designed to carefully select appropriate pages to be swapped to the disk and to schedule prefetching of swapped-out pages back into memory. To this end we need to integrate spatial locality with the traditional consideration of temporal locality in the selection of pages for swapping, evaluate access spatial locality, and schedule the swapping of pages in and out.

A. Integration of Temporal Locality and Spatial Locality

Because swap space is on a device slower than DRAM, temporal locality in page access maintained by keeping the most frequently accessed pages in memory. To achieve this the operating system buffers physical pages that have been mapped to virtual memory in the system page cache and tracks their access history. A replacement policy is used to select pages with weakest temporal locality as targets for swapping out. Consecutively identified target pages are swapped together and are highly likely to be contiguously written in a region of the swap space. However, there is no assurance that these pages will be swapped in together in the future. Furthermore, the replacement policy may not identify pages that were contiguously swapped in as candidates for contiguously swapping out with sequential writes. Ignoring spatial locality can increase swap-in latency and page-fault penalty, especially when the swap space is on disk. In the context of HybridSwap, spatial locality refers to the phenomenon that contiguous pages on the swap space are the targets of page faults occurring together and can be swapped in together. While swap-in efficiency is critical for the effectiveness of hard disk as a swapping medium, spatial locality is integrated with temporal locality when HybridSwap swaps pages out to the hard disk. To this end, among pages of weak temporal locality we identify candidate pages with potentially strong spatial locality and evaluate their temporal locality according to their access history. Only sequences of pages with weak temporal locality and strong spatial locality will be swapped to the hard disk, and those of weak spatial locality will be swapped to the SSD.

In the LRU (least recently used) replacement algorithm, one of the more commonly used replacement algorithms in operating systems, it is relatively easy to recognize sequences of candidate pages. HybridSwap is prototyped in the Linux 2.6 kernel that adopts an LRU variant similar to the 2Q replacement [15]. Here the kernel pages are grouped into two LRU lists, an active list to store recently or frequently accessed pages, and an inactive list to store pages that have not been accessed for some time. A faulted-in page is placed at the tail of the active list, and pages at the tail of the inactive list are considered to have weak temporal locality and are candidates

for replacement. A page is promoted from the inactive list into the active list if it is accessed, and demoted to the inactive list if it has not been accessed for some time. Pages that have been accessed together when they are added into the inactive list will stay close in the lists. However, a sequence of pages in the lists may belong to different processes, as a page fault of one process leads to scheduling of another process. Such a sequence is unlikely to repeat as the involved processes usually do not coordinate their relative progress. Therefore HybridSwap groups pages at the tail of the inactive list according to their process IDs, and then evaluates their spatial locality within each process.

B. Evaluation of Spatial Locality of Page Sequences

For a sequence of pages at the tail of the inactive list we need to predict the probability of them being swapped in together if they are swapped out. As the basis of this prediction we check the page access history to determine whether the same access sequence has appeared before. The challenge is how to efficiently detect and record page accesses. One option is to use *mprotect()* to protect the pages of each process to detect page access when an *mprotect*-triggered page fault occurs, but this could be overly expensive because the system may not always conduct page swapping, and not all pages are constantly involved in the swapping. Instead, HybridSwap records a page access only when a page fault occurs. In this way there is almost zero time cost for detecting page accesses, and the space overhead for recording access is proportional to number of faulted pages. Assuming that programs have relatively stable access patterns and that the replacement policy can consistently identify pages of weak locality for swapping, the access history recorded in this manner is sufficient to evaluate the spatial locality of the sequence of pages to be swapped out.

When a sequence of pages is swapped out together and sequentially written to the disk, they will be swapped in together—or sequentially prefetched into the memory—in response to a fault on any page in the sequence. However, sequential disk access does not necessarily indicate efficient swapping because for efficiency the prefetched pages must be used to resolve future page faults before being evicted from memory. In other words, the spatial locality for a sequence of prefetched pages is characterized by how close in time they are used to resolve page faults. Quantitatively the locality is measured by the time gap between any two fault occurrences on the pages in the sequence. Ideally it is not larger than the lifetime of a swapped-in page, or the time period from its swap-in to its subsequent swap-out.

