STORAGE SYSTEMS

FOR NON-VOLATILE MEMORY DEVICES

A Dissertation

by

XIAOJIAN WU

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

August 2011

Major Subject: Computer Engineering

STORAGE SYSTEMS

FOR NON-VOLATILE MEMORY DEVICES

A Dissertation

by

XIAOJIAN WU

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Approved by:

Chair of Committee,	A.L.Narasimha Reddy
Committee Members,	Riccardo Bettati
	Serap Savari
	Srinivas Shakkottai
Head of Department,	Costas N. Georghiades

August 2011

Major Subject: Computer Engineering

ABSTRACT

Storage Systems

for Non-volatile Memory Devices. (August 2011)

Xiaojian Wu, Ph.D., Texas A&M University

Chair of Advisory Committee: A.L.Narasimha Reddy

This dissertation presents some research work on how to use non-volatile memory devices to build storage systems. There are many types of non-volatile memory devices, and they usually have better performance than regular magnetic hard disks in terms of both throughput and latency. This dissertation focuses on two of them, NAND flash device and Phase Change Memory (PCM). This work consists of two parts.

The first part is to design a high-performance hybrid storage system employing Solid State Drives that are build out of NAND flash devices and Hard Disk Drives. In this hybrid system, we propose two different policies to improve its performance. One is to exploit the fact that the performances of Solid State Drive and Hard Disk Drive are asymmetric and the other is to exploit concurrency on multi devices. We implement the prototype in Linux and evaluate both policies in multiple workloads and multiple configurations. The results show that the proposed approaches improve the performance significantly, and adapt to different configurations of the system under different workloads.

The second part is to implement a file system on a special class of memory devices, Storage Class Memory (SCM), which is both byte addressable and also non-volatile, e.g. PCM. We claim that both the existing regular file systems and the memory based file systems are not suitable for SCM, and propose a new file system, called SCMFS, which is implemented on the virtual address space. In SCMFS, we utilize

the existing memory management module in the operating system to do the block management and keep the space always contiguous for each file. The simplicity of SCMFS not only makes it easy to implement, but also improves the performance. We implement the prototype of SCMFS in Linux and evaluate its performance through multiple benchmarks.

To Kelsey Wu

ACKNOWLEDGMENTS

Place your acknowledgment within these braces.

TABLE OF CONTENTS

CHAPTER Page

LIST OF TABLES

LIST OF FIGURES

FIGURE Page

FIGURE Page

CHAPTER I

INTRODUCTION

There are many kinds of non-volatile memory devices, such as EEPROM, Flash, ferroelectric RAM, Magnetoresistive RAM, memristor, Phase Change Memory(PCM). This dissertation presents our research work on how to build storage systems on two of them, NAND flash and PCM.

To build storage systems, we have several choices of the architecture layers. We can implement it on the file system level or the block device level. Whatever layer we implement it on, we need to consider the compatibility to the existing applications, since the external interfaces always change much slower than the internal ones. For example, on the block device level our system should export generic block device interfaces to the operating system, and on the file system level our system should behaves as a normal directory based file system. Besides the compatibility, our main consideration in this dissertation is to improve the performance, including both throughput and latency. The reliability is also taken into consideration in the system design.

A. NAND Flash

NAND flash memory device has some special characteristics including good random read performance, no support for "in-place" updates, and limits of erasure times. Based on these characteristics, researchers have proposed several special file systems for it. However these file systems are not built on the normal generic block layer as regular file systems, and their scalability is relatively bad. These file systems are usually used in the embedded systems. Another popular way to use NAND flash is Solid State Drive(SSD). SSD is a type of storage device build on NAND flash memory. Inside SSD, there is a layer called Flash Translation Layer (FTL)

that emulates harddisk interfaces on flash memory. Thus, users can run normal file systems on SSDs. The functionalities such as block mapping, wear leveling, garbage collection are implemented inside the layer FTL.

Our research work on NAND flash is to incorporate SSD into a hybrid storage system, instead of to build systems on NAND flash directly. SSD has better performance as well as higher price than magnetic disk. Our hybrid storage system utilizes SSD devices to boost the storage performance. In such systems, there are two different straightforward approaches. One is to use SSD device as the cache of the HD device. This approach has some disadvantages. First, using the cache mechanism, the hot data cached in the SSD device are duplicated, and the capacity of the whole storage system is equal to that of the HD device. However, the capacity of SSD is too large to be ignorable. Using cache mechanism will waste the capacity. Second, using cache mechanism, the performance of the storage system will be always worse than that of SSD device. Actually, when most of the workload is directed to the SSD device, the bandwidth of HD device is wasted. We should be able to improve the throughput by balancing some workload to HD device. The other straightforward approach is to use stripping, like that in RAID technology. Different from RAID, we can put more hot data on SSD than HD to improve the whole performance. This approach is not adaptive.

In this dissertation, we present a hybrid storage system and propose two approaches to improve the performance. Both approaches are measurement driven and adaptive. In the first approach, we utilize the fact that SSD and HD devices exhibit different performance characteristics of read and write behavior. SSD performs much better than HD on random read while worse on random write. In this approach, we will monitor the performance of the both devices and access patterns of data. We use these information to migrate data between the devices. For example, we can

move read-intensive data to SSD device and write-intensive data to HD. The other approach exploits concurrency on multiple devices and improves both throughput and latency simultaneously. Our experiments show that both approaches can improve the performance significantly. We present these two approaches in Chapter II and III respectively.

B. Phase Change Memory

As a newly emerging memory device, Phase Change Memory (PCM) interests a lot of researchers. Since PCM has really good scalability, one usage of PCM is to build large capacity main memory device. A lot of papers have been published on this area. In this dissertation, we only consider how to use PCM as a persistent storage device instead of a main memory device.

To use PCM as a persistent storage device, the most straightforward way is to use Ramdisk to emulate a disk device on the PCM device and run the regular file systems on it, such as Ext2Fs, Ext3Fs, etc.. In this approach, the file systems access the storage devices through generic block layer and the emulated block I/O operations. However, the overhead caused by the emulation and the generic block layer is not necessary, and we can reduce it by designing a file system specially for memory devices. When storage devices are attached to the memory bus, both the storage device and the main memory will share system resources such as the bandwidth of the memory bus, the CPU cache and the TLB, and the overhead of file systems will impact the performance of the whole system. File systems for PCM should consider these factors.

In this dissertation, we present a new file system - SCMFS, which is specifically designed for Storage Class Memory(SCM). SCM is defined as a special class of mem-

ory devices that are both byte addressable and non-volatile, such as PCM. SCMFS exports the identical interfaces as the regular file systems do, and is compatible with all the existing applications. In this file system, we minimize the CPU overhead of file system operations. We build SCMFS on virtual memory space and utilize the memory management unit (MMU) to map the file system address to physical addresses on SCM. The layouts in both physical and virtual address spaces are very simple. We also keep the space contiguous for each file in SCMFS to simplify the process of handling read/write requests in the file system. The results based on multiple benchmarks show that the simplicity of SCMFS makes it easy to implement and improves the performance. We present SCMFS in Chapter IV.

CHAPTER II

EXPLOIT DEVICE ASYMMETRIES IN A FLASH AND DISK HYBRID STORAGE SYSTEM

We consider the problem of efficiently managing storage space in a hybrid storage system employing flash and disk drives in this chapter. The flash and disk drives exhibit different performance characteristics of read and write behavior. In this chapter, we present a technique for balancing the workload properties across flash and disk drives in such a hybrid storage system. We consider various alternatives for managing the storage space in such a hybrid system and show that the proposed technique improves performance in diverse scenarios. This approach automatically and transparently manages migration of data blocks among flash and disk drives based on their access patterns. This work has been published in [45].

A. Background

Novel storage devices based on flash memory are becoming available with price/performance characteristics different from traditional magnetic disks. Many manufacturers have started building laptops with these devices. While these devices may be too expensive (at this time) for building larger flash-only storage systems, storage systems incorporating both flash-memory based devices and magnetic disks are becoming available.

Traditionally, storage systems and file systems have been designed considering the characteristics of magnetic disks such as the seek and rotational delays. Data placement, retrieval, scheduling and buffer management algorithms have been designed to take these characteristics into account. When both flash and magnetic disk drives are employed in a single hybrid storage system, a number of these policies may need to be revisited.

Flash based storage devices exhibit different characteristics than magnetic disk drives. For example, writes to flash devices can take longer than magnetic disk drives while reads can finish faster. The flash drives have no seek and rotational latency penalties unlike their magnetic counterparts, but have a limit on the number of times a block can be written. Flash drives also typically have more uniform performance (especially for reads) depending on file size, where magnetic disks typically perform better with larger file sizes. The write performance of flash drives can experience much larger differences in peak to average completion times due to block re-mapping done in the Flash Translation Layer (FTL) and necessary erase cycles to free up garbage blocks. These differences in characteristics need to be taken into account in hybrid storage systems in efficiently managing and utilizing the available storage space.

Fig 1 shows the time taken to read, write a data block from a disk drive and two flash drives considered in this study. As can be seen from the data, the flash and disk drives have different performance for reads and writes. First, flash drive is much more efficient than the magnetic disk for small reads. While flash drive read performance increases with the read request size, the magnetic disk performance improves considerably faster and at larger request sizes, surpasses the performance of the flash devices. Small writes have nearly the same performance at both the devices. As the write request size grows, the magnetic disk provides considerably better performance than the flash device. the disk drive is more efficient for larger reads and writes. These characteristics are observed for a 250GB, 7200 RPM Samsung SATA magnetic disk (SP2504C), a 16GB Transcend SSD (TS16GSSD25S-S) and a 32GB MemoRight GT flash drives. While different devices may exhibit different performance numbers, similar trends are observed in other drives.

As can be seen from this data, requests experience different performance at

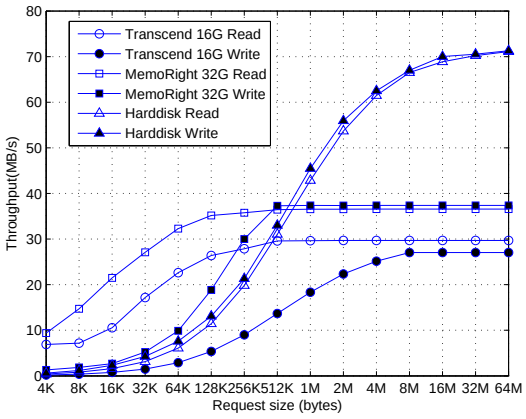


Fig. 1. Read, write performance of different devices

different devices based on the request type (read or write) and based on the request size (small or large). This work is motivated at managing storage space to maximize performance in a hybrid system that utilizes such devices together.

Various methods have been proposed to compensate and exploit diversity in device characteristics. Most storage systems use memory for caching and manage memory and storage devices together. Most of these approaches deal with devices of different speeds, storing frequently or recently used data on the faster devices. However, the flash and disk storage devices have asymmetric read/write access characteristics, based on request sizes and whether the requests are reads or writes. This asymmetry makes this problem of managing the different devices challenging and interesting in a flash+disk hybrid system. In order to accommodate these characteristics, in this work, we treat the flash and disk devices as peers and not as two levels in a storage hierarchy.

Our approach is contrasted with traditional approach of a storage hierarchy in Figure 2. In a traditional storage hierarchy, hot data is moved to the faster (smaller) device and cold data is moved to the larger (slower) device. In our approach, the cold

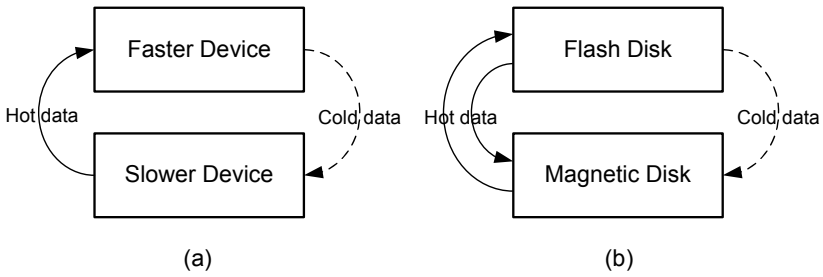


Fig. 2. (a)Migration paths in traditional hybrid storage systems. (b)Migration paths in space management layer.

data is still moved to the larger device. However, the hot data may be stored in either device because of the performance asymmetry. For example, hot data that is mostly read may be moved to the Flash device while large files that are mostly written may be moved to the disk drive.

Flash technology is advancing rapidly and device performance characteristics differ from manufacturer to manufacturer, from one generation to the next. This warrants that a solution to managing the space across the devices in a hybrid system should be adaptable to changing device characteristics.

We consider two related issues for managing space across flash and disk drives in a hybrid system. The first issue is allocation. Where should data be allocated? Should we first allocate data on flash before starting to use the disk drive? Should we employ striping across the two devices until space constraints force us to do something else? Clearly, these decisions will have implications on load on different devices, their utilization and the performance of the storage system.

A second issue we consider is this: given a block allocation, can we adapt the storage system to better utilize the devices in a hybrid system? We approach this problem through data redistribution or migration of data from one device to another device to match the data access characteristics with the device characteristics. For example, if a data block has a higher number of reads compared to writes, it may

be better suited to flash device and moving it from its currently allocated disk drive to flash drive may improve its performance. Similarly, if a file and its blocks are sequentially written and read, it may be better located on the magnetic disk without any loss in performance (compared to the flash device) while preserving space on the smaller flash device for other blocks. In this chapter, we focus on the problem of data redistribution (through migration), in storage systems built out of flash and magnetic drives.

We propose a measurement-driven approach to migration to address these issues. In our approach, we observe the access characteristics of individual blocks and consider migrating individual blocks, if necessary, to another device whose characteristics may better match the block access patterns. Our approach, since it is measurement driven, can easily adapt to different devices and changing workloads or access patterns.

In this chapter, we study the two related issues of allocation and data migration among the flash and disk drives to manage these diverse devices efficiently in a hybrid system. We make the following significant contributions in this work: (a) we propose a technique for managing data efficiently in a hybrid system through dynamic, measurement-driven data migration between flash and disk drives, (b) we study the impact of different allocation decisions in such a system, (c) we show that the performance gains possible through flash devices may be strongly contingent on the workload characteristics observed at the storage system and (d) that flexible block management in a hybrid system is possible through a demonstration of results from a Linux-based prototype.

B. Measurement-driven Migration

Our general approach is to pool the storage space across flash and disk drives and make it appear like a single larger device to the file system and other layers above. We manage the space across the different devices underneath, transparent to the file system. We allow a file block on one device to be potentially moved later to another device that better matches the access patterns of that data block. We also allow blocks to move from one device to another device as workloads change in order to better utilize the devices.

In order to allow blocks to be flexibly assigned to different devices, an indirection map, containing mappings of logical to physical addresses, needs to be maintained. Every read and write request is processed after consulting the indirection map and determining the actual physical location of data. Similar structures have been used by others [1, 11, 12]. In our system, as the data migrates from one device to another device, there is a need to keep track of the re-mapping of the block addresses. When data is migrated, the indirection map needs to be updated. This is an additional cost of migration. We factor this cost into the design, implementation and evaluation of our scheme. In order to reduce the costs of maintaining this map and updates to this map, we consider migration at a unit larger than a typical page size. We consider migration of data in chunks or blocks of size 64KB or larger.

We keep track of access behavior of a block by maintaining two counters, one measuring the read and the other write accesses. These counters are a "soft" state of the block i.e., the loss of this data is not critical and affects only the performance, but not the correctness of data accesses. This state can be either maintained in memory or on disk depending on the implementation. If memory is employed, we could maintain a cache of recently accessed blocks for which we maintain this state, in order to limit

memory consumption. It is also possible to occasionally flush this data to disk to maintain this read/write access history over longer time. In our implementation, we maintain this information in memory through a hash table. This table is only as large as the set of recent blocks whose access behavior we choose to remember and is not proportional to the total number of data blocks in the system. We employ two bytes for keeping track of read/write frequency separately per chunk (64Kbytes or higher). This translates into an overhead of 0.003% or lower or about 32KB per 1GB of storage. We limit the memory consumption of this data structure by de-allocating information on older blocks periodically.

We employ the read/write access counters in determining a good location for serving a block. A block, after receiving a configured minimum number of accesses, can be considered for migration or relocation. This is to ensure that sufficient access history is observed before the block is relocated. We explain below how a relocation decision is made. If we decide to relocate or migrate the block, the read/write counters are reset after migration in order to observe the access history since the block is allocated on the new device.

We took a general approach to managing the device characteristics. Instead of characterizing the devices statically, we keep track of device performance dynamically. Every time a request is served by the device, we keep track of the request response time at that device. We maintain both read and write performance separately since the read, write characteristics can be substantially different as observed earlier. Dynamically measured performance metrics also allow us to account for load disparities at different devices. It also allows a single technique to deal with diverse set of devices without worrying about any configuration parameters.

Each time a request is sent to the device, a sample of the device performance is obtained. We maintain an exponential average of the device performance by comput-

ing average response time = 0.99 * previous average + 0.01 * current sample. Such an approach is used extensively in networking, in measuring round trip times and queue lengths etc [24]. Such exponential averaging smooths out temporary spikes in performance while allowing longer term trends to reflect in the performance measure. For each device i, we keep track of the read r_i and write w_i response times. We consider all request sizes at the device in computing this average response time to reflect the workload at that device in our performance metric. As we direct different types of request blocks to different devices, the workloads can be potentially different at different devices.

Given a block j's read/write access history through its access counters R_j and W_j and the device response times, we use the following methodology to determine if a block should be moved to a new location. The current average cost of accessing this block j in its current device i, C_{ji}= $(R_j * r_i + W_j * w_i)/(R_j + W_j)$. The cost of accessing a block with similar access patterns at another device k, C_{jk} (computed similarly using the response times of device k) are compared. If $C_{ji} > (1+\delta) * C_{jk}$, we will consider this block to be a candidate for migration, where $0 < \delta$ is a configurable parameter. We experimented with different values of δ in this study. A larger value for δ demands a greater performance advantage before moving a block from one device to another device.

In general, when there are several devices which could provide better performance to a block, a number of factors such as the load on the devices, storage space on the devices and the cost of migration etc. can be considered for choosing one among these devices.

Migration could be carried out in real-time while normal I/O requests are being served and during quiescent periods when the devices are not very active. The migration decision could be made on a block by block basis or based on looking at all the

blocks collectively at once through optimization techniques. Individual migration and collective optimization techniques are complementary and both could be employed in the same system. We primarily focus on individual block migration during normal I/O workload execution in this chapter. Later, we plan to incorporate a collective optimization technique that could be employed, for example, once a day.

