INTEGRATION OF NON-VOLATILE MEMORY INTO STORAGE HIERARCHY

A Dissertation

by

SHENG QIU

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

DOCTOR OF PHILOSOPHY

Chair of Committee,	A.L.Narasimha Reddy
Committee Members,	Riccardo Bettati
	Krishna Narayanan
	Paul Gratz
Head of Department,	Chanan Singh

December 2013

Major Subject: Computer Engineering

ABSTRACT

In this dissertation, we present novel approaches for integrating non-volatile memory devices into storage hierarchy of a computer system. There are several types of non-volatile memory devices, such as flash memory, Phase Change Memory (PCM), Spin-transfer torque memory (STT-RAM). These devices have many appealing features for applications, however, they also offer several challenges. This dissertation is focused on how to efficiently integrate these non-volatile memories into existing memory and disk storage systems. This work is composed of two major parts.

The first part investigates a main-memory system employing Phase Change Memory instead of traditional DRAM. Compared to DRAM, PCM has higher density and no static power consumption, which are very important factors for building large capacity memory systems. However, PCM has higher write latency and power consumption compared to read operations. Moreover, PCM has limited write endurance. To efficiently integrate PCM into a memory system, we have to solve the challenges brought by its expensive write operations. We propose new replacement policies and cache organizations for the last-level CPU cache, which can effectively reduce the write traffic to the PCM main memory. We evaluated our design with multiple workloads and configurations. The results show that the proposed approaches improve the lifetime and energy consumption of PCM significantly.

The second part of the dissertation considers the design of a data/disk storage using non-volatile memories, e.g. flash memory, PCM and nonvolatile DIMMs. We consider multiple design options for utilizing the nonvolatile memories in the storage hierarchy. First, we consider a system that employs nonvolatile memories such as PCM or nonvolatile DIMMs on memory bus along with flash-based SSDs. We

propose a hybrid file system, NVMFS, that manages both these devices. NVMFS exploits the nonvolatile memory to improve the characteristics of the write workload at the SSD. We satisfy most small random write requests on the fast nonvolatile DIMM and only do large and optimized writes on SSD. We also group data of similar update patterns together before writing to flash-SSD, as a result, we can effectively reduce the garbage collection overhead. We implemented a prototype of NVMFS in Linux and evaluated its performance through multiple benchmarks.

Secondly, we consider the problem of using flash memory as a cache for a disk drive based storage system. Since SSDs are expensive, a few SSDs are designed to serve as a cache for a large number of disk drives. SSD cache space can be used for both read and write requests. In our design, we managed multiple flash-SSD devices directly at the cache layer without the help of RAID software. To ensure data reliability and cache space efficiency, we only duplicated dirty data on flash-SSDs. We also balanced the write endurance of different flash-SSDs. As a result, no single SSD will fail much earlier than the others.

Thirdly, when using PCM-like devices only as data storage, it's possible to exploit memory management hardware resources to improve file system performance. However, in this case, PCM may share critical system resources such as the TLB, page table with DRAM which can potentially impact PCM's performance. To solve this problem, we proposed to employ superpages to reduce the pressure on memory management resources. As a result, the file system performance is further improved.

ACKNOWLEDGEMENTS

I am sincerely thankful to my advisor, Dr. Reddy, for his patience on guiding and helping me in my research, as well as my life. I would like to thank my committee members, Dr. Bettati, Dr. Gratz and Dr. Narayanan, for their precious time to review my dissertation and gave me valuable feedback. I would like to thank all my friends, without their help I might not solve some challenges while studying and living abroad. Finally, I would like to thank my parents for their love throughout my whole life. Without their support, it would have been impossible for me to finish my Ph.D. degree.

TABLE OF CONTENTS

LIST OF FIGURES

FIGURE Page

LIST OF TABLES

TABLE Page

1. INTRODUCTION

Within existing storage hierarchy of a computer system, main memory and disk storage are two important components as shown in figure 1.1. The main memory is normally composed of Dynamic Random Access Memory (DRAM) which is volatile and supports fast random read and write accesses. For personal desktop or laptop, it's sufficient to have several GBs' memory, while large computing servers usually require much larger capacity memory systems. However, DRAM technology is now hitting hard power and capacity constraints that will limit its future process technology scaling [30]. The technology scaling constraints for DRAM memory recently led to the emergence of Phase Change Memory as an alternative form of main memory in future processor designs [67]. We will introduce this in section 1.1.