In evaluating locality there are three goals in effectively recording page access. First, the space overhead should be small. Second, it must be possible to determine in $O(1)$ time whether a page sequence has appeared before. Third, from the history information it should be possible to predict whether future faults on the pages in the sequence would occur *together*. To achieve these goals, for each process we build a table, its *access table*, that is the same as the process's page table, except that (1) only pages that have had faults are in the table; and (2) the leaf node of the tree-like table, equivalent to the PTE (page table entry) in the Linux page table, is used to store the times when page faults occur on its corresponding page. We set a global clock that is incremented whenever a page fault occurs in the system. For the page associated with the fault we record the current clock time in the page's access table entry. We also record a page's most recent swap-in time to obtain the page's most recent in-memory lifetime. When a page is swapped out its most recent lifetime is calculated as the difference between the current clock time and its swap-in time. We compute a moving average of the lifetimes of any swapped pages in the system and use it as a threshold to evaluate a page sequence's spatial locality. The average L_k is updated after serving the kth request by $L_k = (1-\alpha) * L_{k-1} + \alpha * Lifetime$, where $Lifetime$ is the lifetime calculated for the most recently swapped out page, and α is between 0 and 1 and is used to control how quickly history information decays. In our reported experiments $\alpha = 2/3$ so that more recent lifetimes are better represented. Our experiments show that system performance is not sensitive to this parameter over a large range. In the prototype we use 32 bits to represent a time. (This allows a process to swap 16TB in its lifetime before counter wrap, assuming 4KB pages. If this were an issue a 64-bit value could be used.) The space overhead for storing timestamps is then 32 bits/4KB, or 1B/1KB, a very modest 0.1% of the virtual memory involved in page faults.

Only sequences of high spatial locality are eligible to be swapped to disk. When there is a candidate sequence of pages selected from the tail of the inactive list that belong to the same process we use the process's access table to determine whether the sequence has sufficiently high spatial locality. If the difference between any two pages' access times is greater than the system's current average lifetime, or there are anonymous pages in the sequence without history access times, the sequence's spatial locality is deemed low.

C. Scheduling Page Swapping

There are three steps for swapping out pages in HybridSwap: selecting a candidate sequence, evaluating its spatial locality, and determining swapping destinations. For each swap we select a process and remove all of its pages from the last N pages of the inactive list, where N is a parameter representing the tradeoff between temporal locality and spatial locality. A smaller N will ensure that only truly least recently used pages are swapped but provides less opportunity for producing long page sequences. In contrast, overly large N can benefit disk efficiency but may lead to the replacement of recently used pages. With today's memory sizes the list can be relatively long, so N can be large enough for high disk efficiency without compromising the system's temporal locality. In current Linux kernels eight pages are replaced in each swap. Previous research has suggested that disk access latency can be well amortized with requests of 256KB or larger [25], so N is set to $p * 128KB/s$, where p is the number of processes with pages in the tail area of the inactive list and s is the page size

(4KB). Sequence selection is rotated among the processes for fair use of memory.

Spatial locality is evaluated for each selected candidate sequence. If the result indicates weak locality the sequence is swapped to SSD. Otherwise, in principle the sequence will be sent to the disk. A sequence that is written to the disk is recorded in the corresponding process's access table using a linked list embedded in the leaf nodes of the table recording their virtual addresses. When a fault occurs on a page in the sequence the first page of the sequence will be read in synchronously, then the other pages of the sequence will be asynchronously prefetched from the disk into memory.

While asynchronous prefetching of pages is expected to allow their page faults to be resolved in memory, one deficiency in the swapping-in operation is the long access latency experienced by the read of the first page. To hide this latency, for a sequence intended for swapping to the disk we examine its pages' history access times recorded in the access table to see whether they had been accessed in a consistent order. If so we order them in a list of ascending access times and divide them into two segments. The first segment, the SSD segment, will be swapped out to the SSD, and the second one containing pages of longer access times, the disk segment, will be swapped out to the disk. If there is a fault on the page in the SSD segment both the SSD and disk segments are immediately prefetched. The objective of this scheduling is to hide the long disk access latency behind the time for the process to do computation on the data from the SSD. To determine the length of the SSD segment we need to compare the data consumption rate of the process to the disk access latency. To achieve this we track the time periods between any two intermediate page faults for each process, and calculate their moving average T_{compt} with a formula similar to the one used for calculating system average lifetime. We also track the access latencies associated with each disk swap-in, and calculate their moving average $T_{disk-latency}$, also with a similar formula. If $T_{disk-latency}/T_{compt}$ is smaller than the sequence size then it is the SSD segment size. Otherwise, the entire sequence will be swapped to the SSD. Thus overly-short sequences are swapped to the SSD even if they have strong spatial locality. We do not need to explicitly evaluate prefetch accuracy: inaccurate prefetching is due to changing access pattern, and inconsistent patterns recorded in the access table will automatically cancel future swapping of the involved pages to the disk.

D. Building QoS Assurance into HybridSwap

HybridSwap is designed to judiciously select pages for swapping to the disk to attain a good balance between reducing SSD writes and retaining the performance advantage of SSD. At the same time HybridSwap allows a user to specify the prioritization of these two goals to influence how the tradeoff is made for specific programs. We use the ratio of program stall time due to page faults and its run time as the input. This ratio is a mandatory upper bound on the cost of swapping on a program's run time. To implement this HybridSwap dynamically maintains the ratio of current total stall time and program execution time.