There are many considerations to be taken into account before a block is migrated. The act of migration increases the load on the devices. This could affect device performance. Hence, it is necessary to control the rate of migration. Second, a fast migration rate, may result in moving data back and forth, causing oscillations in workloads and performance at different devices. In order to control the rate of migration, we employ a token scheme. The tokens are generated at a predetermined rate. Migration is considered only when a token becomes available. In our current implementation, we experiment with a conservative, static rate of generating tokens.

When a block is migrated from one device i to another device k, the potential cost of this migration could be $r_i + w_k$, r_i for reading the block from device i and w_k for writing the block to its new location on the device k. In order to reduce the costs of migration, we only consider blocks that are currently being read or written to the device, as part of normal I/O activity. For these blocks, we can avoid the cost of reading the data from the current device location as its memory copy can be used during the migration to the new device. Migrating non-active blocks is carried out by a background process during quiescent periods.

Depending on the initial allocation, there may be several blocks that could benefit from migration. A number of strategies could be employed in choosing which blocks to migrate.

We maintain a cache of recently accessed blocks and migrate most recently and frequently accessed blocks that could benefit from migration. When ever a migration

token is generated, we migrate a block from this cached list. The cached list helps in utilizing the migration activity to benefit the most active blocks.

Migration is carried out in blocks or chunks of 64KB or larger. Larger block size increases migration costs, reduces the size of the indirection map, can benefit from spatial locality or similarity of access patterns. We study the impact of the block size on the system performance. We investigate migration policies that consider the read/write access patterns and the request sizes.

We study several allocation policies since the allocation policies could not only affect the migration performance, but can also affect the system performance significantly (even without migration). These policies include (a) allocation of all the data on flash while it fits; allocating the later data on magnetic disk when flash device becomes full. (b) allocation of all the data on the disk drive, (c) striping of data across both flash and disk drives; when flash device becomes full, we will allocate data only on the magnetic device, and (d) allocation of metadata on flash; metadata typically observes a higher request rate than normal data and it has been previously suggested that placing the metadata on flash may be beneficial for system's performance.

We also consider a combination of some of these policies when possible. We consider the impact of migration along with the allocation policies in our study.

C. Implementation

We developed a Linux kernel driver that implements several policies for migration and managing space across flash and disk drives in our system. The architecture and several modules within the space management layer are shown in Fig. 3. The space management layer sits below the file systems and above the device drivers. We implemented several policies for detecting access patterns of blocks. The sequential

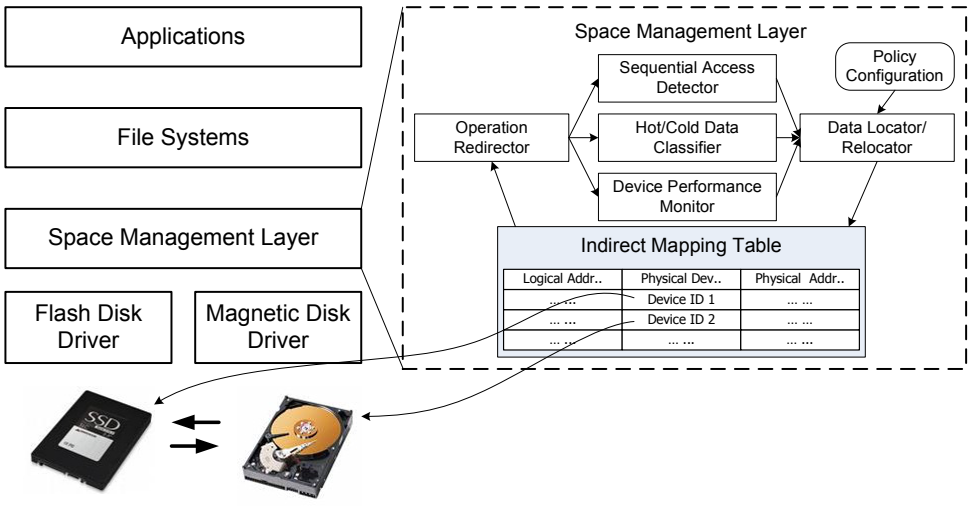


Fig. 3. Architecture of space management layer.

access detector identifies if blocks are being sequentially accessed by tracking the average size of sequential access to each block. The device performance monitor keeps track of the read/write request response times at different devices. The hot/cold data classifier determines if a block should be considered hot. The indirection map directs the file system level block addresses to the current location of those blocks on the underlying flash and disk devices. The indirection map is maintained on the hard disk, a memory copy of it allows faster operations. The block access characteristics, in our implementation, are only maintained in memory. The block access counters are initialized to zero, both on bootup and after a migration.

In the experimental environment, the NFS server was a commodity PC system equipped with an Intel Pentium Dual Core 3.2GHz processor, 1GB of main memory. The magnetic disk used in the experiments was one 7200 RPM, 250G SAMSUNG SATA disk (SP2504C), the flash disk drives are a 16GB Transcend SSD (TS16GSSD25S-S), and a 32GB MemoRight GT drive, which were connected to Adaptec SCSI Card 29160 through a SATA-SCSI converter(ADSALVD160). All the

NFS clients are simulated by one load generator in the environment.

The operating system on the NFS server was Fedora 9 with a 2.6.21 kernel, and the file system used was the Ext2 file system. The hybrid storage system is connected to the server and a number of other PCs are used to access the server as clients. We clean up the file system before each experiment (using file system mkfs command). Device level performance is measured using processor jiffies.

The next section presents performance evaluation and comparison of different policies.

D. Evaluation

1. Workload

We used multiple workloads for our study. The first workload, SPECsfs benchmark, represents file system workloads. The second workload, Postmark benchmark, represents typical access patterns in an email server. We also use IoZone [25] benchmark to create controlled workloads at the storage system in order to study the impact of read/write request distributions on the performance of the hybrid system.

SPECsfs 3.0 is the SPEC's benchmark for measuring NFS file server's performance [22]. This synthetic benchmark generates an increasing load of NFS operations against the server being evaluated and measures its response time and the server throughput as load increases. The operations of the benchmark consists of small metadata operations and reads and writes.

SPECsfs reports a curve of response time vs. *delivered* throughput (not offered load). The system performance is measured by *base response time*, the *slope* of the response time vs. throughput curve and its *throughput saturation point*. The speed of the server and client processor, the size of file cache, and the speed of the server devices

determine these measured characteristics of the server [23]. We employed SPECsfs 3.0, NFS version 3 using UDP. At higher loads, the delivered throughput can decrease as requests time out if not completed within the time limit of the benchmark (50ms).

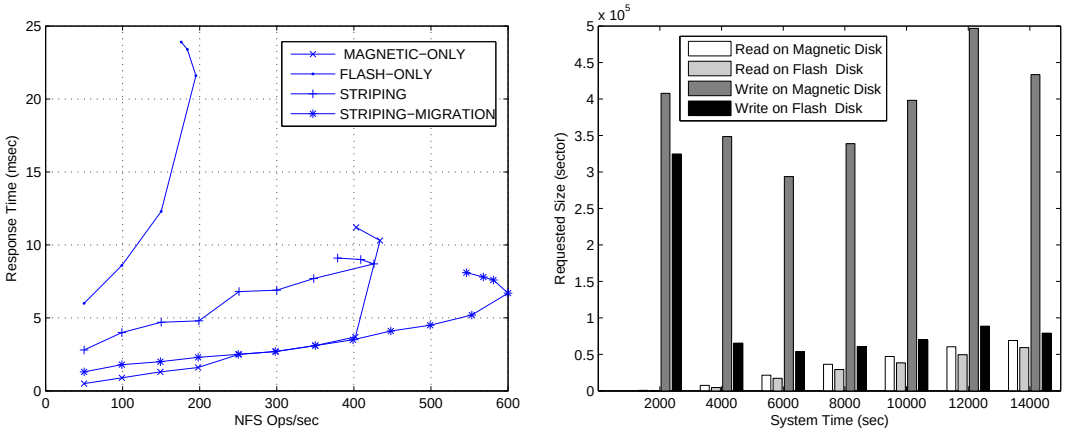
SPECsfs is used for our evaluation because it reflects realistic workloads. It tries to recreate a typical workload based on characterization of real traces by deriving its operation mix from much observation of production systems [22]. This benchmark also reflects the locality properties of real workloads, in terms of block accesses.

The SPECsfs benchmark exhibits a read write ratio of roughly 1:4 in our experimental environment, i.e., for every read request, roughly 4 write requests are seen at the storage system.

In order to study the impact of different workloads, we also employed IoZone. IoZone is a synthetic workload generator. We employed 4 different processes to generate load. Each process can either read or write data. By changing the number of processes reading or writing, we could control the workload read/write ratio from 100%, 75%, 50%, 25% and 0%. We employed Zipf distribution for block access to model the locality of accesses. In this test, we also bypassed the cache to keep a careful control of the read/write mix at the storage system.

We use Postmark [41] as a third workload. Postmark is an I/O intensive benchmark designed to simulate the operation of an e-mail server. The lower bound of the file size is 500 bytes, and upper is 10,000 bytes. In our Postmark tests, we used Postmark version 1.5 to create 40,000 files between 500 bytes and 10 kB and then performed 400,000 transactions. The block size was 512 bytes, with the default operation ratios and unbuffered I/O.

We expect these three workloads generated by SPECsfs, IoZone and Postmark to provide insights into the hybrid system performance in typical file system workloads and in other workloads with different read/write request distributions.



(a) (b)

Fig. 4. (a)Benefit from measurement-driven migration. (b)Request distribution in measurement-driven migration.

2. Benefit From Migration

In the first experiment, we evaluate the benefit from the measurement-driven migration. We compare the performance of the following four policies:

- **FLASH-ONLY:** Data is allocated on the flash disk only.
- **MAGNETIC-ONLY:** Data is allocated on the magnetic disk only.
- **STRIPING:** Data is striped on both disks.
- **STRIPING-MIGRATION:** Data is striped on and migrated across both disks.

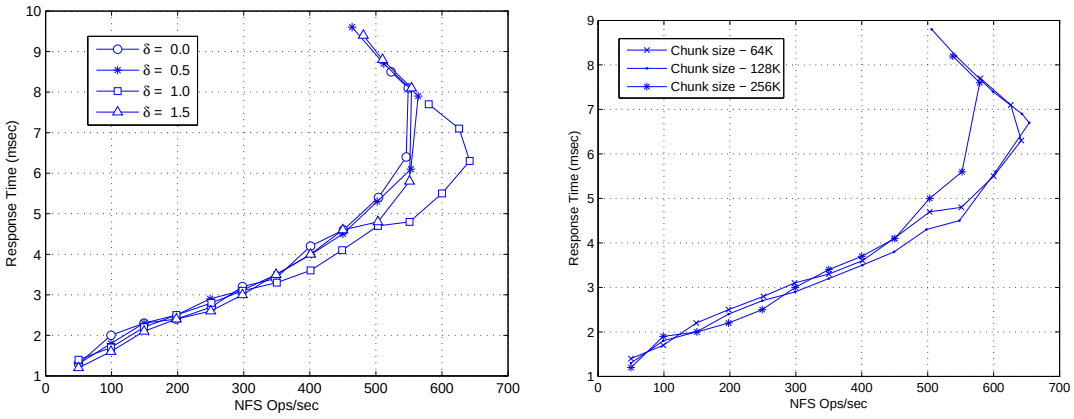
The results are shown in Fig. 4(a) for Transcend flash drive. Since there are more write requests in the workload than read requests, and write performance of the flash disk is much worse than that of the magnetic disk, the response times for FLASH-ONLY are longer than those for MAGNETIC-ONLY. Both these systems utilize only one device. The performance of STRIPING is between these two systems even though we get benefit from utilizing both the devices. The performance of the hybrid system is again impacted by the slower write response times for the data allocated on the flash device. Ideally, the addition of a second device should improve

performance (which happens with the faster MemoRight flash drive as shown later).

As can be seen from Fig. 4(a), the measurement-based migration has the throughput saturation point at 600 NFS operations/sec, that is much better than 426 NFS operations/sec in the STRIPING policy, and 434 NFS operations/sec in the MAGNETIC-ONLY policy. This improvement benefits from the data redistribution which matches the read/write characteristics of blocks to the device performance.

Fig. 4(b) shows the request distribution in different system periods. At the first, beginning at 2000 seconds, the number of the write requests directed to the magnetic disk and to the flash disk are quite close. However, over time, more and more write-intensive data are migrated to the magnetic disk, resulting in more write requests at the magnetic disk. For example, in Fig. 4(b), at the system time between 12000 and 14000 seconds, there are 433343 sectors written to the magnetic disk while only 79043 sectors are written to the flash disk (i.e. nearly 5.5 times as many sectors are written to disk compared to flash), while the read request sizes to the devices are close to each other at 69011 and 59390 sectors respectively. This request pattern at the end of the simulation shows that our approach is succeeding in redistributing write-intensive blocks to the magnetic disk even though the initial allocation of blocks distributes these blocks equally on the two devices.

This experiment shows that the measurement-driven migration can effectively redistribute the data to the right devices and help decrease the response time while improving the throughput saturation point of the system.



(a) (b)

Fig. 5. Performance impact of design parameters. (a)Impact of different δ values. (b)Impact of different chunk Sizes.

3. Sensitivity study

a. Impact of Migration Threshold Parameter δ

As explained earlier, δ is a configurable parameter, that controls how much performance advantage is expected (after migration) for the accesses to the block that is being migrated. In this experiment, we evaluated the impact of using different δ values. Generally, the smaller the δ, the higher the probability that the chunk will be migrated. As a result, too small a value of δ can cause higher migration. On the other hand, if the value is too large, the efficiency of the migration can be weakened as the data can not be remapped to the right device. Based on the results from the experiment shown in Fig. 5(a), the value $\delta = 1$ is chosen for the rest of the tests.

b. Impact of Chunk Size

Fig. 5(b) shows the impact of using the different chunk sizes. Chunk sizes of 64KB and 128KB had nearly the same performance at various loads while a larger chunk size of 256KB showed worse performance. In all the following experiments, we used the chunk size of 64KB.

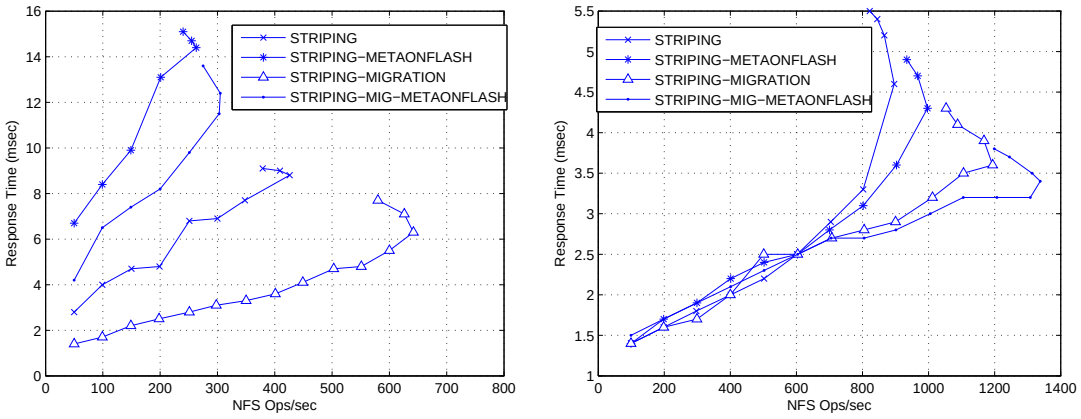


Fig. 6. (a)Impact of file system metadata placement on SPECsfs 3.0 using Transcend 16G. (b)Impact of file system metadata placement on SPECsfs 3.0 using MemoRight 32G.

4. Impact of File System Metadata Placement

Previous research states that on the magnetic disk, file system metadata placement and access efficiency is important for the overall performance [27, 28, 38]. Since flash disk has faster read speed, many researchers believe that it is better to store the file system metadata on the flash device. In this experiment, we evaluate the impact of the file system metadata placement policy, especially the effect of storing the metadata on the flash device only. In order to conduct this experiment, we relied on file system information that is generally not available at the device level. The metadata blocks are written by the ext2 file system at the time of file system creation and hence they occupy addresses at the beginning of the device. We give this range of block addresses special consideration to conduct this experiment. We restrict the blocks within this address range to be placed on flash for this experiment and also study the impact of not allowing these blocks to be migrated between flash and disk.

In this experiment, a new mechanism for dealing with the file system metadata is provided to the STRIPING and STRIPING-MIGRATION policies. All the file system

metadata are mapped to the flash disk and this metadata region is not considered for migration. STRIPING-METAONFLASH shows the results of the system when metadata is placed on the flash disk and no migration takes place and STRIPING-MIG-METAONFLASH shows the results of the system where metadata is fixed on the flash disk while normal data is allowed to migrate. We compare these two new systems with the two earlier systems STRIPING (with no migration) and STRIPING-MIGRATION (where data and metadata are both striped and allowed to migrate). The test results for Transcend flash drive are shown in Fig. 6(a) and the MemoRight drive in Fig. 6(b).

For the Transcend drive, fixing the file system metadata on the flash disk did not give any improvement to the performance. Instead, the response time of the whole system has increased significantly. Meanwhile, the throughput saturation point was also greatly decreased. The phenomenon is caused by the abundant write requests to the file system metadata in the workload. Even though the reading response time reduced, the writing response time was increased significantly. As a result, the total performance was not improved. This is primarily because of the very slow write performance of the Transcend flash drive.

For the MemoRight drive, fixing metadata on the flash drive is shown to improve the performance of the system by about 10-15%. The faster write performance of this flash drive clearly improved the performance compared to the system with the Transcend drive.

The results from these experiments with different flash devices highlight a number of points. First, the allocation policies of the hybrid system will depend on the individual drives as seen by the fact that performance improved in one system and worsened in another system when metadata is fixed on the flash drive. However, migration successfully improved performance in both systems with different flash

devices. Moreover, in both the systems, migration improved performance irrespective of initial allocation of data (whether metadata was fixed on the flash drive or not). This points to the flexibility gained through measurement-based migration policy. These results also point to the need for faster write performance on flash devices for hybrid systems to be more effective.