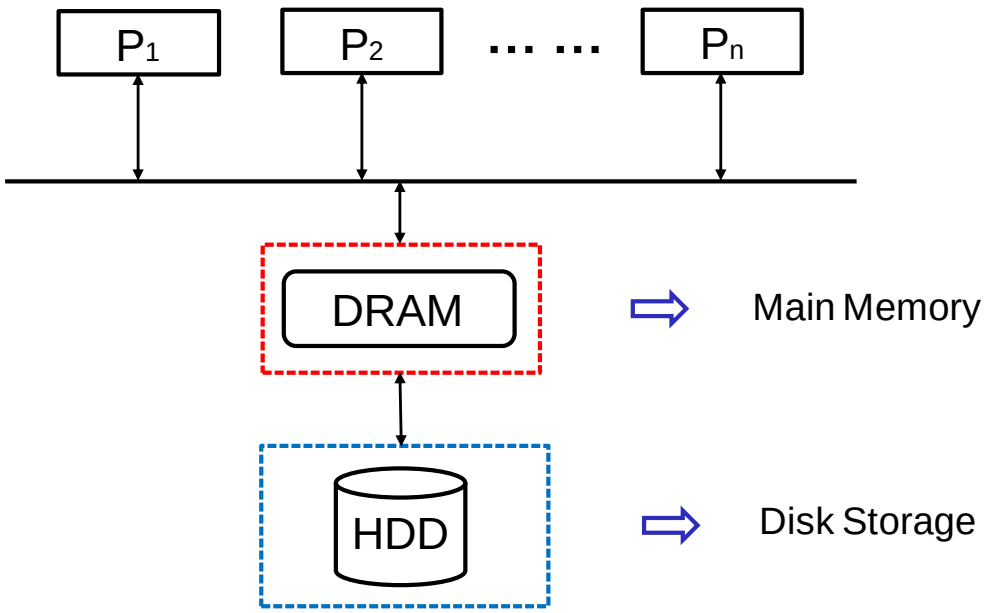


Figure 1.1: Main memory and disk storage of a computer system.

Another important component is the data/disk storage. When data are not in memory, we need to first fetch them from disk storage. Moreover, to ensure data endurance, existing memory systems need to write data updates to the disk storage within a short time since DRAM is volatile and cannot sustain power failure. Therefore, the read and write speed of the disk storage can directly affect the overall performance of the whole system. Traditional hard disk has low performance for random operations, which led to the emergence of solid state disk (SSD) and other emerging nonvolatile memories as an alternative or a complementary building block for storage. We will describe it in section 1.2.

1.1 Main memory technology

Dynamic random access memory (DRAM) is the predominant technology for main memory in current processor systems. DRAM technology, however, is now hitting hard power and capacity constraints that will limit its future process technology scaling [30]. Phase-Change Memory (PCM) has been proposed as an alternate technology for processor memory systems [45, 63, 88, 61, 65].

PCM technology utilizes a class of materials known as chalcogenides. An alloy of Germanium, Antimony and Tellurium ($Ge_2Sb_2Te_5$) is one such alloy used by some manufacturers [57, 3]. These materials can exist in two different states, either crystalline or amorphous. By heating the chalcogenides, the phase (or the state) can be changed or reversed between amorphous and crystalline states. The material exhibits high resistivity in amorphous state and low resistivity in crystalline state allowing binary states of 0/1 to be represented. While DRAM, is volatile and must be refreshed, leading to a constant power consumption even when idle, PCM is nonvolatile and retains the state even when the power is off.

Figure 1.2 [18] shows a diagram of a single PCM device cell and the circuit used

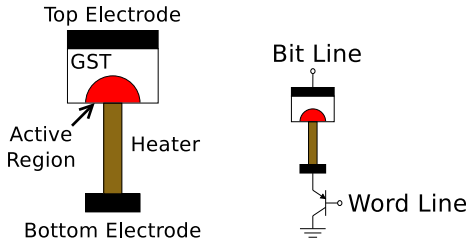


Figure 1.2: PCM device and circuit. [18]

to construct a memory cell from a PCM device. A PCM cell consists of two electrodes with a layer of chalcogenide in between. As shown in Figure 1.2, a resistive heating element extends from the bottom electrode to the layer of chalcogenide. Current injected through the heating element changes the state of the chalcogenide through local heating. The density of PCM arrays is expected to scale with process technology better than the capacitance used in a DRAM cell as the semiconductor technology progresses for two reasons: 1) As the access transistor in capacitive DRAM cells shrink, their sub-threshold leakage increases, eventually making further shrinks impractical. Resistive PCM cells do not rely upon capacitive charge to determine state [45]. 2) Future PCM cells promise the capacity of storing more than one bit per cell further increasing their density versus DRAM cells [3].