If no QoS requirement is specified page swapping will be managed in the default manner. Otherwise, HybridSwap tracks the ratio of the current ratio and the required ratio. If this ratio, which we call the shift ratio, is larger than 1 we need to shift subsequent swapping-page sequences towards the SSD. Initially a sequence's SSD segment is obtained as described in Section III.C. If the shift ratio is larger than 1 the SSD segment size is increased by this ratio (up to the entire sequence size). If the shift ratio is still larger than 1 with almost all recent pages swapped to SSD, the sequence is shifted towards memory in a similar manner. Pages to be kept in memory are simply skipped when selecting pages for swapping out. If the shift ratio is consistently smaller than 1 by a certain margin, HybridSwap will shift sequences towards SSD or the disk. To effectively enforce the required ratio, HybridSwap initially places pages on the SSD when a program starts to run.

IV. Performance Evaluation and Analysis

HybridSwap is implemented primarily in the Linux kernel (2.6.35.7) and supports a user-level QoS tool. Most code modifications are in the memory management system, for example in the *handle_pte_fault()* function to record both major and minor page faults in the access tables, and in the *shrink_page_list()* function to select sequences for spatial-locality evaluation and swapping out. At user level we use a script to read the user-specified QoS requirement from the command line. This tool is only needed for users requesting a QoS requirement. We experimentally answer the following questions.

- Can the number of I/O writes to SSD be significantly reduced by using the hard disk?
- Can HybridSwap achieve performance comparable to using SSD alone?
- Is HybridSwap effective for workloads with mixed memory access patterns or for very heavy workloads?
- What is the overhead of HybridSwap?

A. Experimental Setup

We conducted experiments on a Dell Poweredge server with an Intel 2.4GHz dual-core processor and 4GB DRAM. Except where otherwise stated we limited the memory available for running processes to 1GB to intensify swapping and better reveal performance differences between different swapping strategies. The server is configured with a 160GB hard disk (WDC WD1602 ABKS). We used two types of SSD devices in the experiments, Intel SSDSA2M080G2GC, referred to as Intel SSD, and OCZ-ONYX, referred to as OCZ SSD, with performance characteristics summarized in Table I. Except where otherwise stated the OCZ SSD is used for the experiments. NCQ is enabled on the disk and SSDs. Per standard practice we used CFQ [2] as the disk scheduler and NOOP [3] as the SSD scheduler. We compared HybridSwap with the swapping configuration using only an SSD, referred to as *SSD-Swap*. Linux supports swapping on multiple storage devices

	Intel SSD	OCZ SSD	Hard Disk
Capacity (GB)	80	32	160
Sequential Read (MB/s)	238	140	85
Sequential Write (MB/s)	125	110	70
Random Read (MB/s)	25	34	4
Random Write (MB/s)	10	4	0.8

Table I
CAPACITIES AND SEQUENTIAL/RANDOM READ/WRITE THROUGHPUT OF TWO SSDS AND ONE HARD DISK. 4KB REQUESTS ARE USED.

	Memcached(5)	Image(2)	CC(6)	Matrix(7)
SSD-Swap	269,618	450,352	911,148	471,201
HybridSwap	170,316	273,477	712,660	396,291
Reduction Ratio	37%	40%	22%	16%

Table II
NUMBER OF WRITES TO THE SSD WITH AND WITHOUT USE OF THE HARD DISK MANAGED BY HYBRIDSWAP. THE NUMBER OF CONCURRENT INSTANCES IS GIVEN IN PARENTHESES.

	Memcached (packets/s)	Image(s)	CC(s)	Matrix(s)
SSD-Swap	16,734	108	431	864
HybridSwap	16,597	103	400	860
Improv. Ratio	-0.8%	4.6%	7.1%	0.5%

Table III
PERFORMANCE OF THE BENCHMARKS WHEN EITHER SSD-SWAP OR HYBRIDSWAP IS USED. MEMCACHED'S PERFORMANCE IS IN TERMS OF THROUGHPUT (QUERIES/SEC), AND THE OTHERS IN RUNTIME (SEC)

with a pre-configured swapping traffic distribution. To compare HybridSwap with a less-optimized hybrid swapping system, we also used an SSD and a hard disk with 1:1 distribution as a Linux-managed swapping space, referred to as *RAID-Swap*. In RAID-Swap the swap space is equally striped on the SSD and the disk, giving a data layout similar to RAID 0.

B. Benchmarks

We used four real-world memory-intensive benchmarks with differing memory access patterns to form the workloads: Memcached, ImageMagick, matrix inverse, and correlation computation.

Memcached provides a high-performance distributed caching infrastructure to form an in-memory key-value cache [17]. We set up a Memcached client to issue requests for storing (*PUT*) and retrieving (*GET*) data at the Memcached server with different dispatch rates and key distributions.