5. Comparison with 2-harddisk Device

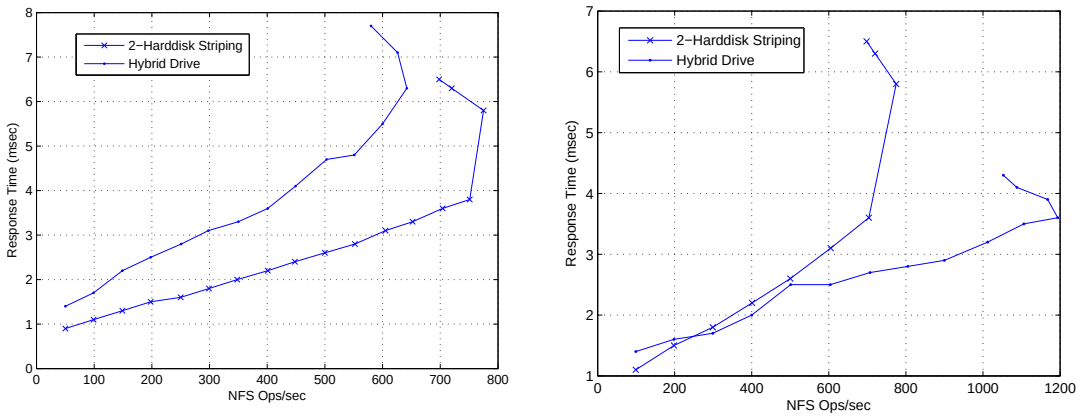
In this experiment, we compared the performances of the hybrid storage system and a 2-hard disk striping storage system (data is striped on two hard disks and no migration is employed). First, we used SPECsfs 3.0 to generate the workload.

Fig. 7(b) shows the comparison of the MemoRight-based hybrid system against the 2-disk system. It is observed that the hybrid system outperforms the 2-harddisk striping storage system, achieving nearly 50% higher throughput saturation point. The hybrid storage system delivers higher performance, fulfilling the motivation for designing such a system.

Fig. 7(a) shows the comparison of the Transcend-based hybrid system against the 2-disk system. It is observed that the 2-harddisk striping storage system works better than the hybrid drive on both the saturation point and the response time.

We used IoZone to generate workloads with different read/write ratio to find out what kind of workloads are more suitable for the hybrid storage system. In this experiment, we employed four processes to generate workload at the storage system. By varying the number of processes doing reads vs. writes, we could create workloads that 100% writes, to 75%, 50%, 25% or 0% write workloads (0R4W, 1R3W, 2R2W, 3R1W and 4R0W). We employed a Zipf distribution for data accesses in each process and bypassed the cache to maintain a controlled workload at the storage system.

The results are shown in Fig. 8. Each workload name consists of <*number of*



(a) (b)

Fig. 7. (a)Hybrid drive vs. 2-harddisk striping device on SPECsfs 3.0 using Transcend 16G drive. (b)Hybrid drive vs. 2-harddisk striping device on SPECsfs 3.0 using MemoRight 32G drive.

reading processes $\boldsymbol{R}$ *number of writing processes* $\boldsymbol{W}$>. Each process reads from or writes to a 512MB file according to a Zipf distribution. We adjusted the number of processes to control the ratio of read/write requests to the device. To bypass the buffer cache, we used direct I/O. While the 2-harddisk system did not employ migration, we tested the hybrid system with two policies STRIPING and STRIPING-MIGRATION.

As we can see from Fig. 8, the performance of Transcend-based hybrid drive with STRIPING policy is not as good as that of the 2-harddisk system, especially the writing performance. However, with migration employed, the performance of the hybrid drive achieved significant improvement, even surpassed the 2-harddisk system. The results show that the hybrid drive with migration can get higher performance improvement when the ratio of read/write requests is higher, even with the slower Transcend flash drive. When the ratio was 1:3 (in workload 1R3W), the performances of 2-harddisk system and hybrid drive with STRIPING-MIGRATION policy are almost the same.

These results indicate that read/write characteristics of the workload have a critical impact on the hybrid system. With migration, the hybrid system's performance

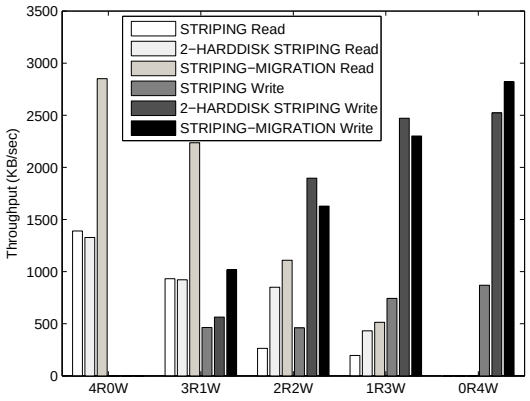


Fig. 8. Hybrid drive vs. 2-harddisk striping device on IoZone.

can be improved greatly and made to offer a performance improvement over a 2-disk system over much wider workload characteristics than it would have been possible.

The results with Postmark benchmark experiment are shown in Table I. In our experiment we employed two simultaneous processes to generate sufficient concurrency, both processes running the Postmark benchmark with default parameters. We ran our experiments across both the hybrid storage systems with Transcend and MemoRight flash drives and on the 2-disk system. It is observed that migration improved the transaction rate, read/write throughputs in both the hybrid storage systems, by about 10%. It is also observed that the Transcend-based hybrid system could not compete with the 2-hard disk system. However, the MemoRight-based hybrid system could outperform the 2-hard disk system, by roughly about 10-17%.

	Process 1			Process 2		
	Transactions /sec	Read Throughput	Write Throughput	Transactions /sec	Read Throughput	Write Throughput
Transcend-Hybrid(Striping)	52	211.46	49.22	50	204.70	47.65
Transcend-Hybrid(Migration)	58	237.18	55.21	57	229.68	53.46
MemoRight-Hybrid(Striping)	126	509.17	118.52	125	502.98	117.08
MemoRight-Hybrid(Migration)	137	553.17	128.76	134	538.76	125.41
2-Harddisk	121	487.77	113.54	115	462.86	107.74

Table I. Performance of different systems with Postmark workload. Throughput is in KBytes/second.

	Process 1			Process 2		
	Transactions /sec	Read Throughput	Write Throughput	Transactions /sec	Read Throughput	Write Throughput
Transcend-Hybrid (Striping)	17	3.19M	434.88K	17	3.19M	434.69K
Transcend-Hybrid (Migration-1)	18	3.35M	449.43K	17	3.33M	449.75K
Transcend-Hybrid (Migration-2)	19	3.47M	461.86K	18	3.45M	460.23K
MemoRight-Hybrid (Striping)	25	5.39M	610.72K	25	5.33M	602.92K
MemoRight-Hybrid (Migration-1)	26	5.69M	644.48K	26	5.67M	642.35K
MemoRight-Hybrid (Migration-2)	33	5.91M	740.05K	30	6.38M	722.17K
2-Harddisk	24	5.15M	583.10K	24	5.13M	580.32K

Table II. Performance of different systems with Postmark workload. Throughput is in Bytes/second. The lower bound of the file size is 500 bytes, and the upper bound is 500,000 bytes.

We conducted a second experiment with Postmark benchmark to study the impact of request size. In this experiment again, we employed two simultaneous Postmark processes. The file size was varied from a low of 500 bytes to 500 Kbytes, with each process generating and accessing 4,000 files. Each process conducts 80,000 total transactions in this experiment. The results of these experiment are shown in Table II. We employed two migration policies. In policy labeled Migration-1, only read/write characteristics were considered as earlier. In the second policy labeled Migration-2, request size was considered as detected by our sequentiality detector module. If the observed request size is less than one migration block (64KB in this experiment), we allowed the block to be migrated based on the read/write request patterns of that block. If the request size is observed to be larger, we allowed this data to exploit the gain that can be had from striping data across both the devices. If the block is accessed as part of a request larger than 64KB, it is not migrated.

The results of these two policies are shown in Table II. It is observed that migration policy based on read/write patterns improved the performance over striping. When we considered the request sizes and the read/write access patterns (Migration-2), the performance is observed to be higher. While the performance improved by about 7% for MemoRight based hybrid storage system when read/write patterns are considered, the performance on an average improved by about 20% when both

read/write patterns and the request size are considered. The performance for Transcend based storage system also improves in both the policies, the performance improvement is not as substantial. These experiments show that both read/write and request size patterns can be exploited to improve performance.

CHAPTER III

EXPLOIT CONCURRENCY IN A FLASH AND DISK HYBRID STORAGE SYSTEM

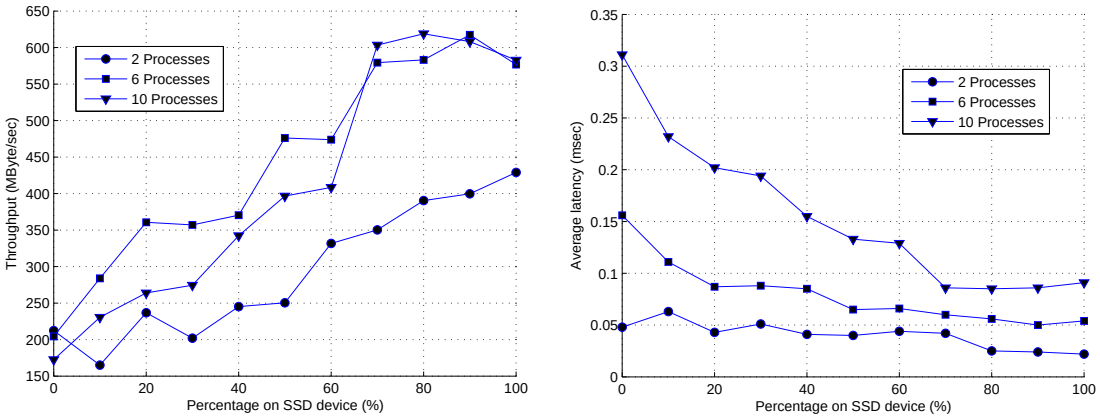
In this chapter, we propose another approach to improve the performance of hybrid storage system employing solid state disks and hard disk drives. We utilize both initial block allocation as well as migration to reach "Wardrop equilibrium", in which the response times of different devices equalize. We show that such a policy allows adaptive load balancing across devices of different performance. We also show that such a policy exploits parallelism in the storage system effectively to improve throughput and latency simultaneously. This work has been published in [46].

A. Background

Traditionally, memory systems and storage systems employ a hierarchy to exploit the locality of data accesses. In such systems, data is cached and accessed from the faster devices while the slower devices provide data to the faster devices when data access results in a miss at the faster devices. Data is moved between the different layers of the storage hierarchy based on the data access characteristics.

Employing faster devices as caches generally improves performance while hiding the complexity of handling the diversity of multiple devices with different characteristics to the upper layers. However, caching generally results only in realizing the capacity of the larger (and slower) devices since the capacity of the faster (and smaller) devices is not exposed to the upper layers.

When the capacity of the devices at different layers can be comparable, it is possible to employ other organizations to realize the combined capacity of the devices. Migration is one of the techniques employed in such situations. In such systems, the



(a) (b)

Fig. 9. Performance of hybrid device with static allocation ratios. Workload: dbench, SSD device: Intel X25-M 80GB SATA SSD, Hard drive device:15K RPM, 73G Fujitsu SCSI disk. (a) Throughput(higher is better), (b) Average latency(lower is better).

frequently or recently accessed data is stored on (or migrated to) the faster devices to improve performance while realizing the combined capacity of the devices. Even when migration is employed, the storage system can still be organized as a hierarchy with faster devices at higher layers being accessed first on any data access.

When the storage system is organized as a hierarchy (either with caching or migration), the data throughput can be limited by the rate at which data can be accessed from the faster device (even when all the data is accessed from the faster devices). The throughput is limited by the fact that the faster device(s) has to be accessed on every access. However, it is possible to provide higher throughput if data can be accessed directly from all the devices without enforcing a strict hierarchy. In such a system, it is possible to provide throughput higher than what can be provided by a strictly hierarchical system.

We explore this option, in this chapter, in organizing SSDs and magnetic disks in a hybrid system. This is driven partly by the fact that while the small read (or

random read) performance of SSDs can be significantly higher than magnetic disks, the large read/write access performance of SSDs is comparable to magnetic disks and random write performance of SSDs can be sometimes worse than that of magnetic disks, depending on the choice of SSDs and magnetic disks. Second, depending on the prices of storage on different devices, systems will be designed with different amounts of storage on SSDs and magnetic disks. We are motivated to design a storage organization that works well across many different organizations with various levels of SSD storage in a hybrid storage system.

Other systems have explored the organization of storage systems in a non-hierarchical fashion, for example [1]. The data can be stored in either type (faster or slower) device and accessed in parallel in such systems. For example, on a miss, read data may be directly returned from the slower device while making a copy in the faster device, in the background, for future references. Similarly, on a write miss, the writes may be directly written to the slower device without moving data to the faster device. However, most such systems employ faster devices until their capacity is exhausted before they start moving data to the slower devices. In the system, we explore here, the capacity of the faster device may not be completely utilized, before data is allocated on the slower devices and used in parallel with the faster devices. Our approach tries to improve both the average latency and throughput of data access simultaneously by exploiting the parallelism that is possible through accessing multiple devices concurrently.

When data is striped across multiple devices, data can be accessed in parallel from all the devices, potentially making use of all the devices in the system. However, striping allocates data uniformly across all the devices without regard to the relative performance of the devices. In the approach we adopt here, the data is distributed differently across the different devices based on the observed performance of

the devices and thus tries to match load across the devices to dynamically observed performance of the devices.

In Chapter II, we present a measurement-driven approach to exploit the performance asymmetry in such a hybrid storage system. In this approach, we observe the access characteristics of individual extents and consider migrating individual extents, if necessary, to another device whose characteristics may better match the block access patterns. This approach is self adaptive. In earlier SSDs, read performance was superior to hard drives, while write performance was inferior. In such systems, matching read dominant requests to SSDs and write dominant requests to hard drives improved performance. However current SSDs have better performance than hard drives for both read and write. Such policies will not work any longer when the SSD device performs beyond the hard drive on both read and write operations.

In order to accommodate the diversity of the devices (both SSDs and magnetic disks) and the continuous evolution of the SSD architectures and characteristics, we employ a completely performance-driven approach to this problem. We measure the performance of the devices for various requests and use the measured characteristics to drive our data organization across the multiple devices in the hybrid storage system. Our approach hence adjusts to the workloads and the configuration of number of different devices in the system. We will show through results, based on two workloads and multiple configurations of storage systems, that our approach improves performance, compared to a strict-hierarchical approach.

B. Design and Implementation

Our design is motivated by the results shown in Fig. 9. We consider a system with one Intel SSD 2.5 inch X25-M 80GB SATA SSD and one 15K RPM, 73G Fujitsu SCSI

disk. We ran a workload of dbench benchmark [47] on a storage system employing these two devices in various configurations. We consider a system where all the data is stored entirely on the SSD and different levels of allocation across the two devices. For example, in a system employing an 80/20 allocation, 80% of the data is allocated on the SSD and 20% of the data is allocated on the hard disk. The benchmark we consider is small enough such that the entire dataset can fit on the SSD. Hence, the results shown in Fig. 9 exclude the overheads of managing the data across the two devices in various configurations. It is seen that the performance is not necessarily maximized by allocating all the data on the faster device, in this case the Intel SSD.

The dbench benchmark uses throughput as a performance measure and throughput, in this case, is clearly improved when the data is stored across both the devices, enabling both the devices to be used in parallel. We also measured the average latency of all the NFS operations in the benchmark. It is observed that the throughput and latency of the system continues to improve as more and more data is allocated on the SSD, until 100% of the data is allocated on SSD, for the two workloads of 2 and 6 processes. However, with 10 processes, the throughput is maximized at about 80% allocation (on SSD) and latency is minimized at about 70% allocation.

The results in Fig. 9 also clearly show the impact of a strict hierarchical access of data. As we increase the number of processes, the SSD performance peaked at about 620MB/s and at higher number of processes, organizations that employ both SSD and disk in parallel can do better. This is an important observation that leads to our design below. When the amount of parallelism in the request stream is low, a hierarchical design works well. However, as the amount of parallelism increases in the request stream, the requests can be better served by exploiting multiple devices, even at the cost of employing slower devices in serving some of the requests. The reduction in waiting times to serve requests can contribute to improvement in performance.

It is emphasized that our experiment here employed static allocations of different percentages across the devices to study the feasibility of doing better than a strict hierarchical system and our results do not include the costs of managing the data across the two devices (in all configurations). However, these results point to the potential of exploiting the parallelism in the storage system through concurrent data access from multiple devices.

These results motivate the work in this chapter. We design a system that can utilize all the devices in the hybrid storage system irrespective of the number of devices in the storage system. This system automatically adjusts the allocation to different types of devices to improve the performance of the whole system. The allocation percentages are automatically adjusted based on the number of different types of devices in the system. We implement a Linux prototype employing our ideas. We show through trace-driven evaluation of our system, that our system can improve throughput and latency of data access simultaneously compared to a strictly hierarchical system.

The architecture of the prototype is same as that in previous chapter (Fig. 3). We also take a block level approach so that the solution can work with any file system above. Measuring device level performance at the file system is made difficult due to page cache buffering and read ahead policies of the operating system.

The space management layer provides an indirection mapping between the physical device addresses employed by the file system and the actual physical blocks on the device where data is written. It maintains a block map of where a given block is located on which device. We manage space in extents of 128K bytes. For each logical extent, we use one bit to indicate the underlying device and store a physical extent number where this extent can be found on the device. The overhead introduced by the mapping table is low. For example, for a system whose capacity is 1T bytes, the

size of the full mapping table is $(1T/128K) * \lceil(\log_2(1T/128K) * 2 + 1)/8\rceil = 48M$ bytes, and the overhead is about 0.0046%. The space management layer also allocates new blocks. The allocation ratios and the migration rates from one device to another are controlled by the observed performance of the devices, always preferring the devices with faster performance. The device performance monitor in this layer keeps track of the request response times at different devices. And the hot/cold data classifier determines if a block should be considered hot. The cold data will only be migrated in the background.

The space management layer may allocate blocks on magnetic disks even before the space on the SSDs is completely allocated, depending on the observed performance of the SSDs relative to the magnetic disks.

This architecture supports flexible policy configuration. In this work, we propose a new policy that only considers the aggregate response time as a performance metric and uses allocation as the main vehicle to balance the workload. As we will show, this policy can improve both throughput and latency even when the SSD device performs beyond the HDD device on both read as well as write operations. In the rest of this section, we describe the details of the policy.

Our basic approach is driven by a goal of trying to reach "Wardrop equilibrium". When the devices in the system are at a Wardrop equilibrium, the average response time for a data item cannot be improved by moving that data item alone from its current device to another device. Typically, Wardrop equilibrium can be reached when the response times of different devices equalize. Such an approach is employed in road transportation problems and in multi-path routing in networks [48, 19]. In order to equalize the response times of devices, we may have to subject the devices to different loads.