Since writing a PCM cell involves thermal energy, writes take higher energy. Table 1.1 details the read and write power for PCM versus DRAM in 78nm technology. A write to PCM typically requires more power compared to an equivalent write to DRAM. On the other hand, reads to PCM are less expensive in power consumption than a DRAM. As this data shows, while PCM is 2.5x more efficient than DRAM for reads, writes are 4x more expensive for PCM. The table also shows, while DRAM row read latency can be about 15ns, the read latency of PCM can range from 15-28ns. Similarly a DRAM row write latency can be about 20ns, the write latency in

PCM is about 150ns [18].

As Table 1.1 shows, both the power consumption and latency characteristics of read/write operations, PCM exhibits asymmetric performance characteristics, with reads being much more efficient than writes. These characteristics require attention when designing a memory system using PCM. The asymmetry of read/write characteristics require that read/write accesses be differently optimized.

Table 1.1: PCM vs DRAM characteristics [18]

	PCM	**DRAM**
Row read power	78mW	210mW
Row write power	773mW	200mW
Initial row read latency	28ns	15ns
Row write latency	150ns	20ns
Same row read/write latency	15ns	15ns

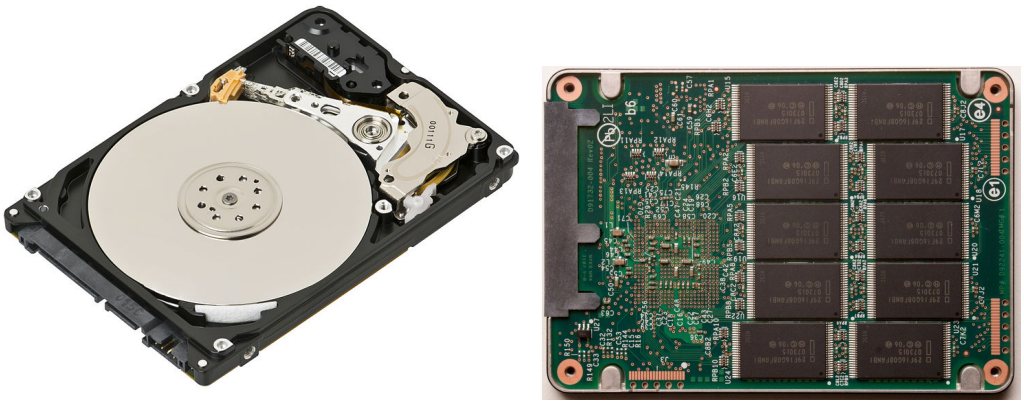
As introduced above, PCM has different characteristics than DRAM. PCM is expected to be available in higher densities than DRAM in the future; PCM memory is also non-volatile. These characteristics of PCM have spurred novel memory hierarchy designs. Relative to DRAM, PCM memory, however, introduces some new design constraints. PCM memory's read and write access characteristics are asymmetric. Reads are more efficient in access time and power consumption than writes. PCM memory cells also have a limited number of write cycles before they wearout. Hence, a memory system employing PCM needs to address this asymmetry in its design.

In Chapter 2, we present our design for employing PCM in memory system considering the impact of this asymmetry. We propose low-complexity techniques, utilizing existing cache memory systems, to substantially improve the lifetime durability and

energy consumption of PCM main memory.

1.2 Disk storage technology

Traditional data/disk storage is built using hard disks. Hard disk stores data on rapidly rotating disks (platters) coated with magnetic material. Hard disks, shown in Figure 1.3a, can retain data even when powered off. The sequential read and write operations are much faster than the random ones due to the so called seek latency. The factors that limit the time to access the data on an HDD (Hard Disk Drive) are mostly related to the mechanical nature of the rotating disks and moving heads. Seek time is a measure of how long it takes the head assembly to travel to the track of the disk that contains data. For random read or write requests, we might need to frequently change the head assembly to different disk tracks which results in much worse performance than sequential accesses.



(a)Hard Disk Drive of Laptop (b)Solid State Drive

Figure 1.3: Hard disk drive VS. solid state drive

New disk devices based on NAND-flash memory are becoming available with different performance characteristics from traditional magnetic hard disks. Figure