ImageMagick (*Image* for short) is a software package providing command-line functionality for image editing [14]. In the experiments we enlarge a file of 17MB by 200%, and convert the file from its original JPG format to PNG format.

Matrix Inverse (*Matrix* for short) is a scientific computation program from ALGLIB, an open-source and cross-platform numerical analysis and data processing library [1]. The implementation of matrix inverse in ALGLIB employs several optimizations, such as efficient use of CPU cache, to achieve maximal performance. The input matrix size is 4096*4096 in the experiments.

Correlation Computation (*CC* for short) is also from ALGLIB. It finds the statistical dependence between two matrices by calculating their correlation coefficient. The input matrix size is 4096*4096 in the experiments.

C. Reduction of SSD Writes

A major goal of HybridSwap is to select appropriate pages for swapping to disk to reduce writes to the SSD. We measured the number of page writes for each of the four benchmarks, running multiple concurrent instances to increase the aggregate memory demand and create more complicated dynamic memory access patterns. Table II lists the total number of page writes on the SSD for each of the four benchmarks when only SSD is used and when both SSD and disk are used and managed by HybridSwap.

For Memcached we ran five concurrent instances. To simulate the use of the caching service by applications running on a client, we assume each instance has its own set of data items for constant *PUT*s and *GET*s in a certain time period before operating on another set of items. Item size is uniformly distributed between 64KB and 256KB. The total item size is 1.95GB, for 0.95GB data on the swap space and the rest in the 1GB main memory. Memcached is pre-populated using *PUT*s, then the client simultaneously issues queries to different data sets in different Memcached instances, with each set receiving 500 queries at a time. HybridSwap detects relatively strong spatial locality in memory accesses when the same set of data items is accessed together more than once, and reduces writes to the SSD by 37%. Because only sets of data items of strong locality are placed on disk, and the sequential disk bandwidth is comparable to the non-sequential bandwidth of the SSD, we observed only 0.8% slowdown in the rate of query transfer between Memcached and the client (Table III).

We ran two concurrent instances of Image, which has a sequential memory access pattern and requires 2.4GB memory. An interesting observation is that not only is the number of I/O writes reduced by 40% (Table II), but the run time of the benchmark is also reduced by 4.6% (Table III). One reason is that aggressive prefetching by HybridSwap based on sequences on the hard disk reduces the total number of major faults from 53,557 to 37,929 (29% reduction), and transforms the remaining faults to minor faults (hits on prefetched pages). A second is that the SSD and hard disk can concurrently serve requests with higher aggregate I/O bandwidth than SSD alone.

We ran six instances of CC, which requires 3.4GB memory. The results show that 22% of writes to SSD are eliminated by HybridSwap due to its detection of access sequences with strong locality and their efficient swapping to disk. However, there is a greater fraction of writes to SSD than for Image, which is explained by the sequences generated by the two benchmarks. Figure 1 shows the cumulative distributions of their sequence sizes. For CC and Image, 87% and 75% of the sequences have fewer than 30 pages, respectively, that is, CC has more shorter sequences. With short sequences, even if they are qualified to be swapped to the disk, pages in their SSD segments can be a larger proportion of the sequences and

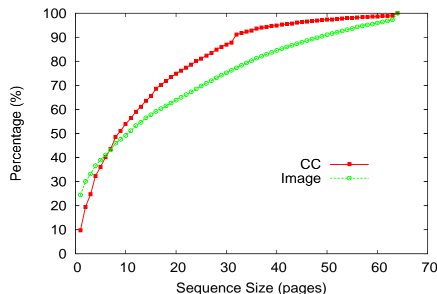


Figure 1. Cumulative distributions of sequence sizes for *Image* and *CC*.

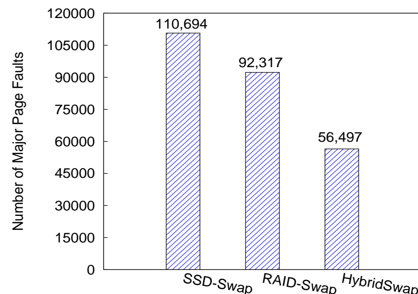


Figure 2. Number of major faults when running the CC benchmark with SSD-Swap, RAID-Swap, and HybridSwap. For SSD-Swap and RAID-Swap we use the Linux default read-ahead policy.

are stored on the SSD. Accordingly, the relative performance of CC is greater than that of Image (Table III).

Because the Matrix benchmark has small memory demand (260MB) we ran seven instances to stress the swap devices. Though it produces a significant number of short sequences, HybridSwap can still exploit the detected locality to carry out disk-based swapping without compromising performance, with writes to SSD reduced by 16%. Our measurements show that the swapped-out data is distributed on the SSD and hard disk at a ratio of 5:1 when the swap space reaches its maximal size.