In this work, we consider device performance in making initial storage alloca-

tions as well as migration of blocks after they are allocated. It is emphasized that these allocations can span across all the devices before the faster device's capacity is filled. In the current system, whenever a new block is written to the storage system, the storage system dynamically decides where to store that block of data based on observed performance of the devices. A space management layer keeps track of the location of the different blocks across the devices in the storage system, providing a mapping of file system logical block addresses to physical block addresses in the storage system. In order to keep this space management layer efficient, data is managed in "extents", different from file system blocks. An extent, typically, consists of multiple file system blocks.

This allocation policy tends to allocate more blocks on better performing devices and exploits slower devices as the faster devices get loaded with increasing request rates. As a result of this dynamic performance-driven approach, initially the data is allocated on the faster devices, SSD in our system. As the waiting times increase at the SSDs, the performance of a data access gets worse and at some point in time, data can be potentially accessed faster at an idle magnetic disk. When that happens, data is allocated on the magnetic disk as well, thus allowing parallelism to be exploited in serving requests. The details of this allocation policy are explained below, including how performance is measured, how allocation is shifted based on observed performance etc.

Blocks of data are migrated from one device to another based on several considerations. Cold data is moved to the larger, slower devices in the background, during idle times of the devices. Hot data could also be migrated when the loads on the devices are imbalanced or current device is not providing as good a performance to this block of data as it is likely to receive at another device. Migration happens asynchronous to the arriving request stream so that the delays in migration do not

affect the request completion time. In order to ensure that migration of data doesn't impact the performance of the foreground user requests, we give migration requests lower priority than the current user requests. Migration can take three forms in our system. First, cold data is migrated, in the background, from faster device to larger devices to make room on the smaller, faster devices. Second, hot data is migrated to lighter-loaded devices, in the background, to even out performance of the devices. Third, data is migrated on writes, when writes are targeted to the heavier-loaded device and the memory cache has all the blocks of the extent to which the write blocks belong (explained further below). To distinguish hot data from cold data, we maintain a LRU list of accessed blocks whose length is limited. Only the blocks in the list are considered to be migrated. In the implementation, this is the same LRU list used in the write-through cache .

Among the two options of adjusting load across the devices, allocation is simpler than migration. Data can be allocated on the currently fastest device or least-lightly loaded device. As a result of an allocation decision, we impact only one device. Migration impacts two devices, the device where the data currently resides and the new device where the data is migrated to. Ideally, data is moved from the most heavily loaded device to the least lightly loaded device. We make use of both the mechanisms to balance load and to exploit parallelism in the system. We employ migration when the current allocation rate is not expected to be sufficient to balance the load and when the smaller device starts filling up. In the first case, hot blocks are migrated from higher-loaded devices to lighter-loaded devices and in the second case, data is migrated from devices that are filling up to the devices that still have unfilled capacity

(and when there is a choice among these devices, to the lighter-loaded device).

$$
\begin{aligned}
&\bar{R} = R_s \times Q_s + R_h \times Q_h \\
&P_s = Q_s + \alpha[\bar{R} - R_s] \times Q_s \\
&P_h = Q_h + \alpha[\bar{R} - R_h] \times Q_h \\
&TokenNumbers = |P_s - Q_s| \times \beta \\
&Direction = \begin{cases} \text{HD}->\text{SSD}, & \text{if } P_s > Q_s; \\ \text{SSD}->\text{HD}, & \text{else;} \end{cases}
\end{aligned}
\tag{3.1}
$$

We use a timer routine to track the performance of each device and the actual percentage of workload on each device. This routine also use this information to calculate the number of load-balancing tokens and the migration direction as specified in equation (3.1). We explain our mechanism in relation to two devices SSD and HD here to make things easier to understand. In equation (3.1), R_s and R_h are measured response times on both devices, Q_s and Q_h are measured workload distribution on each device, P_s and P_h are the target distribution we want to reach in the next round, and α and β are design parameters. If the measured response time of the device is worse(better) than the average response time in the system, the workload on that device is reduced (increased). In order to ensure that the performance of the devices can be measured, the allocations are lower bounded for each device (i.e., they can't be zero). We maintain an exponential average of the device performance by computing average response time = 0.99 previous average + 0.01 current sample. Such an approach is used extensively in networking, in measuring round trip times and queue lengths etc. Such exponential averaging smooths out temporary spikes in performance while allowing longer term trends to reflect in the performance measure.

The load-balancing tokens are consumed by both allocation and migration. The number of tokens control the rate at which the load is adjusted across the devices. As

explained earlier, allocation is a preferred mechanism for achieving our equilibrium goals. The load balancing tokens are used first by the allocation process. Depending on the values in the equations above, the allocation is slowly tilted towards the lighter-loaded device. The migration thread will only do migration when the load-balancing tokens are not completely consumed by the allocation process (for example, because there are no new writes). When a write request to new data arrives, the new data will be allocated to lighter-loaded device and consume one token if there is any. If there is no token available, the new data will be allocated according to the distribution P_s/P_h. A natural question that may come to mind is what happens to our system

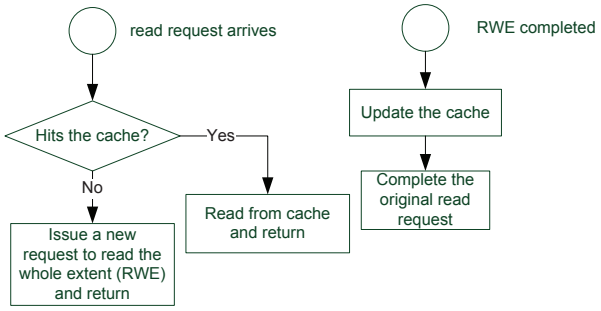


Fig. 10. Handling the read requests.

when the storage system is completely filled once. How does an allocation policy help in managing the load across the storage system? SSDs employ a copy-on-write policy because of the need for erasing the blocks before a write. In order for SSDs to carry out the erase operations efficiently, it is necessary for SSDs to know which blocks can be potentially erased (ahead of time, in the background) such that sufficient number of free blocks are always available for writing data efficiently. To accommodate this need, TRIM ATA command has been developed [49]. This command allows a file system to communicate to the storage system which of the blocks on the storage system are not live blocks in the file system. We exploit this new command in developing our system.

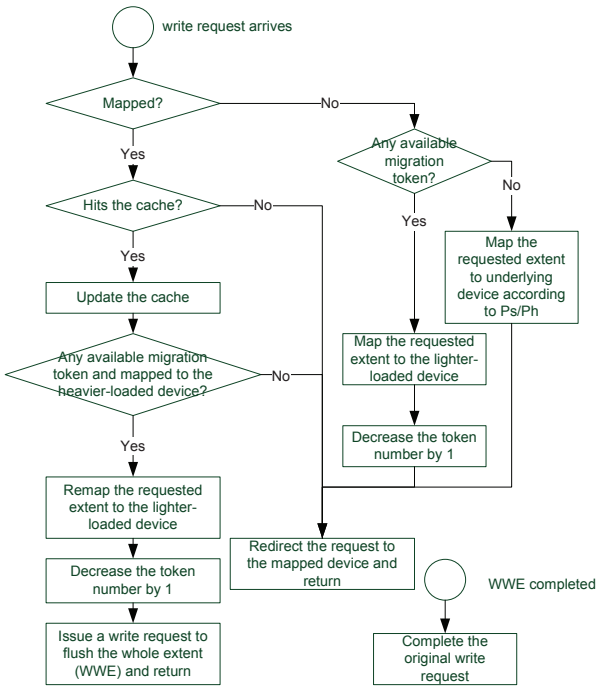


Fig. 11. Handling the write requests.

When TRIM commands are issued, the storage system can keep track of available free space on different devices and continue to employ allocation decisions in guiding its policy of balancing load and performance of different devices. A write to a logical block that is previously allocated and "trimmed" can be considered as a new allocation. With such an approach, allocation can continue being a useful vehicle for balancing load across the devices until the system is completely full.

In addition to the performance-driven migration and performance-driven allocation decisions as explained above, we also employ a number of techniques to improve performance. These include caching of entire extents in memory even if only a few of the blocks in the extent are read. This full-extent caching is expected to help during migration of blocks. Without such full-block caching, a hot block that requires migration from one device to another device may require several operations: reading of remaining blocks from the older device, merging of recently written/accessed data

with the older blocks and migration of the entire extent of blocks to the new device. When the entire extent is read and cached, migration requires only the last operation since all the blocks of the extent are currently in memory.

We implement the cache as a write-through cache with pre-fetch mechanism as shown in Fig. 10 and 11. When a read request arrives, if it hits the cache, it will be read from the cache and returned immediately. Otherwise, a new request (RWE) is issued to fetch the whole extent that contains the requested data. When this request is completed, the whole extent will be inserted into the cache and the original read request is returned to the user process. When a write request arrives, if the requested address is not mapped (i.e, write to a new extent), the requested address will be mapped to an underlying device. If there is any migration token waiting, it will be mapped to the lighter-loaded device, otherwise, the request is allocated according to the targeted distribution P_s/P_h in equation 3.1. If the requested address is mapped and the request misses in the cache, it will be redirected to the mapped device according to the mapping table. If the request hits in the cache and the requested address is mapped to the heavier-loaded device, the request extent might be re-mapped to the lighter-loaded device when there is an available token. We only need to flush the updated cache to the new mapped device to complete the migration from the old mapped device. As described above, the cache is write-through, which prevents any data loss during exceptions such as power failure. It is noted that caching here refers to memory caching and not caching in SSD.

We employ a cache of 100 recently read extents in memory. This cache is employed in all the configurations in this work to keep the comparisons fair.

We compare our system against two other systems, one in which the capacity on the SSD is allocated first and a second system that stripes data across the two devices. The first system employs caching at SSD when the capacity of the SSD

is exhausted. In the current system, the allocation across the devices is driven by dynamic performance metrics in the system and is not decided ahead of time.

Table III. Numbers of operations by type in one process of different workloads

	dbench	NetApp Traces
NTCreateX	3390740	46614
Close	2490718	46610
Rename	143586	145
Unlink	684762	29
Deltree	86	0
Mkdir	43	0
Qpathinfo	3073335	99896
Qfileinfo	538596	22506
Qfsinfo	563566	37255
Sfileinfo	276214	50527
Find	1188248	34393
WriteX	1690616	47626
ReadX	5315377	43044
LockX	11042	0
UnlockX	11042	0
Flush	237654	118

Table IV. Average sizes of ReadX/WriteX operations in different workloads

	dbench	NetApp Trace
Average ReadX size (bytes)	11853	18105
Average WriteX size (bytes)	25720	6204

C. Evaluation

1. Testbed Details

In the experimental environment, the test machine is a commodity PC system equipped with a 2.33GHz Intel Core2 Quad Processor Q8200, 1GB of main memory. The magnetic disks used in the experiments are 15K RPM, 73G Fujitsu SCSI disks (MBA3073NP), the flash disk is one Intel SSD 2.5 inch X25-M 80GB SATA SSD (SSDSA2MH080G1GC). The operating system used is Fedora 9 with a 2.6.28 kernel, and the file system used is the Ext2 file system. We clean up the file system before each experiment. To show our policy's adaptivity, we also conduct some experiments on other two SSDs, one 16GB Transcend SSD (TS16GSSD25S-S), and one 32GB MemoRight GT drive.

2. Workloads

Dbench [47] is a file system benchmark using a single program to simulate the workload of the commercial benchmark Netbench. This benchmark reports both the latency of each NFS operation and the total throughput as metrics. To evaluate our policy extensively, we also use dbench to replay some traces from real world. The real traces are obtained from an engineering department at NetApp [50]. We developed a tool to translate the NetApp traces into the format that dbench can use. The tool

generates two dbench trace files from each NetApp trace file. One of them is only used to initialize the file system image, the other one contains the exact same operation sequence as that in the original trace file. The workload is replicated under different directories as we increase the number of processes such that each process replays a workload of the original trace, different and independent of other processes.

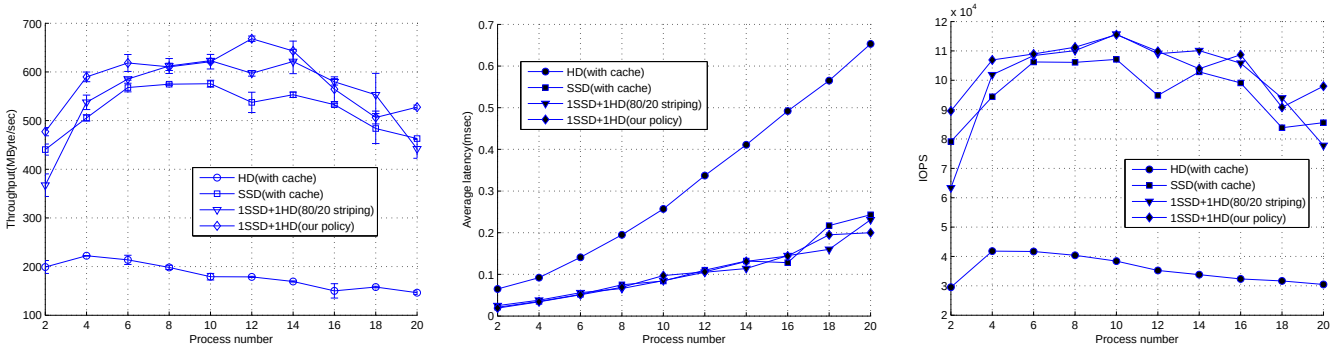
The characteristics of request sizes and read/write ratios for the two workloads are shown in Table III and IV. As seen from the table, the NetApp trace has smaller read/write ratio and smaller write request sizes.

In all the experiments, we take the design parameters $\delta = 10/ms$ and $\beta = 100$.

3. Results

Results of our experiments on a storage system with one SSD and one magnetic disk (called 1SSD+1HD configuration here) in a workload of dbench benchmark are shown in Fig. 12. Each experiment was run 5 times and the averages are shown. The 95% confidence bars are computed for all simulations, but not necessarily shown in the figure, to make it easier to read the data. The figures show a comparison of throughput (MB/s and IOPS/s) and latency across four systems, entirely running on the SSD, entirely running on the HD, on a hybrid system employing our policy and on a hybrid system employing a static allocation mix of 80/20 on SSD/HD (as identified earlier to be a good static configuration in Fig. 9). The two configurations 70/30 and 80/20 perform similarly and we use the 80/20 configuration as an example. It is observed that our policy does nearly as well as or better than the 80/20 static configuration and achieves higher throughput than the SSD or HD alone at higher number of processes. This shows that our policy dynamically adapts the allocation rate to individual devices and the workload to achieve good performance.

Results of our experiments on the storage system with 1SSD+1HD configuration



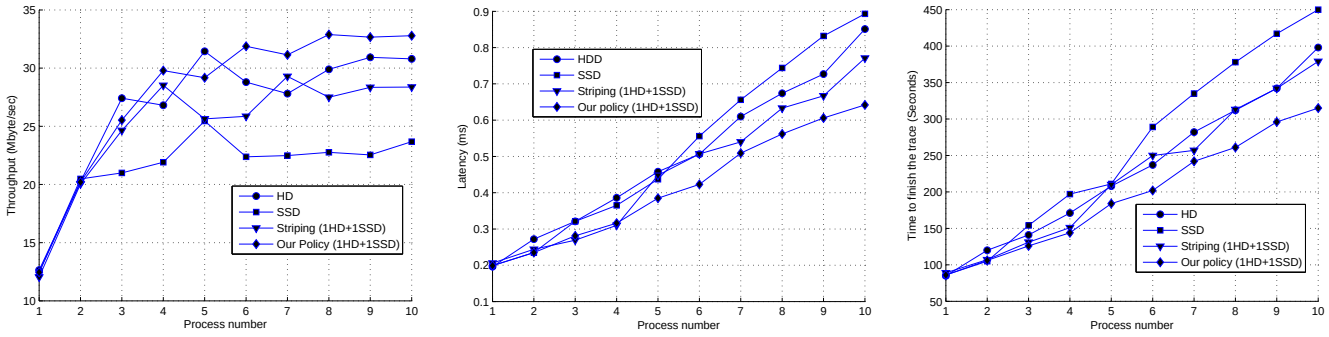
(a) (b) (c)

Fig. 12. Workload: dbench, SSD device: Intel X25-M 80G SSD, Hard drive device:15K RPM, 73G Fujitsu SCSI disk. (a) Throughput(higher is better), (b) Average latency(lower is better), (c) IOPS(higher is better).

in a workload derived from the NetApp traces are shown in Fig. 13. The figures show various performance measures as the number of processes is increased in the workload. It is observed that as the number of processes is increased, the allocation-based policy achieves higher throughput than when the data is entirely stored on the SSD or when the data is striped across the two devices (SSD and hard disk). The allocation-based policy achieves nearly 38% more throughput than the SSD and nearly 16% more throughput than a striping configuration, with 10 requesting processes. The NetApp workload has more write requests than read requests and the write requests are smaller. Both these characteristics contribute to the fact that for this workload,the throughput performance of magnetic disk drive is better than that compared to the SSD.

Fig. 13(b) shows the average latency time for I/O requests. It is observed that the allocation-based policy simultaneously improves latency along with throughput. The allocation-based policy achieves nearly 28% better latency than the SSD and 17% better latency than the striping configuration, with 10 requesting processes. This is primarily due to the simultaneous use of both the devices and the appropriately

proportional use of the devices based on their performance.



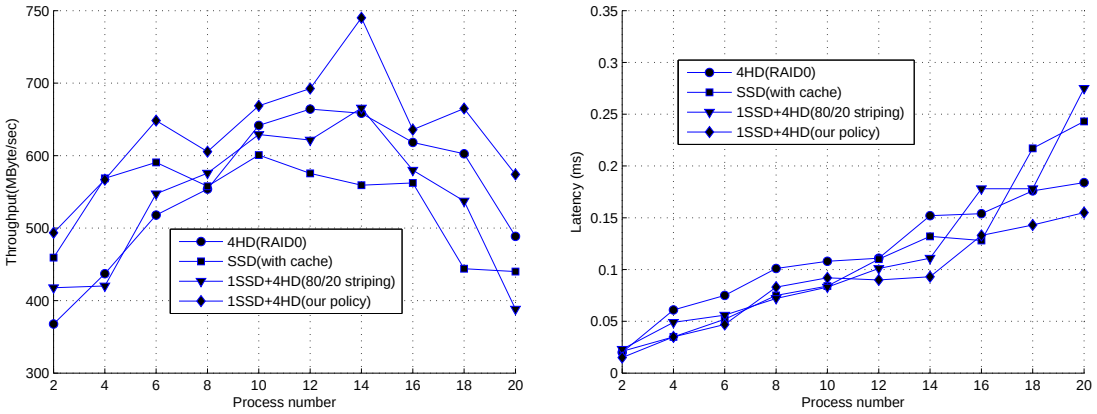
(a) (b) (c)

Fig. 13. Workload: NetApp trace, SSD device: Intel X25-M 80G SSD, Hard drive device:15K RPM, 73G Fujitsu SCSI disk. (a) Throughput(higher is better), (b) Average latency(lower is better), (c) Used time to finish the workload(lower is better).