In the experiments presented in the next several sections we select only one or a subset of the benchmarks that is most appropriate for revealing specific aspects of HybridSwap.

D. Effectiveness of Sequence-based Prefetching

HybridSwap records access history to identify sequences of strong locality for swapping to disk and to enable effective prefetching afterward. Lacking information with which to predict page faults, Linux conservatively sets a limit of eight pages for each prefetch. In contrast, HybridSwap can prefetch as many as 64 pages in one swapping-in. In this section we compare the number of major page faults associated with HybridSwap, SSD-Swap, and RAID-Swap when running CC. As shown in Figure 2, HybridSwap reduces page faults by 49% and 39% compared to SSD-Swap and RAID-Swap, respectively. By exploiting consistent page access patterns such as row-based, column-based, and diagonal-based access, HybridSwap can detect a large number of long sequences of strong spatial locality. There are two factors contributing to HybridSwap's small major-page-fault count. One is the high I/O efficiency in its page swap-in, which helps make pages available in memory before they are requested to resolve faults. The other is the accuracy of its prefetching, which makes the swapped-in pages become the targets of minor page faults. Using only one device, SSD-Swap has the most major faults.

E. Effect of Memory Size

Memory size is inversely correlated to the amount of data swapped out of the memory so we can vary the memory size to influence the swapping intensity and correspondingly affect the relative performance characteristics of HybridSwap. In the following experiments we ran Memcached with different amounts of memory to determine whether the program's relative throughput is substantially affected by a

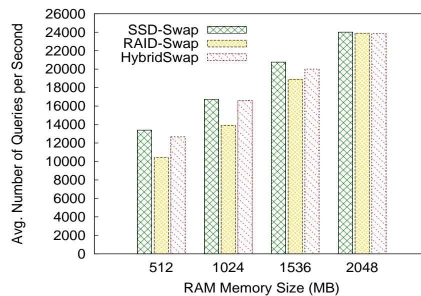


Figure 3. Memcached throughput in terms of average number of queries per second with different memory sizes and different swapping schemes.

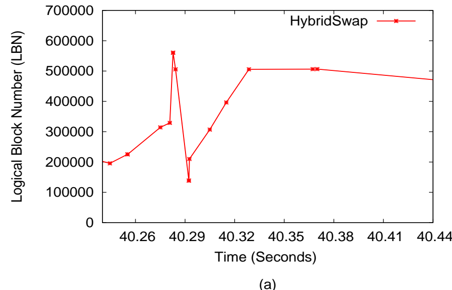
swapping-intensive workload. We increase the memory size from 512MB, 1024MB, 1536MB, to 2048MB. As shown in Figure 3, when memory size is small the throughput of Memcached in terms of average number of queries served per second is higher for SSD-Swap than for HybridSwap. However, the loss in throughput is small—only 5.5%—and is accompanied by a 40% reduction of writes to SSD. Compared to RAID-Swap, HybridSwap's throughput is higher by 22% because Linux does not form sequences that enable effective prefetching. As the memory size increases more accesses can be served in memory, resulting in correspondingly increasing throughput. When the memory size is 2GB, 98% of the programs' working set is in memory and there is little difference between the throughputs of the three schemes.

F. Insights into HybridSwap Performance

Next we investigate how the relative performance of HybridSwap and SSD-Swap changes with differing SSD devices, and why the performance of HybridSwap is much higher even though both use the same combination of SSD and disk. We ran Image and measured its run time and total number of writes to each SSD type using each of the three swapping schemes, as shown in Table IV. Table I shows that the difference in throughput of the two SSD devices ranges from approximately 15% to 150%. However, the relative performance shown in both programs' runtimes and number of major page faults is minimally affected by the SSD types. The reason is that in HybridSwap SSD is mainly used for random access and the

	SSD-Swap	RAID-Swap	HybridSwap
OCZ	108/53557	144/64160	103/37929
Intel	104/58459	143/62457	104/35218

Table IV
RUN TIME/NUMBER OF MAJOR FAULTS DURING THE EXECUTIONS OF THE IMAGE BENCHMARK WITH DIFFERENT SWAPPING SCHEMES AND DIFFERENT TYPES OF SSDS.



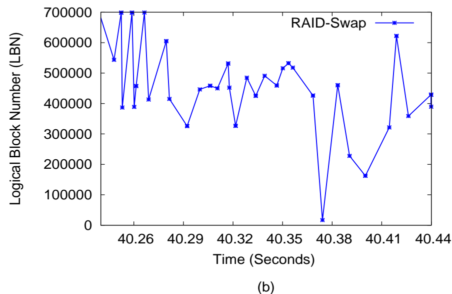


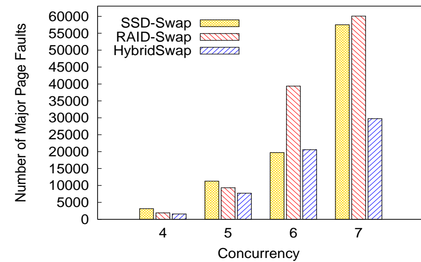
Figure 4. Disk addresses, in terms LBNs of data access on the hard disk in a sampled execution period with HybridSwap (a) and RAID-Swap (b).

disk is used to serve sequential access, and the random access performance of both SSDs is much greater than disk.