As seen from the two workloads above, the allocation-based policy works well across two different workloads.

Below, we show that the proposed allocation-based policy works well in different configurations. We create additional configurations, 1SSD+2HD (1 Intel SSD and 2 Fujitsu magnetic disks) and 1SSD+4HD (1 Intel SSD and 4 Fujitsu magnetic disks). We study the performance of all the previously considered policies now in these new configurations. For the data on HD alone policy, we use the magnetic disks in a RAID0 configuration (striping with no parity protection). The results with dbench workload are shown in Fig. 14. To make the figures clearer, we only plot the result of 1SSD+4HD. The performance of 1SSD+2HD is between those of 1SSD+1HD and 1SSD+4HD. It is observed that our allocation policy improves performance in the new 1SSD+4HD configuration, from 1SSD+1HD configuration. This increase in performance comes from increased use of magnetic disks in supporting the workload. The performance of our policy is better than striping data across all the five devices (1SSD and 4 HDs) as shown in the figure. It has been already shown earlier that our

policy achieves better performance than when all the data resides on the single SSD. The data in Fig. 14(a) compares our policy with striping data on the four disks or when data is statically distributed in a 80/20 ratio across the SSDs and the four disks (the static allocation ratio that is found to work well earlier in the 1+1 configuration). The results show that our policy outperforms these other alternative options.



(a) (b)

Fig. 14. Workload: dbench, SSD device: Intel X25-M 80GB SATA SSD, Hard drive device: RAID0(4 15K RPM, 73G Fujitsu SCSI disks). (a) Throughput(higher is better), (b) Average latency(lower is better).

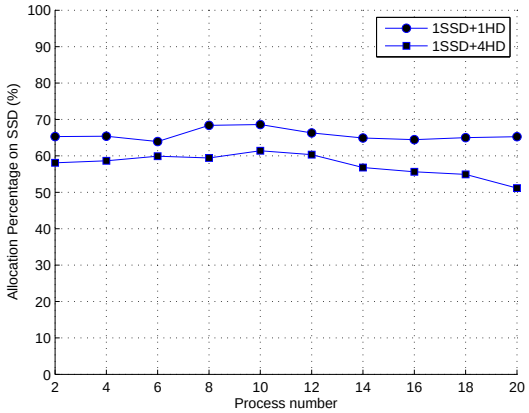


Fig. 15. Allocation percentage on SSD. Workload: dbench, SSD device: Intel X25-M 80GB SATA SSD, Hard drive device: 15K RPM, 73G Fujitsu SCSI disks.

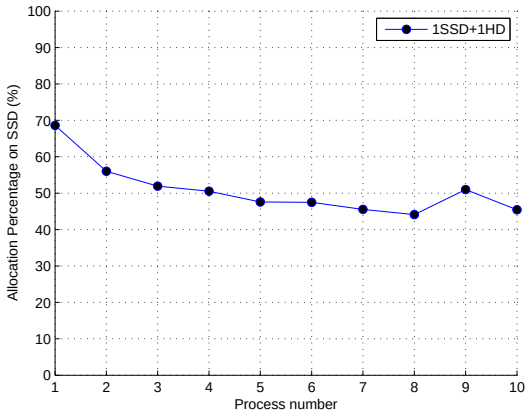
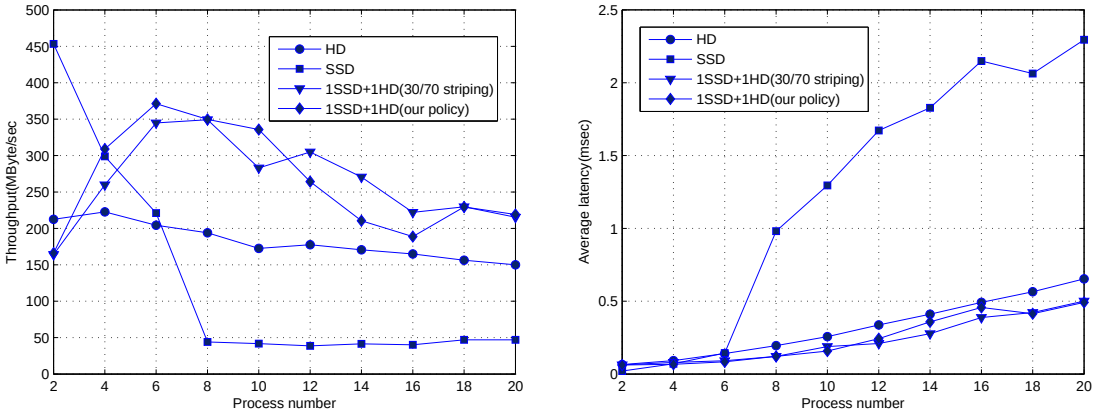


Fig. 16. Allocation percentage on SSD. Workload: NetApp, SSD device: Intel X25-M 80GB SATA SSD, Hard drive device: 15K RPM, 73G Fujitsu SCSI disks.

Table V. Performance in a transitioning workload.

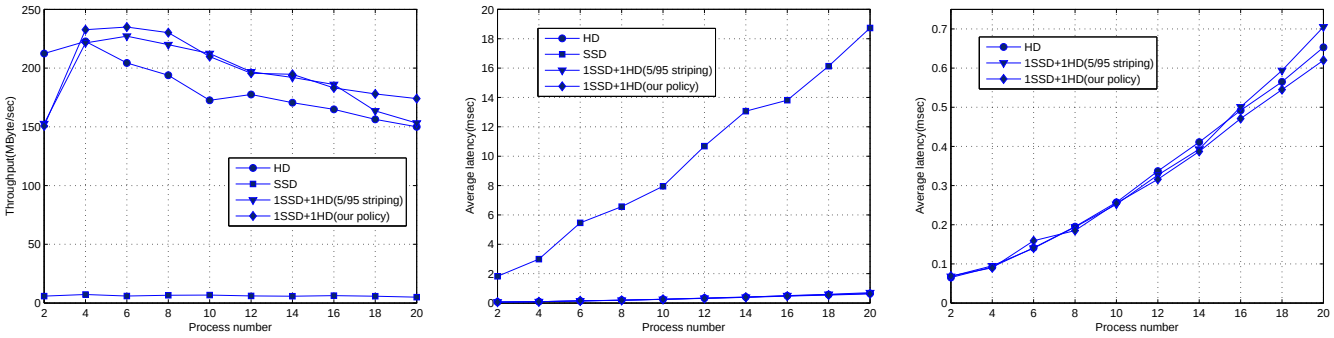
	Throughput (Mbytes/sec)		Average Latency(ms)		Percentage on SSD	
	1 proc.	10 proc.	1 proc.	10 proc.	1 proc.	10 proc.
HD	12.3106	25.9799	0.200	0.894	0%	0%
SSD	12.413	23.6754	0.199	0.933	100%	100%
1SSD+1HD(our policy)	11.5769	36.8594	0.214	0.613	65.25%	55.08%

The allocation percentages across the devices in these experiments, with the dbench workload, are shown in Fig. 15. First, we observe that our policy allocates about 65-70% of data on the SSD in the 1+1 configuration. This allocation percentage is close to one of the better configurations, 70/30, identified earlier through static allocation. We can also see that more data is allocated to magnetic disks in the 1SSD+4HD configuration than in the 1+1 configuration. Our policy adopts allocations to the availability of more disks in the new configuration and allocates smaller amount of data to SSDs.



(a) (b)

Fig. 17. Workload: dbench, SSD device: Memoright 32G SATA SSD, Hard drive device:15K RPM, 73G Fujitsu SCSI disk. (a) Throughput(higher is better), (b) Average latency(lower is better).



(a) (b) (c)

Fig. 18. Workload: dbench, SSD device: Transcend 16G SATA SSD, Hard drive device:15K RPM, 73G Fujitsu SCSI disk. (a) Throughput(higher is better), (b) Average latency(lower is better), (c) Average latency(lower is better).

We also present the allocation percentages across SSD and HD for the NetApp workload in Fig. 16. It is observed that our policy distributes about 50% of the data to SSD and 50% of the data to the hard disk in this workload. As noticed earlier, the hard disk performs better in this workload than in the dbench workload. This is the reason more data is allocated to hard disk in this workload.

As the number of processes increases, in both workloads, the allocation percentage is shifted slightly more towards hard disk compared to the distribution at lower number of processes. With higher number of processes, more devices can be exploited, even if they are slower, to improve throughput. It is observed that the allocation percentage is slightly more than 50% on the hard disk, at higher number of processes, in the NetApp workload, as seen in Fig. 16. This is consistent with the higher performance of the hard disk compared to SSD in this workload, at higher concurrency, as seen earlier in Fig. 13a.

As an additional experiment, we ran an experiment where the workload initially consisted of one process of NetApp workload and halfway during the experiment, the workload transitioned into a load of 10 processes. The results of the experiment are shown in Table V. It is observed that our policy transitions the allocation to suit the changing workload and provides good performance across both the workloads.

To show our policy's adaptivity further, we ran experiments on two other SSDs and the results are shown in Fig. 17 and 18. In these experiments, SSDs perform worse than hard drives due to their bad write performance. Similar to earlier experiments, we compare our policy to SSD only policy, hard drive only policy, and also the best case of static allocation. Our experiments show that even with the configuration where hard drive performs beyond SSD device, we still can get benefits from concurrency through our policy. Fig. 17 and 18 show that our policy adapts allocation/load across the two devices to improve performance in a dbench workload. Our policy matches

the best static allocation for each configuration.

As can be seen from the data, our policy adapts to different workloads and the configuration of the storage system to improve performance. The gains in performance are more significant with higher concurrency in the request stream of the workload.

CHAPTER IV

SCMFS : A FILE SYSTEM FOR STORAGE CLASS MEMORY

In this chapter, we present a new file system, called SCMFS, which is designed for Storage Class Memory (SCM). SCM is special class of memory device that are byte addressable as well as non-volatile. In SCMFS, we utilize the existing memory management module in the operating system to do the block management and keep the space always contiguous for each file. The simplicity of SCMFS not only makes it easy to implement, but also improves the performance.

A. Background

In this chapter, we focus on non-volatile memory which can be attached directly to the memory bus and is also byte addressable. Such nonvolatile memory can be used in the computer system for the main memory as well as for persistent storage of files. The promising nonvolatile memory technologies include Phase Change Memory(PCM)[59, 71, 70], memristor[61], and they offer low latencies that are comparable to DRAM and are orders of magnitude faster than traditional disks.

The emerging and developing of nonvolatile memory technologies bring many new opportunities for researchers. The emerging nonvolatile memory can be attached to memory bus, thus reducing the latencies to access persistent storage. These devices also enable processor to access persistent storage through memory load/store instructions enabling simpler and faster techniques for storing persistent data. However, compared to disk drives, these devices usually have much shorter write life cycles. A lot of work has been done on how to reduce write operations to and how to implement wear leveling on such devices [65, 68, 67, 69]. Since SCM's write endurance is usually 100-1000X+ order of NAND flash, the lifetime issues are expected to be less problem-

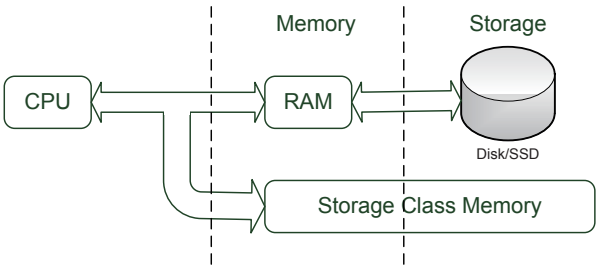


Fig. 19. Storage class memory

atic. In this chapter, we investigate how the characteristics of SCM devices should impact the design of file systems. SCM devices have very low access latency, which is much better than the existing other persistent storage devices and we consider them attached to the memory bus directly (as shown in Fig.19).

To use SCM as a persistent storage device, the most straightforward way is to use Ramdisk to emulate a disk device on the SCM device. Then it becomes possible to use a regular file system, such as Ext2Fs, Ext3Fs, etc.. The traditional file systems assume the underlying storage devices are I/O-bus attached block devices, and are not designed for memory devices. In this approach, the file systems access the storage devices through generic block layer and the emulated block I/O operations. The overhead caused by the emulation and the generic block layer is not necessary, since a file system specially designed for memory devices can be built on top of the memory access interface directly. In the traditional storage hierarchy, the additional overhead is ignorable since the latency to access storage devices is much higher than that to access memory. When the storage device is attached directly to the memory bus and can be accessed at memory speeds, these overheads can substantially impact performance and hence it is necessary to pay attention to avoid such overheads when ever possible. In addition, when storage devices are attached to the memory bus, both the storage device and the main memory will share system resources such as

the bandwidth of the memory bus, the CPU cache and the TLB. In this case, the overhead of file systems will impact the performance of the whole system, and file systems for SCM should consider these factors. In our file system, we will eliminate unnecessary overheads in the hierarchy.

Another choice is to modify the existing memory based file systems, such as tmpfs [63], ramfs. These file systems are designed to use main memory to store the files, and are not for persistent storage devices. So, these file systems do not harden any data on persistent devices to let the system restore the data from rebooting. All the metadata are maintained by the in-memory data structures, and the file data are stored in the temporarily allocated memory blocks. It is not harder to design a new file system from scratch than to adapt these file systems to SCM devices.

In this chapter, we propose a new file system - SCMFS, which is specifically designed for SCM. With consideration of compatibility, this file system exports the identical interfaces as the regular file systems do, in order that all the existing applications can work on it. In this file system, we aim to minimize the CPU overhead of file system operations. We build our file system on virtual memory space and utilize the memory management unit (MMU) to map the file system address to physical addresses on SCM. The layouts in both physical and virtual address spaces are very simple. We also keep the space contiguous for each file in SCMFS to simplify the process of handling read/write requests in the file system. We will show, through results based on multiple benchmarks, that the simplicity of SCMFS makes it easy to implement and improves the performance.

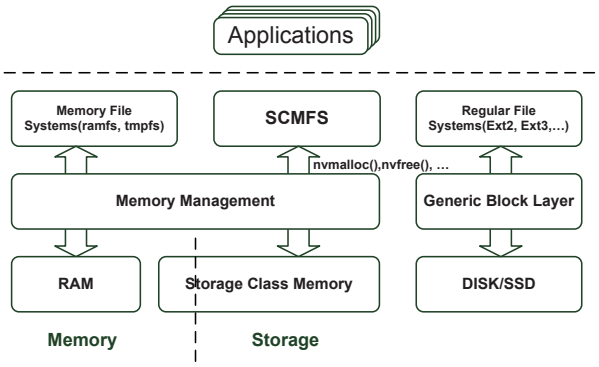


Fig. 20. File systems in operating systems.

B. SCMFS

In this section, we present the design of SCMFS and a prototype implementation on Linux 2.6.33 kernel.

1. Design

In this work, we aim to design a file system for SCM devices. With traditional persistent storage devices, the overhead brought by I/O latency is much higher than that of file system layer itself. So the storage system performance usually depends on the devices' characteristics and the performance of I/O scheduler. However, in the case that storage device is directly attached to the memory bus, the storage device will share some critical system resources with the main memory. They will share the bandwidth of the memory bus, the CPU caches and TLBs for both instruction and data. We believe that the lower complexity of the file system can reduce the CPU overhead in the storage system and then improve the total performance. Our design is motivated by the need to minimize the number of operations required to carry out file system requests, in such a system.

a. Reuse Memory Management

Current file systems spend considerable complexity due to space management. For example, Ext2fs spends almost 2000 SLOCs (source lines of code) on it. Since SCM will be visible through memory bus, it is possible to reuse the memory management module within the operating system to carry out these functions. Memory management has hardware support in the form of TLB and MMU caches to speed up operations of translating from virtual addresses to physical addresses, providing protection mechanisms across users etc. It seems natural to exploit this infrastructure to speed up file system operations as well when storage will be accessible through memory bus. SCMFS is designed to reuse the memory management infrastructure, both in the hardware and the Operating System. It is expected that such a design would benefit from the future enhancements to memory management infrastructure within the processor, through increased TLB sizes and MMU caches.

In our design, we assume the storage device, SCM, is directly attached to CPU, and there is a way for firmware/software to distinguish SCM from the other volatile memories. This assumption allows the file systems be able to access the data on SCM in the same way as normal RAM. With this assumption, we can utilize the existing memory management module in the operating system to manage the space on the storage class memory. As shown in Fig. 20, regular file systems are built on top of the generic block layer, while SCMFS is on top of the modified memory management module.

When the file system relies on the MMU for mapping virtual addresses to physical addresses, these mappings need to be persistent across reboots in order to access the data after a power failure, for example. It is not sufficient to allocate the page mapping table on the SCM since these mappings can be cached at various locations before being

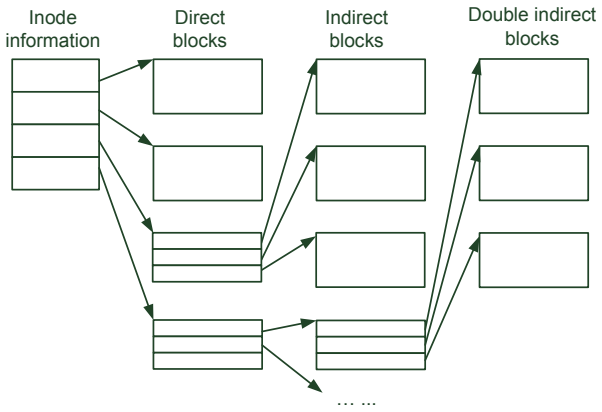


Fig. 21. Indirect block mechanism in Ext2fs

written to memory. We need to immediately harden the address mappings whenever space is allocated on SCM. We made enhancements to the kernel for this purpose, as explained later in Section IV.4.

b. Contiguous File Addresses

Current file systems employ a number of data structures to manage and keep track of the space allocated to a file. The file systems have to deal with the situation that a large file is split into several parts and stored in separate locations on the block device. For example, Ext2fs handles this by using indirect blocks, as shown in Fig. 21. This makes the process of handling the read/write requests much more complicated, and sometimes requires extra read operations of the indirect blocks.

In order to simplify these data structures, we designe the file system such that the logical address space is contiguous within each file. To achieve this, we build the file system on virtual address space, which can be larger than the physical address space of the SCM. We can use page mapping to keep all the blocks within a file to have contiguous logical addresses. In SCMFS, with the contiguity inside each file, we do not need complicated data structures to keep track of logical address

space, and simply store the start address and the size for each file. This mechanism significantly simplifies the process of the read/write requests. To get the location of the request data, the only calculation is adding the offset to the start address of the file. The actual physical location of the data is available through the page mapping data structures, again leveraging the system infrastructure.