Table IV also shows that HybridSwap consistently and significantly outperforms RAID-Swap by reducing both writes to SSD and runtime for both types of SSD devices. This is a consequence of different access patterns on the disk. Figures 4(a) and 4(b) show access addresses on the hard disk in terms of disk LBN (logical block number) for HybridSwap and RAID-Swap, in a sampled execution period of 0.2 seconds. The lines connecting consecutively accessed blocks are indicative of disk head movement. When RAID-Swap is used the disk head frequently moves in a disk area whose LBNs range from 30 to 700,000 indicating low I/O efficiency and high page-fault service time. When HybridSwap is used pages in the same sequence are written and read sequentially. This shows that HybridSwap's performance, which is comparable to that of SSD-Swap and is much higher than that of RAID-Swap, should be mainly attributed to its improved disk efficiency.

G. Multiple-program Concurrency

Concurrently running multiple programs could potentially lead to swapping of a large number of random pages and reduced disk efficiency. We use the Matrix benchmark to evaluate the performance of HybridSwap with varying concurrency



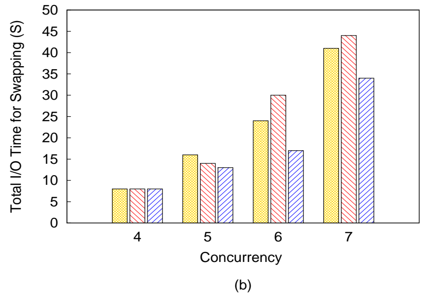


Figure 5. Major faults (a) and total I/O time (b) spent on the swapping in the running of the Matrix benchmark with varying degrees of concurrency.

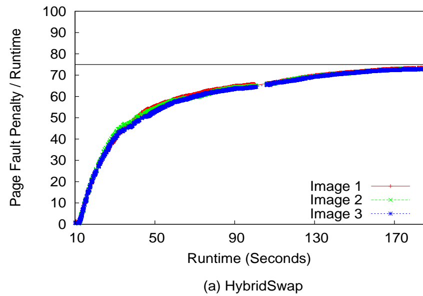
from 4, 5, 6, to 7 program instances.

With four concurrent instances there are only 0.2GB of data to be swapped out and the I/O time spent on faults accounts for only 1.7% of total execution time. As the concurrency increases the total number of page faults increases by 32 times (Figure 5(a)), and I/O time for swapping increases by 4.5 times (Figure 5(b)). From the figures we make three observations: (1) When the system has a swapping-intensive workload, RAID-Swap produces many more major page faults and spends much more I/O time on swapping; (2) SSD-Swap cannot sustain performance for workloads with highly intensive swapping because the bandwidth of one SSD can be overwhelmed; and, (3) HybridSwap maintains a small major fault count and low swapping time even with very high concurrency. This suggests that a heavy workload does not prevent HybridSwap from forming sequences and efficiently using the disk for swapping.

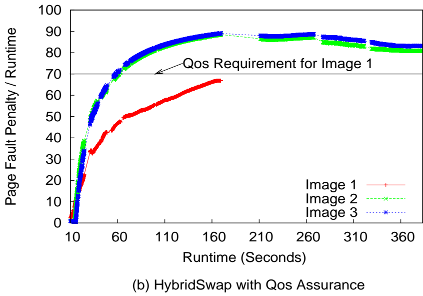
When HybridSwap is used 49% of page faults are eliminated by effective swap-page prefetching based on faults history. Because pages are read and written sequentially on the disk and from the two devices in parallel, HybridSwap achieves even better performance than SSD-Swap. Total swap I/O time is reduced by 20% and 43% compared to SSD-Swap and RAID-Swap, respectively.

H. Bounding the Page Fault Penalty

While swapping data to the storage devices, especially hard disk, can severely extend a program's run time, HybridSwap provides a means to bound the page fault penalty on run time. To demonstrate this QoS control we run three instances of Image. Figure 6(a) shows that the ratio of the aggregate page fault penalty and the runtime, as a function of elapsed run time, without any QoS requirement. Because no instance has



(a) HybridSwap



(b) HybridSwap with Qos Assurance

Figure 6. Running three instances of Image with HybridSwap when a QoS requirement in terms of a bound on the ratio of the aggregate page fault penalty and the runtime is not specified (a) or is specified (b).

a bound on the ratio, all three make best effort in the use of virtual memory and their respective ratio's curves are nearly identical. Initially the ratios rise as each builds up its respective working set and page faults increase as more of their working sets are swapped. In the curves there are segments that are not continuous because we only sample the ratio at each swap-out to reduce the overhead. We next attempt to set a bound on the ratio for one or more of the instances to prioritize their performance. Because the memory size is limited, which is smaller than one instance's working set, an instance prioritized with a tighter bound would move some or all of its working set that would be placed on the hard disk without a QoS requirement to the SSD. Figure 6(b) shows the ratios of the instances when we set a bound of 70% on the ratio of Image instance 1. By enforcing the QoS requirement HybridSwap does keep this instance's ratio below the bound. Concurrently, the other two instances' ratios approach 90% and execution times are significantly extended. We note that Instance 1's execution time is reduced by only 10% and a tighter bound on the instance may not be realized because the use of the hard disk has already been highly optimized in HybridSwap and the relative performance advantage of the SSD is limited. We expect that a faster SSD (or SSD RAID) or larger memory can more effectively reveal the QoS control.

I. Runtime Overhead Analysis

The runtime overhead of HybridSwap has two major components: the time for checking the access tables to determine the spatial locality of candidate sequences and the time for searching access tables for sequences to be prefetched when major page faults occur. Even though these operations are on

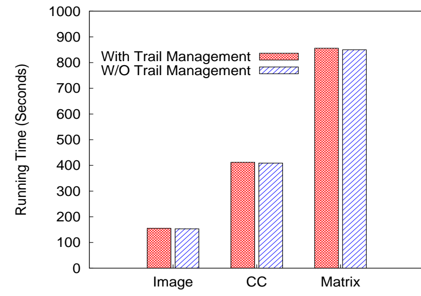


Figure 7. Run times of the Image, CC, and Matrix benchmarks with and without sequences-related operations invoked.

critical I/O time, the time overhead is inconsequential because the time for operations on the in-memory data structure is at least two orders of magnitude less than disk latency and even SSD latency. In this section we quantitatively analyze the overhead of HybridSwap. Instead of directly measuring the overhead we measure the increase in program runtime attributable to the sequence-related operations. More specifically, we measure the runtimes of three selected benchmarks (Image, CC, and Matrix) with RAID-Swap and compare them to the runtimes with RAID-Swap with the function module for these operations added. In the latter case, we ensure that the operations be carried out as they do in HybridSwap. The results are shown in Figure 7. The runtime overhead is less than 1% on average for all the benchmarks.

V. Summary and Conclusions

We propose using a hard disk in addition to SSD to support virtual memory in hosting swap space subject to two constraints: significant write traffic to the swap space should be directed to the disk to improve SSD lifetime, and the performance of the hybrid swapping system should not be significantly less than that of an SSD-only system.

Our design of such a system ensured that (1) temporal locality is effectively exploited so that the memory is fully utilized; (2) spatial locality is effectively exploited so that the disk does not become a performance bottleneck; (3) the swapping of pages to the SSD and to the hard disk is scheduled such that the SSD is used only to serve page faults when it can achieve better performance than the disk; and, (4) the use of SSD and hard disk is coordinated so that the throughput potential of the disk is realized but its access latency is avoided.

We implemented the proposed HybridSwap scheme in Linux and experimentally compared HybridSwap to an SSD-only solution and to an SSD/disk array as swap devices in Linux using applications with diverse memory access patterns. Our evaluation shows that HybridSwap can reduce writes to the SSD by up to 40% with the system's performance comparable to that with pure SSD swapping.

Acknowledgment

This work was supported by US National Science Foundation under CAREER CCF 0845711, CNS 1117772, and CNS 1217948. This work was also funded in part by the Accelerated

Strategic Computing program of the Department of Energy. Los Alamos National Laboratory is operated by Los Alamos National Security LLC for the US Department of Energy under contract DE-AC52-06NA25396.

REFERENCES

[1] Alglib, a Cross-platform Numerical Analysis and Data Processing Library, 2012. http://www.alglib.net/.

[2] J. Axboe. Completely Fair Queueing (CFQ) Scheduler, 2010. http://en.wikipedia.org/wiki/CFQ.

[3] J. Axboe. Noop Scheduler, 2010. http://en.wikipedia.org/wiki/Noop.

[4] A. Badam and V. S. Pai. SSDAlloc: Hybrid SSD/RAM Memory Management Made Easy. In *the 8th USENIX Symposium on Networked Systems Design and Implementation*. USENIX, 2011.

[5] A. T. Campbell. A Quality of Service Architecture. In *Ph.D Thesis, Computing Department, Lancaster University*, 1996.

[6] F. Chen, D. Koufaty, and X. Zhang. Hystor: Making the Best Use of Solid State Drives in High Performance Storage Systems. In *International Conference on Supercomputing*, 2011.