As described above, putting the file system on virtual address space can simplify the design and reduce overheads. However, it also has a side affect that it may cause more TLB misses. Operating systems sometimes map the whole memory in the system to a linear address space using a larger page size (e.g., 2MB), resulting in smaller number of TLB misses. In our current implementation, to minimize the internal fragmentation we use a page size of 4K bytes. Hence, we may incur more TLB misses than if we were to employ linear mapping of the virtual address space corresponding to the file system. We will see its impacts in Section IV.C.

2. File System Layout

Fig. 22 shows the layout of both virtual memory space and physical memory space in SCMFS. The "metadata" in physical memory space contains the information of storage, such as size of physical SCM, size of mapping table, etc. The second part of the physical memory is the memory mapping table. The file system needs this information when mounted to build some in-memory data structures, which are mostly maintained by memory management module during runtime. Any modification to these data structures will be flushed back into this region immediately. Since the mapping information is very critical to the file system consistency, all the updates to this region will be flushed immediately by using the procedure "clflush_cache_range" described in Section IV.6. The rest of the physical space is mapped into virtual memory space and used to store the whole file system.

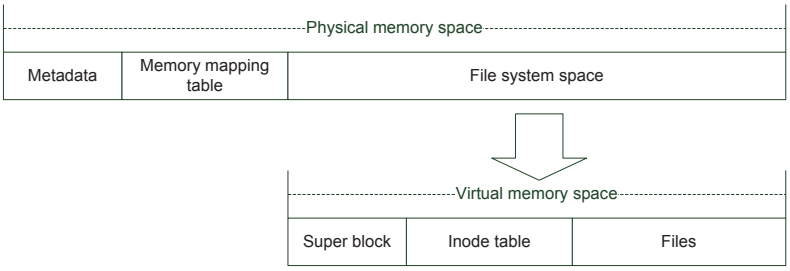


Fig. 22. Memory space layout

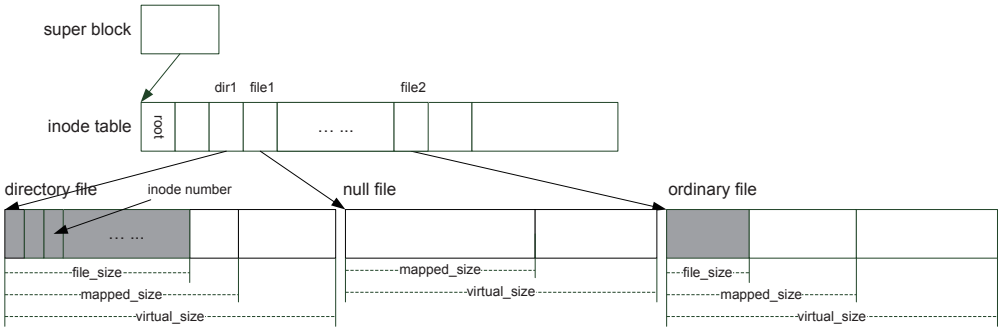


Fig. 23. SCM file system Layout.

In the virtual memory space, the layout of SCMFS is very simple and similar to existing file systems, and it consists of three parts. The first part is the super block, which contains the information about the whole filesystem, such as file system magic number, version number, block size, counters for inodes and blocks, the total number of inodes and blocks, etc.. The second part is the inode table, which contains the fundamental information of each file or directory, such as file mode, file name, owner id, group id, file size in bytes, the last time the file was accessed (atime), the last time the file was modified (mtime), the time the file was created (ctime), start address of the data for the file, etc.. The first item in the inode table is the root inode that is always a directory. All the content of the files in the file system are stored in the third part. In our prototype, the total size of virtual memory space for SCMFS is 2^{47} bytes (range: ffff000000000000 - ffff7fffffffffff), which is unused in original Linux

kernel.

The structure of SCM file system is illustrated in Fig.23. In SCM file system, directory files are stored as ordinary files, except that their contents are lists of inode numbers. Besides ordinary files and directory files, in SCMFS, there is an additional type of file, a null file, which will be described in Section IV.3. There is also a pointer to the start address of inode table in the super block. In the inode table, we use a fixed size of entry, which is 256 bytes, for each inode and it is very easy to get a file's metadata through its inode number and the start address of the inode table.

With the layouts, the file system can be easily recovered or restored after re-booting. First we check if the "metadata" is valid through a signature, and use the information in the "metadata" and "mapping table" to build the mapping between the physical addresses and the virtual addresses. Once we finish this, we can get the information about the file system from the super block in the virtual address space. It is noted that both the physical and the virtual address in the mapping table need to be relative instead of absolute to provide the portability between different machines and systems.

3. Space Pre-Allocation

In traditional file systems, all the data blocks are allocated on demand. The space is allocated to the files only when needed, and once any file is removed, the space allocated for it will be deallocated immediately. Frequent allocation and deallocation can invoke many memory management functions and can potentially reduce performance. To avoid this, we adopt a space pre-allocation mechanism, in which we create and always maintain certain amount of null files within the file system. These null files have no name, no data, however have already been allocated some physical space. When we need to create a new file, we always try to find a null file first. When a file shrinks,

we will not de-allocate the unused space. And when we need to delete an existing file, we will not de-allocate its space but mark it as a null file. Through the space pre-allocation mechanism, we can reduce the number of allocation and deallocation operations significantly, and expect to boost the file system performance.

To support this mechanism, we need to maintain three "size"s for each file. The first one, "file_size" , is the actual size of the file. The second one, "virtual_size" is the size of the virtual space allocated to the file. The last one, "mapped_size", is the size of mapped virtual space for the file, which is also the size of physical space allocated to the file. The value of "virtual_size" is always larger than or equal to that of "mapped_size", whose value is always larger than or equal to that of "file_size".

The space unused but mapped for each file is reserved for later data allocations, and potentially improves the performance of further writing performance. However, these spaces are also likely to be wasted. To recycle these "wasted" spaces, we use a background process. This method is very similar to the garbage collection mechanism for flash based file systems. This background thread will deallocate the unused but mapped spaces for the files when the utilization of the SCM reaches a programmable threshold, and it always chooses cold files first.

4. Modifications to kernel

In our prototype, we make some modifications to original Linux kernel 2.6.33 to support our functionalities. First, we modify the E820 table, which is generated by BIOS to report the memory map to the operating system[78]. We add a new address range type "AddressRangeStorage". This type of address range should only contain memory that is used to store non-volatile data. By definition, the operating system can use this type of address range as storage device only. This modification makes sure the operating system has the ability to distinguish SCM from normal memory

device.

Second, we add a new memory zone "ZONE_STORAGE" into the kernel. A memory zone in Linux is composed of page frames or physical pages, and a page frame is allocated from a particular memory zone. There are three memory zones in original Linux: ZONE_DMA is used for DMA pages, "ZONE_NORMAL" is used for normal pages , and "ZONE_HIGHMEM" is used for those addresses that can not be contained in the virtual address space(32bit platform only). We put all the address range with type "AddressRangeStorage" into the new zone "ZONE_STORAGE".

Third, we add a set of memory allocation/deallocation functions, nvmalloc()/nvfree(), which allocate/deallocate memory from the zone "ZONE_STORAGE". The function nvmalloc() derives from vmalloc(), and allocates memory which is contiguous in kernel virtual memory space, while not necessary to be contiguous in physical memory space. The function nvmalloc() has three input parameters: size is the size of virtual space to reserve, mapped_size is the size of virtual space to map, write_through is used to specify if the cache policy for the allocated space is write-through or write-back. We also have some other functions, such as nvmalloc_expand() and nvmalloc_shrink(), whose parameters are same as that of nvmalloc(). The function nvmalloc_expand() is used when the file size increases and the mapped space is not enough, and nvmalloc_shrink() is used to recycle the allocated but unused space.

All the modifications involve less than 300 lines of source code in kernel.

5. Garbage Collection

As described above, the mechanism of pre-allocation is used to improve the speed of appending data to files. However it causes the waste of space since we may pre-allocate some space for the files ever appended later. To recycle the wasted space, we provide a garbage collection mechanism. Using a garbage collection in a file system

is normal, especially for the flash file systems. To minimize its impact on the system performance, we implement this mechanism in a background kernel thread. When the unmapped space on the SCM is lower than a threshold, this background will try to free the unnecessary space, that is mapped but not used. During the garbage collection, it will check the number of null files first. If the number exceeds a predefined threshold, it will free the extra null files. If we need to free more, this thread will consider the cold files first, that have not been modified for a long time, then the hot files. We can easily classify the cold/hot file through the last modified time. This thread also takes the responsibility of creating null files when there are too few null files in the system.

Even though our current system doesn't implement any wear leveling functions, we expect to incorporate wear leveling into a background process that can work with the garbage collection thread.

6. File System Consistency

File system consistency is always a big issue in file system design. As a memory based file system, SCMFS has a new issue: unsure write ordering. The write ordering problem is caused by CPU caches that stand between CPUs and memories [64]. Caches are designed to reduce the average access latency to memories. To make the access latency as close to that of the cache, the cache policy tries to keep the most recently accessed data in the cache. The data in the cache is flushed back into the memory according to the designed data replacement algorithm. And the order in which data is flushed back to the memory is not necessarily the same as the order data was written into cache. Another reason that causes unsure write ordering is out-of-order execution of the instructions in the modern processors. To address the problem of unsure write ordering, we can use a combination of the instructions MFENCE and CLFLUSH.

This combination has been implemented with the function "clflush_cache_range" and used in the original Linux kernel. The instruction MFENCE is used to serialize all the load/store instructions that are issued prior to the MFENCE instruction, which guarantees that every load/store instruction that precedes, in program order, the MFENCE instruction is globally visible before any load or store instruction that follows the MFENCE instruction becomes globally visible. The instruction CLFLUSH is used to invalidate the cache line that contains the specified address from all levels of the processor cache hierarchy. By using the function "clflush_cache_range", we can provide the ensured write order to any range of addresses.

In SCMFS, we always use the function "clflush_cache_range" when we need to modify the critical information, including "metadata", "superblock", "inode table" and "directory files". This simple mechanism will provide metadata consistency. As to the data consistency, we flush the CPU cache periodically. This provides similar guarantees as the existing regular file systems.

C. Evaluation

To evaluate our ideas, we implement a prototype of SCMFS in Linux. This prototype consists of about 2700 source lines of code, which is only 1/10 of that of ext2fs in Linux. In this section, we present the results by using some standard benchmarks.

1. Bechmarks and Testbed

To evaluate SCMFS thoroughly, we use multiple benchmarks. The first benchmark, IoZone [25], is a synthetic workload generator. This benchmark creates a large file, and issues different kinds of read/write requests on this file. Since the file is only opened once in each test, we use the benchmark IoZone to evaluate the performance

of accessing file data. The second benchmark, postmark [41] is an I/O intensive benchmark designed to simulate the operation of an e-mail server. This benchmark creates a lot of small files and performs read/write operations on them. We use this benchmark to evaluate SCMFS's performance on small files and metadata.

In the experimental environment, the test machine is a commodity PC system equipped with a 2.33GHz Intel Core2 Quad Processor Q8200, 8GB of main memory. We configure 4GB of the memory as the type "AddressRangeStorage", and use it as Storage Class Memory. The operating system used is Fedora 9 with a 2.6.33 kernel.

In all the benchmarks, we compare the performance of SCMFS to that of other existing file systems, including ramfs, tmpfs and ext2fs. Since ext2fs is designed for a traditional storage device, we run ext2fs on ramdisk, which emulates a disk drive by using the normal RAM in main memory. It is noted that ramfs, tmpfs and ramdisk are not designed for persistent memory, and none of them can be used on storage class memory directly.

2. IoZone Results

Using IoZone, we evaluate the sequential and random performance, and the results are shown in the Fig.24(a,b) and Fig.25(a,b) respectively. We also use the performance counters in the modern processors, through the PAPI library [60], to see the detailed performance information related to CPU's functional units, including L1/L2 cache miss rate, Data/Instruction TLB misses. We show this information in the rest of Fig.24 and Fig.25.

In these figures, we can see that the performances of all the file systems decreases dramatically when the record length is more than 1 megabytes. This is because that when record length is too large, L2 cache miss rate and Data TLB misses increases significantly, as shown in the Fig.24(g,h,i,j) and Fig.25(g,h,i,j).

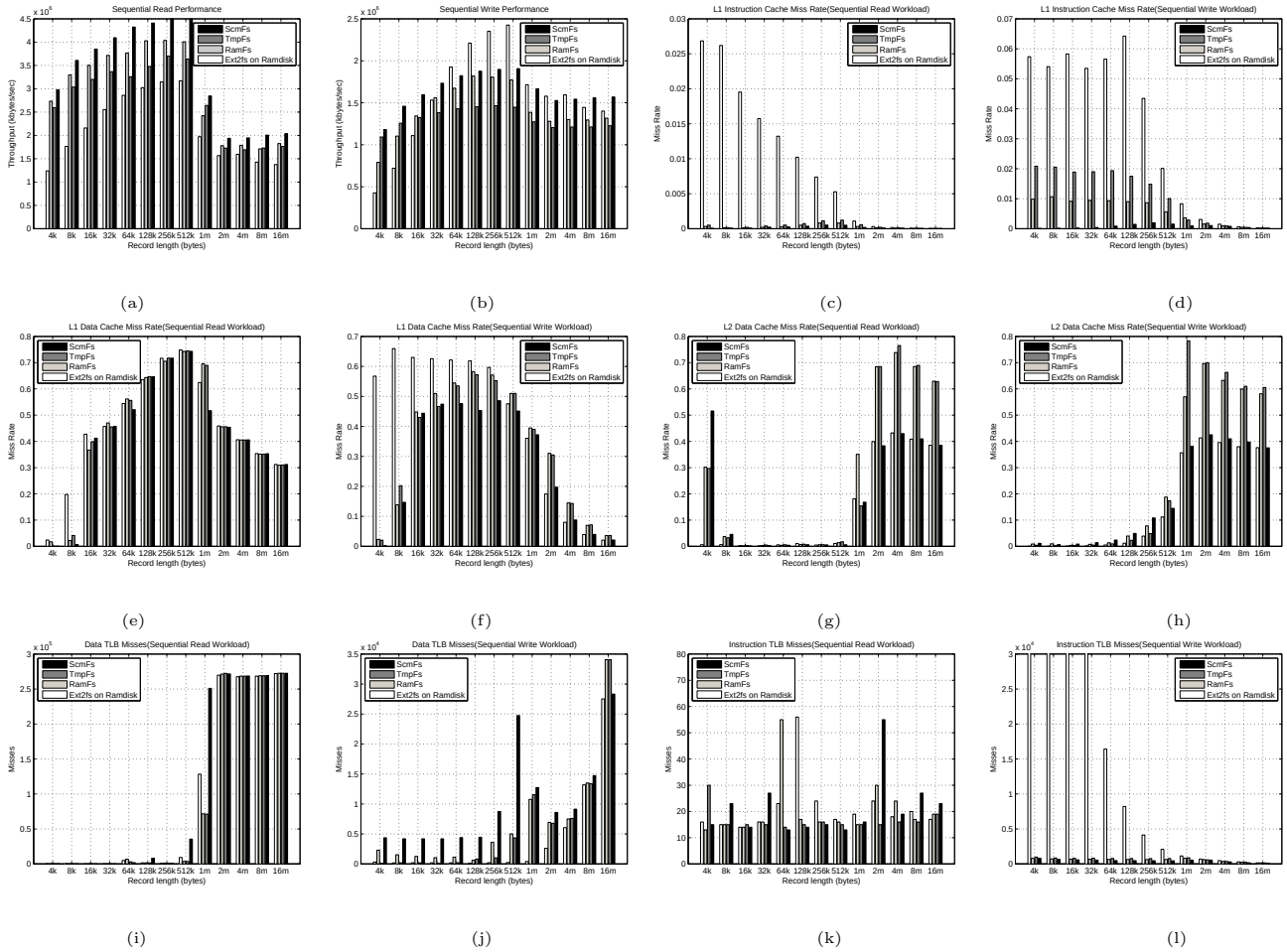


Fig. 24. IoZone results(Sequential workloads)

We notice that the memory based file systems, including RamFs, TmpFs and SCMFS, performs generally much better than Ext2Fs on Ramdisk. The reason is that Ext2 file system is created on the generic block layer and has much higher complexity than the memory based file systems, as we describe in Section IV.B. Simplicity of the hierarchy significantly decreases the size of instruction set. As shown in Fig.24(c,d,k,l) and Fig.25(c,d,k,l), memory based file systems had much lower instruction cache miss rate and instruction TLB misses than Ext2 file system.