[7] M. M. Franceschini and L. Lastras-Montano. Improving Read Performance of Phase Change Memories via Write Cancellation and Write Pausing. In *the 16th IEEE International Symposium on High Performance Computer Architecture*, 2010.

[8] R. F. Freitas and W. W. Wilcke. Storage-class Memory: The Next Storage System Technology. In *IBM Journal of Research and Development*, 52, 2008.

[9] M. B. G. Soundararajan, V. Prabhakaran and T. Wobber. Extending SSD Lifetimes with Disk-based Write Caches. In *the 8th USENIX Conference on File and Storage Technologies*. USENIX, 2010.

[10] L. M. Grupp, A. M. Caulfield, J. Coburn, S. Swanson, E. Yaakobi, P. H. Siegel, and J. K. Wolf. Characterizing Flash Memory: Anomalies, Observations and Applications. In *the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE/ACM, 2009.

[11] L. M. Grupp, J. D. Davis, and S. Swanson. The Bleak Future of NAND Flash Memory. In *the USENIX Conference on File and Storage Technologies*. USENIX, 2012.

[12] S. M. Hand. Self-paging in the Nemesis Operating System. In *the Third Symposium on Operating Systems Design and Implementation*, 1999.

[13] X.-Y. Hu, E. Eleftheriou, R. Haas, I. Iliadis, and R. Pletka. Write Amplification Analysis in Flash-based Solid State Drives. In *SYSTOR'09: The Israeli Experimental Systems Conference*, 2009.

[14] Imagemagick, 2012. http://www.imagemagick.org/script/index.php.

[15] T. Johnson and D. Shasha. 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm. In *International Conference on Very Large Data Bases*, 1994.

[16] S. Ko, S. Jun, Y. Ryu, O. Kwon, and K. Koh. A New Linux Swap System for Flash Memory Storage Devices. In *International Conference on Computational Sciences and its Applications*, 2008.

[17] Memcached, 2011. http:/memcached.org/.

[18] N. Mielke, T. Marquart, N. Wu, J. Kessenich, H. Belgal, E. Schares, F. Trivedi, E. Goodness, and L. R. Nevill. Bit Error Rate in NAND Flash Memories. In *IEEE International Reliability Physics Symposium*, 2008.

[19] J. C. Mogul, E. Argollo, M. Shah, and P. Faraboschi. Operating System Support for NVM+DRAM Hybrid Main Memory. In *the 12th Conference on Hot Topics in Operating Systems*, 2009.

[20] M.Qureshi, J. Karidis, M. Franceschini, V. Srinivasan, L. Lastras, and B. Abali. Enhancing Lifetime and Security of PCM-based Main Memory with Start-gap Wear Levelings. In *the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE/ACM, 2009.

[21] H. Payer, M. A. Sanvido, Z. Bandic, and C. M. Kirsch. Combo Drive: Optimizing Cost and Performance in a Heterogeneous Storage Device. In *the 1st Workshop on Integrating Solid-state Memory into the Storage Hierarchy*, 2009.

[22] T. Pritchett and M. Thottethodi. Sievestore: A Highly-selective, Ensembel-level Disk Cache for Cost-performance. In *Proceeding of 37th International Symposium on Computer Architecture*. ACM, 2010.

[23] J. Ren and Q. Yang. I-Cash: Intelligently Coupled Array of SSD and HDD. In *the 17th IEEE Symposium on High Performance Computer Architecture*, 2011.

[24] M. Saxena and M. M. Swift. FlashVM: Virtual Memory Management on Flash. In *Proceeding of the 2010 USENIX Annual Technical Conference*. USENIX, 2010.

[25] S. W. Schlosser, J. Schindler, S. Papadomanolakis, M. Shao, A. Ailamaki, C. Faloutsos, and G. R. Ganger. On Multidimensional Data and Modern Disks. In *the 4th USENIX Conference on File and Storage Technologies*. USENIX, 2005.

[26] M. Srinivasan and P. Saab. Flashcache: a General Purpose Writeback Block Cache for Linux, 2011. https://github.com/facebook/flashcache.

[27] C. Wang, S. Vazhkudai, X. Ma, F. Meng, Y. Kim, and C. Eagelmann. NVMalloc: Exposing an Aggregate SSD Store as a Memory Partition in Extreme-scale Machines. In *The 26th IEEE International Parallel and Distributed Processing Symposium*, 2012.

[28] P. Zhou, B. Zhao, J. Yang, and Y. Zhang. A Durable and Energy Efficient Main Memory using Phase Change Memory Technology. In *the 36th International Symposium on Computer Architecture*, 2009.

[29] X. Zhang, K. Davis, and S. Jiang. iTransformer: Using SSD to Improve Disk Scheduling for High-performance I/O. In *the 26th IEEE International Parallel and Distributed Processing Symposium*, 2012.