We also notice that in the random/sequential write workload, Ext2 file system performs better than SCMFS when the record length was between 64k and 512k

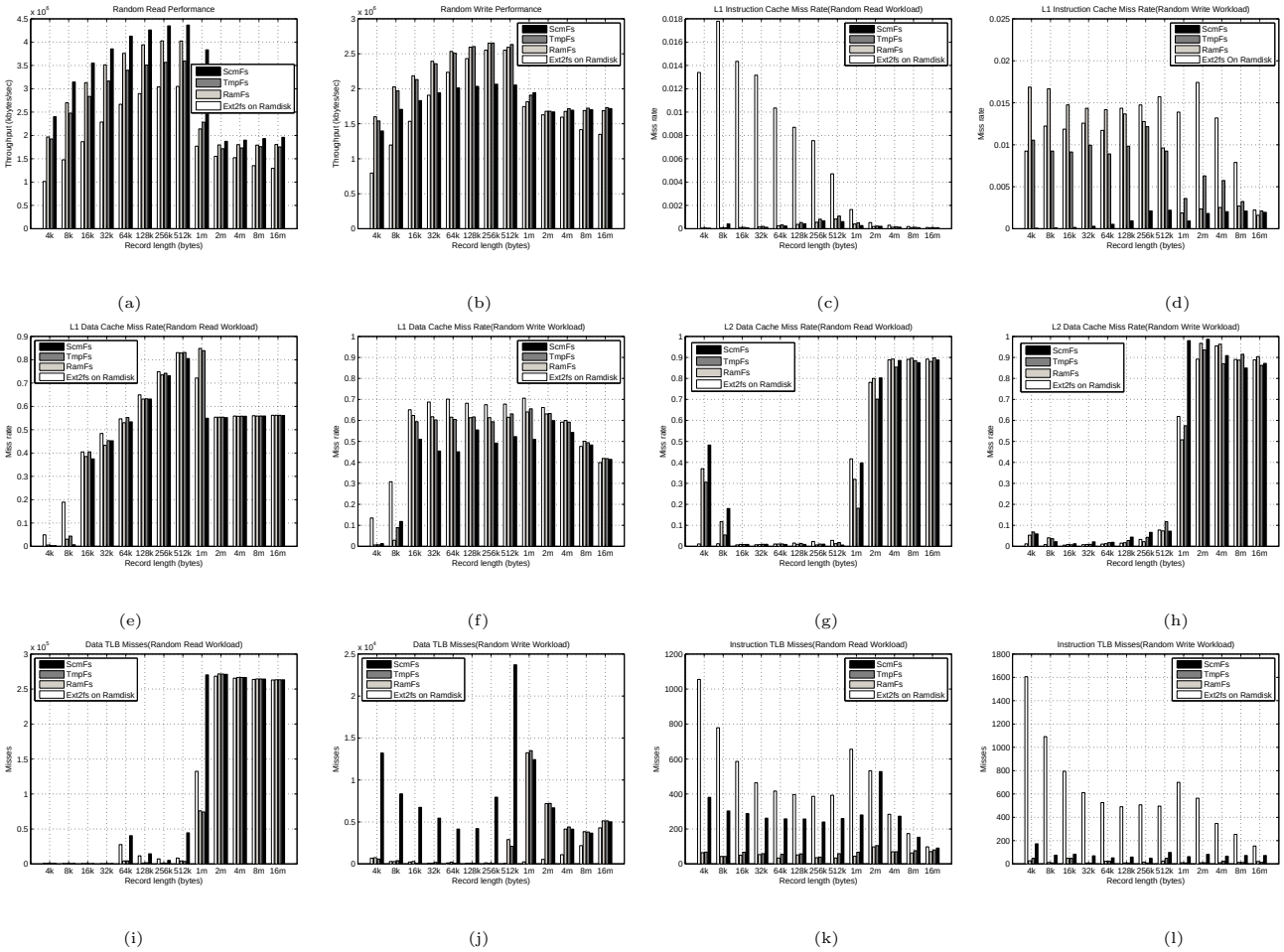
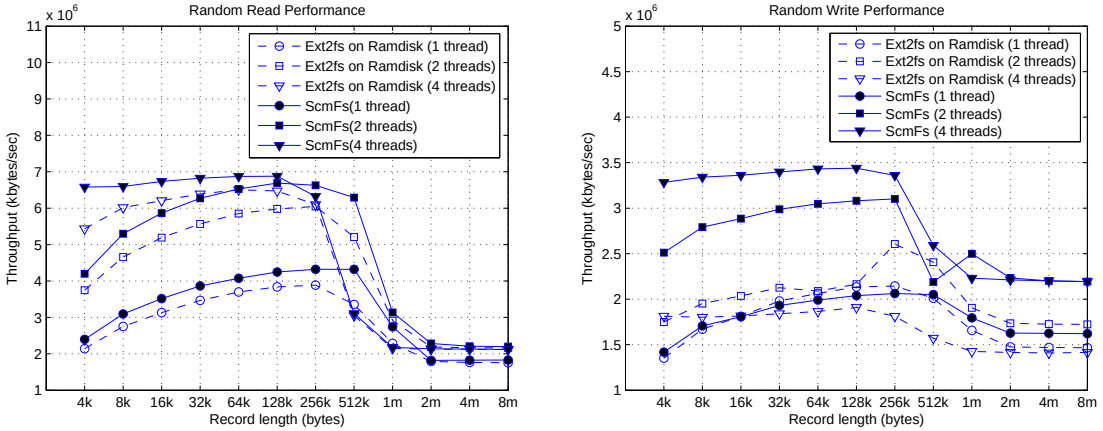


Fig. 25. IoZone results(Random workload)

bytes. We believe this is because SCMFS has much higher TLB misses than Ext2 file system, as shown in Fig.24(j) and Fig.25(j). The reason why SCMFS has much more TLB misses than the other file systems is we operate the data in SCMFS on virtual address space while the others employ device level linear address space. The linear address space is mapped by using a larger page size, such as 2MB. The results shown in Fig. 24 and 25 are for a single client and SCMFS significantly outperforms ext2fs when multiple clients are considered as shown below.

We also have done experiments with IoZone by using multiple threads. Fig.26 shows the result where we use normal read()/write() interfaces, while Fig.27 shows

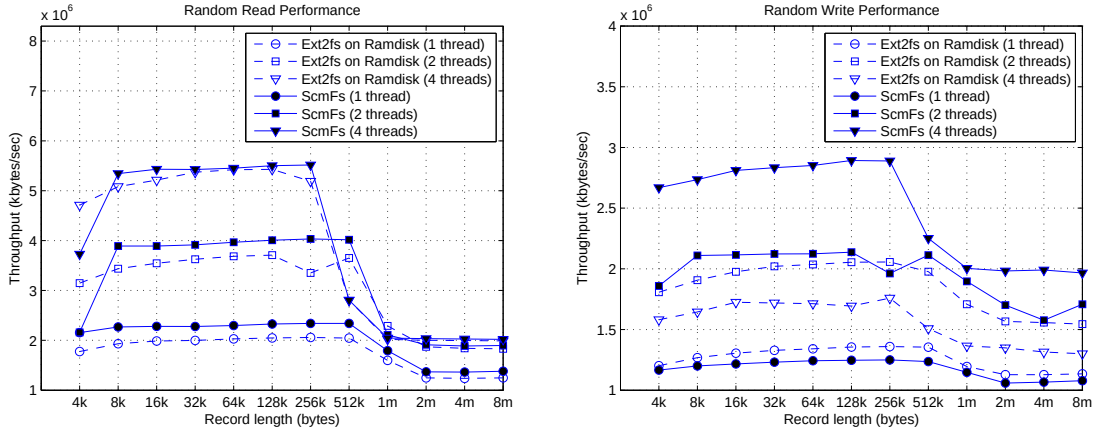


(a) (b)

Fig. 26. IoZone results with multi-thread (Random workload)

the results with mmap() interfaces. When we use mmap() interface, we enable XIP features for both the file systems. We can see, in both cases SCMFS performs better than Ext2Fs and obtains higher throughput with more threads. Another observation is that using mmap()/bcopy() does not perform beyond normal read()/write() interfaces. Through our investigation, we believe this is also caused by high TLB misses. In Ext2fs on Ramdisk, using mmap() will map the address into user address space, which is not using large page size. By using performance counters, we find that the number of TLB misses with read()/write() interfaces is only around 200(Ext2fs) or 4,000(SCMFS), while it is more than 2,000,000 with mmap() interfaces in the same workload.

It is observed from Fig. 26 that SCMFS obtains up to 7GB/s read throughput, about 70% of the memory bus bandwidth of 10GB/s on our system. It is observed that the read throughput generally saturates at twice the saturation throughput of writes, since writes require two memory operations compared to one operation on read requests.



(a) (b)

Fig. 27. IoZone results with multi-thread, using mmap (Random workload)

3. Postmark Results

We show the results of postmark in Fig.28. We use postmark to generate both read intensive and write intensive workloads. The file size in the workload is varied between 4k and 40k bytes. In each workload, we create 10,000 files under 100 directories and perform 400,000 transactions. We again use the PAPI library to investigate the detailed performance information.

In this test, we not only evaluate original SCMFS, but also evaluate SCMFS with pre-allocation mechanism, as described in Section IV.3, and with file system consistency, as described in Section IV.6. We do not include these with the IoZone workload, since IoZone workload operates on one file and does not exhibit much difference in performance with these mechanisms.

In the figures, we can see that the performance of all the file systems is close to each other in the Postmark workload. Postmark workload has many more metadata operations than the IoZone workload and hence these metadata operations dominate the file sysem performance. Since the files in the Postmark workload are small, the

possibility to use indirect blocks in Ext2fs is very low, and SCMFS doesn't have much advantage over Ext2fs. Through the results, we can see that SCMFS still has lower instruction cache miss rate than Ext2fs, especially in the write workload. Even though SCMFS has higher data TLB misses, SCMFS provides higher performance beyond ext2fs.

When we add the pre-allocation mechanism, the read performance of SCMFS drops slightly and the write performance improves. As we describe in Section IV.3, the pre-allocation mechanism helps reduce the time to allocate space for new data. In the last configuration, we add the support of file system consistency that is described in Section IV6. As anticipated, the performance of SCMFS drops significantly when write ordering issues are addressed. In the write workload, SCMFS still performs better than Ext2fs. In the read workload, even though the content of the files are not changed, the latest access time of each file needs to be updated. Each time the file metadata gets updated, the costly function "clflush_cache_range" is called to flush the cache. That is why the read performance decreases significantly. It is noted that Ext2fs on ramdisk does not support metadata consistency as SCMFS does. SCMFS does not rely on any new architectural mechanisms such as epochs in BPFS [64], which could reduce the costs of enforcing write ordering.

In the Postmark workload, the saturation throughputs are lower than observed earlier with the IoZone workload, because more metadata operations are involved in the Postmark workload. It is observed that the TLB misses are significantly higher in the Postmark workload compared to the IoZone workload.

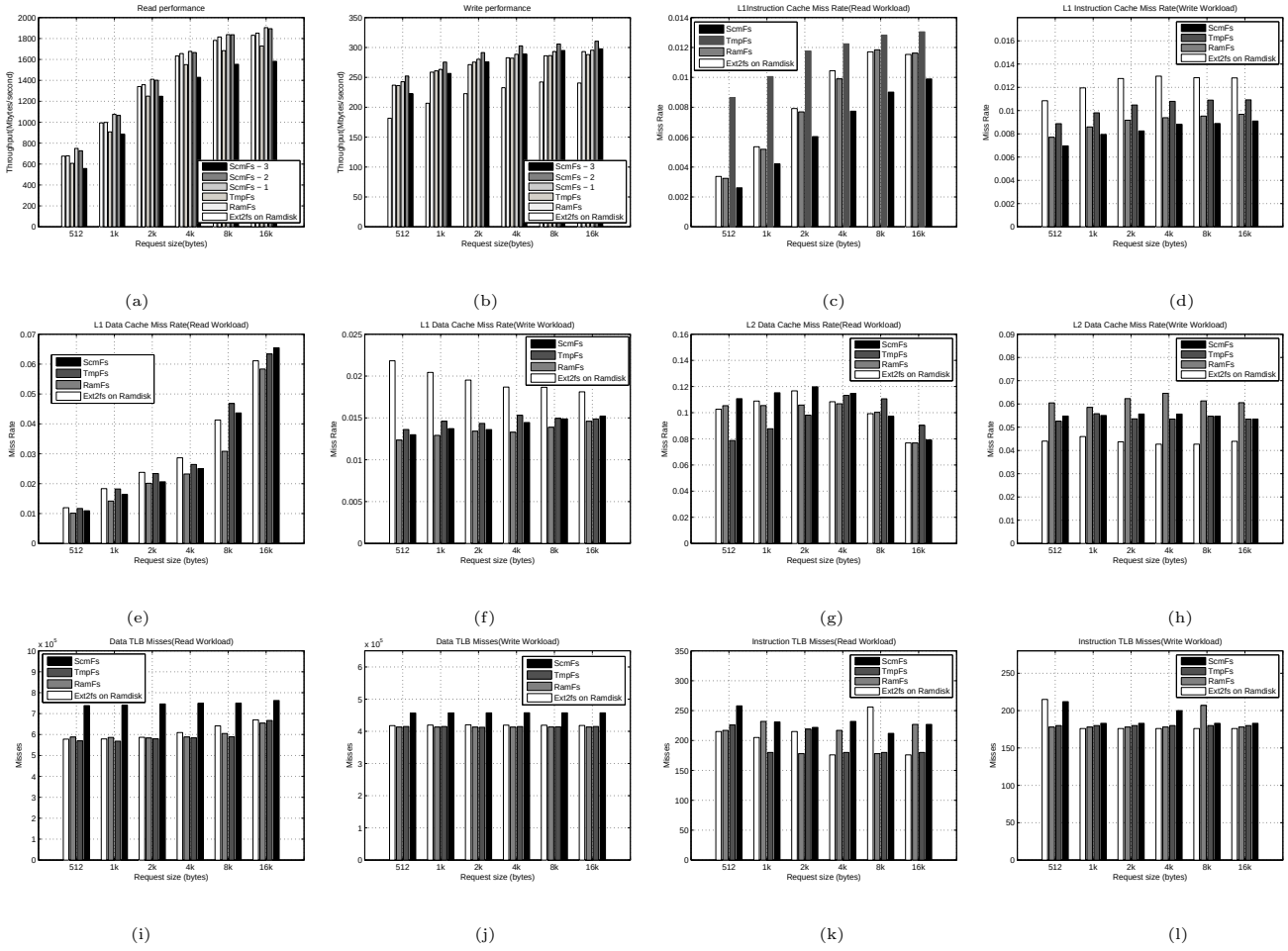


Fig. 28. Postmark results. SCMFS-1, original SCMFS. SCMFS-2, SCMFS-1 with space pre-allocation. SCMFS-3, SCMFS-2 with file system consistency.

CHAPTER V

RELATED WORK

A. Flash and SSD

Flash storage has received much attention recently. A number of studies have been done on wear leveling and maintaining uniformity of blocks [33, 34, 35]. Studies have been done on block management and buffer management [30, 31, 32, 37]. File system design issues for flash storage have been considered in [36], and a number of file systems have also been designed for flash devices([76, 77]). Flash translation layers (FTLs) are employed to mimic log structured file systems in current flash storage devices. Flash memory has been studied for reducing the power consumption in a storage system [44].

Performance issues of internal SSD organization and algorithms are well studied. For example, [36] presents how to build a high-performance SSD device. [31] focuses on improving random writes for SSD devices. [33] proposes a new page allocation scheme for flash based file systems. [34] works on garbage collection and [35] on wear-leveling for the flash based storage systems. Characterization of SSD organization parameters also has been studied in [57, 58]. [37] provides a very good survey of the current state of algorithms and data structures designed for flash storages. SSDs are employed to improve the system performance in [29, 30].

B. Hybrid Storage System

File system based approaches for flash and non-volatile storage can be found, for example in [42, 62, 43]. These systems can take file level knowledge (data types and their typical access behavior)into account which may not be available at the device

level. We consider a device level approach in this dissertation and we manage storage across the devices at a block or chunk level instead of at the file level. Second, our approach can potentially, leave parts of a file on the flash drive and parts of the file on a disk drive depending on the access patterns, which is not possible in file-based approaches. In addition, the device level approach may obviate the need for new file systems or changes to every file system that employs a flash drive. It is not our intent to argue that the device level approach is better than the file system approach or vice versa, but that they are based on different assumptions and lead to different design points and benefits.

Non-uniform striping strategies have been advocated before for utilizing disks of different capacity and performance [52, 53]. However, these strategies employ measurement characteristics that do not consider workloads or dynamic performance characteristics such as waiting times. Our approach in this dissertation is guided by dynamically measured performance measures and hence can adapt to changing workloads and configurations without preset allocation ratios.

Several papers recent look at issues in organizing SSDs and magnetic disks in hybrid storage systems. A file system level approach has been used in [54]. In [55], magnetic disks are used as write cache to improve the performance as well as extend the lifetime of SSD devices.

C. Data Migration

Data migration is considered previously in networked devices [6], in large data centers [4, 5]. This body of work considers optimization of data distribution over longer time scales than what we consider here. This work considers migration of datasets across different systems unlike our focus here on migration of data blocks within a single

system. Our work is inspired by much of this earlier work. Migration has also been employed in memory systems in multiprocessors, for example in [7]. Request migration has been employed to balance load in web based servers, for example in [8]. While we employ migration here for load balancing, we also used allocation as a vehicle for dynamically balancing the load across the different devices in our system.

Migration and caching are extensively studied in the context of memory and storage systems. Migration has been employed earlier in tape and disk storage systems [20] and in many file systems [9, 21] and in systems that employ dynamic data reorganization to improve performance [3, 5, 10, 11, 14, 17, 12, 26, 51, 72]. While some of these systems employed techniques to migrate data from faster devices to slower devices in order to ensure sufficient space is available on the faster devices for new allocations, migrated data in these systems tends to be "cold" data that hasn't been accessed for a while. HP's AutoRAID system considers data migration between a mirrored device and a RAID device [1]. In these systems, hot data is migrated to faster devices and cold data is migrated to slower devices to improve the access times of hot data by keeping it local to faster devices. When data sets are larger than the capacity of faster devices in such systems, thrashing may occur. Some of the systems detect thrashing and may preclude migration during such times [1]. Adaptive block reorganization to move hot blocks of data to specific areas of disks has been studied in the past [39, 40].

In our hybrid storage system, data can move in both directions from flash to disk and disk to flash for performance reasons. And, the realized performance in our system depends on read/write characteristics as well as the recency and frequency of access. Characteristics of data migration can be much different in our system compared to earlier hierarchical systems.

Aqueduct [14] also takes a control-theoretic approach to data migration among

storage devices without significantly impacting the foreground application work. Aqueduct uses I/O request response time as a performance measure, but there are significant differences from our work. First, Aqueduct is guided by static policy decisions of the system administrator unlike our dynamic choice of blocks during run time. Data distribution and migration issues are considered in [15, 16, 17] as a result of configuration changes due to additions and removal of disks in large scale storage systems. Adaptive file set migration for balancing the metadata workload among servers in shared-disk file systems is considered in [18]. Observed latencies are used for migration. Our work is different from this work: (a) Our system is dealing with migrating block-level data in contrast to file-level data (e.g., file sets), and (b) we consider read and write requests separately.

D. Non-volatile Byte-addressable Memory

BPFS [64] is proposed as a file system designed for non-volatile byte-addressable memory, which uses shadow paging techniques to provide fast and consistent updates. It also requires architectural enhancements to provide new interfaces for enforcing a flexible level of write ordering. Our file system, SCMFS aims to simplify the design and eliminate the unnecessary overhead to improve the performance. DFS[56] is the most similar file system to SCMFS. DFS incorporates the functionality of block management in the device driver and firmware to simplify the file system, and also keeps the files contiguous in a huge address space. It is designed for a PCIe based SSD device by FusionIo, and relies on specific features in the hardware.

A number of projects have previously built storage systems on non-volatile memory devices. Rio [62] and Conquest [42] use the battery-backed RAM in the storage system to improve the performance or provide protections. Rio uses the battery-

backed RAM to store the file cache to avoid flushing dirty data, while Conquest uses it to store the file system metadata and small files. In the eNVy storage system [43], the flash memory is attached to the memory bus to implement a non-volatile memory device. To make this device byte addressable they designed a special controller with a battery-backed RAM buffer. Our work assumes that nonvolatile memory is large enough for both data and metadata and focuses on leveraging memory management infrastructure in the system. A data structure level approach to achieve data consistency on non-volatile memory is described in [79].

Solutions have been proposed to speed up memory access operations, to reduce writes, and for wear-leveling on PCM devices. Some of these solutions improve the lifetime or the performance of PCM devices at the hardware level. Some of them use a DRAM device as a cache of PCM in the hierarchy. Modifications to the memory controller to eliminate unnecessary bit writes have been proposed [66, 75]. [73, 67, 74, 68] proposed several wear leveling schemes to protect PCM devices from normal applications and even malicious attacks. Since our SCMFS is implemented on the file system layer, all the hardware techniques can be integrated with our file system to provide better performance or stronger protection.

CHAPTER VI

CONCLUSION AND FUTURE WORK

A. Hybrid Storage System

In Chapter II, we study a hybrid storage system employing both flash and disk drives. We propose a measurement-driven migration strategy for managing storage space in such a system in order to exploit the read/write performance asymmetry of these devices. Our approach extracts the read/write access patterns and request size patterns of different blocks and matches them with the read/write advantages of different devices. We show that the proposed approach is effective, based on realistic experiments on a Linux testbed, employing three different benchmarks. The results indicate that the proposed measurement-driven migration can be beneficial in such a system. Our study also provides a number of insights into the different performance aspects of flash storage devices and allocation policies in such a hybrid system. Our work show that the read/write characteristics of the workload have a critical impact the performance of such a hybrid storage system.

In Chapter II, we consider several allocation policies: initially all the data on flash, all the data on disk, striping, allocation partially determined by the observed device performance characteristics and metadata on the flash device. Random allocation has been proposed by others [2, 13] to achieve load balancing with heterogeneous devices. We will consider random and other allocation schemes that employ file system or application level hints, in the future.

Our results show that it is possible to detect the read/write access patterns and the request sizes to migrate the blocks to the appropriate devices to improve the device performance in a hybrid system. The measurement-drive approach is shown

to be flexible enough to adapt to different devices and different workloads.

In Chapter III, we consider block allocation and migration as a means for balancing the response times of devices across a workload. Dynamically observed performance of the devices is used to guide these decisions. In this part of work, we only consider the aggregate performance of the devices irrespective of the nature of the requests (reads/writes, small/large etc.). Our policy allow data to be allocated to slower devices before the space on the faster devices is exhausted. Our performance-driven allocation and migration improves performance of the hybrid storage system, both in improving latency of the requests and the throughput of the requests. We also show that our policy adapts to different configurations of the storage system under different workloads.

In the future, we plan to consider the nature of the requests and the performance of the requests on different devices to improve performance further, in combination with the allocation based policy presented in this dissertation.

B. SCMFS

In Chapter IV, we present the design of SCMFS, a new file system specially for the storage class memory. SCMFS utilizes the existing memory management module in the operating system to help the block management, and keeps the space for each file always contiguous in the virtual address space. The design of SCMFS simplifies its implementation, and improves the performance, especially for small size requests. However, this file system has some disadvantages and limits, and we will consider them in our future work.

In the current experiment environment, the size of simulated SCM is very small(4GB), so the size of required mapping table is also very small. The size of mapping table

will become very large when the SCM is scaled to tens or even hundreds of Gigabytes. The large mapping table will significantly increase the time to mount the entire file system. To address this problem, we can delay the memory mapping process, which means only the virtual address space for metadata and inode table will be mapped during the time of mounting the file system. All the other address spaces will be mapped in background after the file system is mounted. If a request to an unmapped file is received, a page fault will be triggered. In the page fault handler, we will read the SCM mapping table and map the address. To achieve this, we also need to maintain bitmaps (or other compressed data structures) for MM to indicate those physical addresses and virtual addresses that are already used. The bitmaps are also loaded to MM module during the mount procedure. Another potential issue the large scale of SCM may cause is that the TLB cannot cover enough range of memory and results in many TLB misses. We may use superpages to increase the coverage of TLB and decrease the TLB misses that require expensive address translations.

In current implementation, we reserve a large virtual space for SCMFS and do not consider the extreme case of fragmentation, in which there is enough physical space but there is no contiguous virtual space for a new file. In the future work, we consider to add defragmentation of virtual address space into the thread of garbage collection.

Most SCM technologies have limits on write cycles to individual memory locations. In our current work, we do not incorporate any algorithms for wear leveling of the underlying SCM. We plan to include this as part of allocation process in the future.

REFERENCES

[1] J. Wilkes, R. A. Golding, C. Staelin, T. Sullivan. The HP AutoRAID Hierarchical Storage System. *ACM Trans. Comput. Syst.* 14(1): 108-136 (1996).

[2] IBM Almaden Research Center. Collective Intelligent Bricks (CIB). *http://www.almaden.ibm.com/StorageSystems/autonomic_storage/CIB/index.shtml*, 2001.

[3] B. Gavish and O. Sheng. Dynamic File Migration in Distributed Computer Systems. *Commun. ACM*, pages: 177-189, 1990.

[4] E. Anderson et al., Hippodrome: Running circles around storage administration. *USENIX FAST Conf.*, Jan. 2002.

[5] L. Yin, S. Uttamchandani and R. Katz. SmartMig: Risk-modulated Proactive Data Migration for Maximizing Storage System Utility. In *Proc. of IEEE MSST* (2006).

[6] S. Kang and A. L. N. Reddy. User-centric data migration in networked storage devices. Proc. of IPDPS, Apr. 2008.

[7] E. P. Markatos and T. J. LeBlanc. Load Balancing vs. Locality Management in Shared-Memory Multiprocessors. *Tech. Report: TR399, University of Rochester*, 1991.

[8] V. S. Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenepoel, and E. M. Nahum. Locality-Aware Request Distribution in Cluster-based Network Servers. In *Proc. of ASPLOS* (1998).

[9] E. Miller and R. Katz. An Analysis of File Migration in a UNIX Supercomputing Environment. In *Proc. of USENIX* (1993).

[10] M. Lubeck, D. Geppert, K. Nienartowicz, M. Nowak and A. Valassi. An Overview of a Large-Scale Data Migration. In *Proc. of IEEE MSST* (2003).

[11] R. M. English and A. A. Stepanov. Loge: A Self-Organizing Disk Controller. In *Proc. of USENIX Winter* (1992).

[12] R. Wang, T. E. Anderson, and D. A. Patterson. Virtual Log-Based File Systems for a Programmable Disk. In *Proc. of OSDI* (1999).

[13] S. Ghemawat, H. Gobioff and S-T. Leung. The Google File system. In *Proc. of SOSP* (2003).

[14] C. Lu, G. A. Alvarez and J. Wilkes Aqueduct: online data migration with performance guarantees In *Proc. of FAST* (2002).

[15] A. Brinkmann, K. Salzwedel and C. Scheideler. Efficient, distributed data placement strategies for storage area networks (extended abstract). In *Proc. of SPAA* (2000).

[16] Y. Saito, S. Frølund, A. Veitch, A. Merchant and S. Spence. FAB: building distributed enterprise disk arrays from commodity components. In *Proc. of ASPLOS* (2004).

[17] R. Honicky and E. Miller. A Fast Algorithm for Online Placement and Reorganization of Replicated Data. In *Proc. of IPDPS* (2003).

[18] C. Wu and R. Burns. Handling Heterogeneity in Shared-Disk File Systems. In *Proc. of SC* (2003).

[19] V. Borkar and P. R. Kumar. Dynamic Cesaro-Wardrop Equilibration in Networks. *IEEE Trans. on Automatic Control* 48(3): 382-396 (2003).

[20] A. J. Smith. Long Term File Migration: Development and Evaluation of Algorithms. *Communications of ACM* 24(8): 521-532 (1981).

[21] V. Cate and T. Gross. Combining the Concepts of Compression and Caching for a Two-level File System. In *Proc. of ASPLOS* (1991).

[22] Standard Performance Evaluation Corporation. SPEC SFS97_R1 V3.0 *http://www.spec.org/osg/sfs97r1.*

[23] R. P. Martin and D. E. Culler. NFS Sensitivity to High Performance Networks. In *Proc. of SIGMETRICS* (1999).

[24] L. Peterson and B. Davies. Computer Networks: A Systems Approach. Morgan Kauffman Publishers (2000).

[25] IOZONE file system benchmark. http://www.iozone.org/, accssessed 01/2011.

[26] S. D. Carson and P. F. Reynolds, Adaptive disk reorganization. *Tech. Rep.: UMIACS-TR-89-4, University of Maryland, College Park, Maryland*, January 1989.

[27] M. McKusick and W. Joy and S. Leffler and R. Fabry. A Fast File System for UNIX. *ACM Transactions on Computer Systems, 2(3):181-197*, August 1984.

[28] G. Ganger and Y. Patt. Metadata Update Performance in File Systems. *USENIX Symposium on Operating Systems Design and Implementation, pp. 49-60*, November 1994.

[29] I. Koltsidas and S. Viglas. Flashing up the storage layer. *Proc. of VLDB*, Aug. 2008.

[30] T. Kgil, D. Roberts and T. Mudge. Improving NAND Flash Based Disk Caches. *Proc. of ACM Int. Symp. Computer Architecture*, June 2008.

[31] H. Kim and S. Ahn. BPLRU: A buffer management scheme for improving random writes in flash storage. *Proc. of USENIX FAST Conf.*, Feb. 2008.

[32] H. Kim and S. Lee. An effective flash memory manager for reliable flash memory space management. *IEICE Trans. on Information systems.*, June 2002.

[33] S. Baek et al. Uniformity improving page allocation for flash memory file systems. *Proc. of ACM EMSOFT*, Oct. 2007.

[34] J. Lee et al. Block recycling schemes and their cost-based optimization in NAND flash memory based storage system. *Proc. of ACM EMSOFT*, Oct. 2007.

[35] Li-Pin Chang. On efficient wear leveling for large-scale flash-memory storage systems. *Proc. of ACM SAC Conf.*, Mar. 2007.

[36] A. Birrell, M. Isard. C. Thacker and T. Wobber. A design for high-performance flash disks. *ACM SIGOPS Oper. Syst. Rev.*, 2007.

[37] E. Gal and S. Toledo. Algorithms and data structures for flash memories. *ACM Computing Surveys*, June 2005.

[38] S. Brandt, L.Xue, E. Miller and D.Long. Efficient metadata management in distributed storage systems. *Proc. of IEEE Mass Storage Symp.*, 2003.

[39] S. Akyurek and K. Salem. Adaptive block rearrangement. *Computer Systems*, vol.13, no.2, pages 89-121, 1995.

[40] R. Tewari, R. King, D. Kandlur and D. Dias. Placement of Multimedia blocks on zoned disks. *Proc. of Multimedia Computing and Networking*, Jan. 1996.

[41] Jeffrey Katcher. Postmark: A New File System Benchmark. *http://www.netapp.com/tech library/3022.html.*

[42] A.-I. A. Wang, P. Reiher, G. J. Popek, and G. H.Kuenning. Conquest: Better performance through a disk/persistent-ram hybrid file system. *In Proceedings of the USENIX Annual Technical Conference*, June 2002.

[43] M.Wu and W. Zwaenepoel. envy: A non-volatile, main storage system. *Proc. of ASPLOS*, 1994.

[44] L. Useche, J. Guerra, M. Bhadkamkar, M. Alarcon and R. Rangaswami. EXCES: External caching and energy saving storage systems. *Proc. of HPCA*, 2008.

[45] X. Wu and A. L. N. Reddy. Managing storage space in a flash and disk hybrid storage system. IEEE MASCOTS Conf., 2009.

[46] X. Wu and A. L. N. Reddy. Exploiting concurrency to improve latency and throughput in a hybrid storage system. IEEE MASCOTS Conf., 2010.

[47] Dbench benchmark. Available from ftp://samba.org/pub/tridge/dbench/

[48] J. G. Wardrop. Some theoretical aspects of road traffic research. Proc. of Inst. of Civil Engineers, part II, vol.1, 1952.

[49] F. Shu and N. Obr. Data set management commands proposal for ATA8-ACS2. www.t13.org, Dec. 2007.

[50] A. W. Lueng, S. Pasupathy, G. Goodson and E. L. Miller. Measurement and Analysis of large-scale network file system workloads. Usenix Technical Conf., 2008.

[51] M. Bhadkamkar, J. Guerra, L. Useche, S. Burnett, J. Liptak, R. Rangaswami, and V. Hristidis. BORG: Block-reORGanization for Self-Optimizing Storage Systems. Proc. of USENIX FAST, 2009.

[52] S. Kashyap and S. Khuller. Algorithms for non-uniform sized data placement on parallel disks. Foundations of software technology and theoretical computer science. Springerlink., 2003.

[53] J.Wolf, H.Shachnai and P. Yu. DASD Dancing: A disk load-balancing optimization scheme for on-demand video-on-demand computer systems. ACM SIGMETRICS, 1995.

[54] J. Garrison, A. L. N. Reddy. Umbrella File system: Storage management across heterogeneous devices. Texas A&M University Tech. Report, Decc. 2007.

[55] G. Soundararajan, V. Prabhakaran, M. Balakrishnan and T. Wobber. Extending SSD lifetimes with disk-based write caches. Proc. of USENIX FAST, 2010.

[56] W. K. Josephson, L.A. bongo, D. Flynn and Kai Li. DFS: A new file system for virtualized flash storage. Proc. of USENIX FAST, 2010.

[57] J-H. Kim, D. Jung, J-S. Kim and J. Huh. A methodology for extracting performance parameters in solid state disks (SSDs). Proc. of IEEE MASCOTS, 2009.

[58] F. Chen, D. Koufaty and X. Zhang. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. Proc. of ACM

SIGMETRICS, 2009.

[59] S. Raoux, G. W. Burr, M. J. Breitwisch, C. T. Rettner, Y. -c. Chen, R. M. Shelby, M. Salinga, D. Krebs, S. -h. Chen, H. -l. Lung and C. H. Lam. Phase-change random access memory: A scalable technology. 2008

[60] Philip J. Mucci, Shirley Browne, Christine Deane and George Ho. PAPI: A Portable Interface to Hardware Performance Counters. In Proceedings of the Department of Defense HPCMP Users Group Conference, 1999

[61] D. B. Strukov, G. S. Snider, D. R. Stewart and R. S. Williams. The missing memristor found. Nature, vol.453, pp. 80-83, May 2008.

[62] Peter M. Chen, Wee Teck Ng, Gurushankar Rajamani and Christopher M. Aycock. The Rio File Cache: Surviving Operating System Crashes. In Proc. 7th Intl. Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS,1996.

[63] Peter Snyder. tmpfs: A virtual memory file system. In Proceedings of the Autumn 1990 European UNIX Users' Group Conference, pp. 241–248, 1990.

[64] Engin Ipek, Jeremy Condit, Benjamin Lee, Edmund B. Nightingale, Doug Burger, Christopher Frost and Derrick Coetzee. Better I/O Through Byte-Addressable, Persistent Memory. 2009.

[65] Moinuddin K. Qureshi, Vijayalakshmi Srinivasan and Jude A. Rivers. Scalable high performance main memory system using phase-change memory technology. In International Symposium on Computer Architecture (ISCA), pp. 24-33, 2009

[66] Lee, B.C.,Ping Zhou, Jun Yang, Youtao Zhang, Bo Zhao, Ipek, E., Mutlu, O. and Burger, D. Phase-Change Technology and the Future of Main Memory. IEEE

Micro, 2010.

[67] M.Qureshi, John Karidis, Michele Franceschini, Vijayalakshmi Srinivasan and Luis Lastras and Bulent Abali. Enhancing Lifetime and Security of PCM-based Main Memory with Start-Gap Wear Leveling. The 42nd Annual IEEE/ACM MICRO, 2009.

[68] Ping Zhou, Bo Zhao, Jun Yang and Youtao Zhang. A Durable and Energy Efficient Main Memory using Phase Change Memory Technology. The 36th Annual IEEE/ACM ISCA, 2009.

[69] M.Qureshi and Franceschini, M.M. and Lastras-Montano,L.A. Improving Read Performance of Phase Change Memories Via Write Cancellation and Write Pausing. The IEEE 16th HPCA, 2010.

[70] Numonyx. The basics of phase change memory (PCM) technology: A new class of non-volatile memory., 2008.

[71] Bedeschi, F. and Fackenthal, R. and Resta, C. and Donze, E.M. and Jagasivamani, M. and Buda, E.C. and Pellizzer, F. and Chow, D.W. and Cabrini, A. and Calvi, G. and Faravelli, R. and Fantini, A. and Torelli, G. and Mills, D. and Gastaldi, R. and Casagrande, G., A Bipolar-Selected Phase Change Memory Featuring Multi-Level Cell Storage. IEEE Journal of Solid-State Circuits, Jan.2009.

[72] Dhiman, G. and Ayoub, R. and Rosing, T. PDRAM: A Hybrid PRAM and DRAM Main Memory System. The 46th ACM/IEEE DAC, 2009.

[73] Seong, Nak Hee and Woo, Dong Hyuk and Lee, Hsien-Hsin S. Security refresh: prevent malicious wear-out and increase durability for phase-change memory

with dynamically randomized address mapping. The 37th Annual IEEE/ACM ISCA, 2010.

[74] M.Qureshi and Andre Seznec and Luis Lastras and Michele Franceschini. Practical and Secure PCM Systems by Online Detection of Malicious Write Streams. The IEEE 17th HPCA, 2011.

[75] Lee, Benjamin C. and Ipek, Engin and Mutlu, Onur and Burger, Doug. Architecting Phase Change Memory as a Scalable Dram Alternative. The 36th Annual IEEE/ACM ISCA, 2009.

[76] WOODHOUSE, D. JFFS: The journalling flash file system. In Ottawa Linux Symposium, RedHat Inc , 2001.

[77] YAFFS, http://www.yaffs.net/.

[78] Advanced Configuration and Power Interface Specification 3.0.

[79] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, and Roy H. Campbell. Consistent and Durable Data Structures for Non-Volatile Byte-Addressable Memory. In Proceedings of the 9th Usenix Conference on File and Storage Technologies (FAST), 2011.

VITA

Xiaojian Wu received his Bachelor of Science degree and Master of Science degree in Electrical Engineering from Huazhong University of Science & Technology, Wuhan, China, in July 1998 and July 2003, respectively. He entered the Ph.D. program in Computer Engineering at Texas A&M University in August 2007. During 2003-2006, he worked as a software engineer at Intel Asia-Pacific R&D in Shanghai, China. His research interests include storage systems, cloud systems and semantic web. He is a member of IEEE.

The typist for this thesis was Xiaojian Wu.