1.3b shows a NAND-flash based Solid State Drive (SSD). SSDs have no moving mechanical components, therefore there is no seek latency. They have good random performance, especially for read operations. However, NAND-flash cannot support in-place updates and has limited write endurance. When we update existing data, SSDs will write them to new places and mark the original data as invalid. To recycle the invalid pages, we need to erase a whole SSD block which normally contains 64-128 pages. The erase operation is very expensive which limits the write performance of SSDs. There are different types of SSDs, namely SLC, MLC and TLC according to the number of bits can be programmed per single cell. For SLC SSD, each cell can only represent one bit '0' or '1', while the cell of MLC and TLC SSD can represent more than one bit. Therefore, the capacity of SLC SSD is smaller than the corresponding MLC and TLC devices, however, the performance of SLC SSD is better than the MLC and TLC devices. The SSDs reuse the standard hard disk interfaces. To achieve out-of-place updates, there is a layer called Flash Translation Layer (FTL) inside SSD that manages the address mapping. It maps a logical block address (LBA) seen by the operating system to the actual physical block address (PBA) of a flash page. Moreover, to facilitate the space allocation and balance the flash cell's wear-out, FTL also controls the wear leveling and garbage collection. When the available empty blocks of the SSD are not sufficient, the garbage collection process will recycle those used blocks. To recycle a used block, the garbage collection process has to first migrate all the valid pages to a new block, then erase the old block.

Moreover, emerging nonvolatile memory technologies (sometimes referred as Storage Class Memory (SCM)), are poised to close the enormous performance gap between persistent storage and main memory. They can provide better performance than flash-based SSDs. The SCM devices can be attached directly to memory bus

and accessed like normal DRAM. It becomes then possible to exploit memory management hardware resources to improve file system performance. However, in this case, SCM may share critical system resources such as the TLB, page table with DRAM which can potentially impact SCM's performance.

Our research work in this part focuses on building efficient and high performance data storage utilizing flash-based SSDs and the emerging nonvolatile memories.

In chapter 3, we integrate nonvolatile DIMMs and flash SSD as a hybrid storage, instead of building disk storage on SSD directly. The nonvolatile DIMMs combine traditional DRAM, Flash, an intelligent system controller, and an ultracapacitor power source to provide a highly reliable memory subsystem that runs with the latency and endurance of the fastest DRAM, while also having the persistence of Flash (data on DRAM will be automatically backed up to flash memory on power failure). We utilize nonvolatile DIMMs to further improve SSD's performance. We know that SSD has good random read performance, however, small random writes bring down its performance and lifetime. We design a hybrid storage system managed by our proposed file system, NVMFS, which utilizes nonvolatile DIMMs to absorb the small random write requests. When the space of nonvolatile DIMMs is not sufficient, we begin to flush data to SSD in large and optimized write units. As a result, our design effectively improves the performance of SSD while improving the garbage collection overhead.

In chapter 4, we design a secondary disk cache utilizing multiple SSDs, which can be shared by a number of hard disk drives. Instead of managing the SSD devices as a traditional RAID volume, we manage them directly at the cache layer. To improve the cache space utilization, we only duplicate dirty data that are cached in SSDs. This ensures that we won't lose any data for single SSD failure at the cache layer. As a result, our design significantly improved the hit ratio of the SSD caches and the

throughput of the storage system.

In chapter 5, we analyze the problem of increased TLB misses while employing the emerging nonvolatile memories as data storage. We propose to solve this problem by employing superpages to reduce the pressure on memory management resources such as the TLB. As a result, the file system performance is further improved. We also analyze the space utilization efficiency of superpages. We improve space efficiency of the file system by allocating normal pages (4KB) for small files while allocating super pages (2MB on x86) for large files. We show that it is possible to achieve better performance without loss of space utilization efficiency of nonvolatile memory.

2. PROCESSOR CACHE DESIGN TO IMPROVE LIFETIME AND ENERGY IN PCM-BASED MAIN MEMORY

While process technology scaling continues providing ever greater numbers of transistors, current and future process technologies constrain the transistor performance and power gains that traditionally accompanied process scaling [30]. Recently this trend led to the emergence of chip-multiprocessor (CMP) designs as a means to leverage increasing transistor counts to achieve greater application performance more with more power efficiency than traditional monolithic processors. Similar technology scaling constraints for DRAM memory recently led to the emergence of Phase Change Memory (PCM) as an alternative form of main memory in future processor designs [67]. While PCM memory provides better power and density scaling in future process technologies, it introduces new design challenges with respect to lifetime durability and wear-out. In this chapter, we propose low-complexity techniques, utilizing existing cache memory systems, to substantially improve the lifetime durability and energy consumption of PCM main memory.

2.1 Background

2.1.1 Processor cache hierarchies

Current applications are placing greater pressure on their memory systems to maintain data and instruction stream needs. To this end, current chips employ memory system hierarchies with three levels of cache prior to main memory [51]. Multi-level cache hierarchies provide an approximation of a unified, fast, large memory space, through the low latency access times of small, private, first-level caches and the large capacities of shared, last-level caches. As the design of the last-level cache is optimized towards capacity rather than speed, these caches are often highly

associative.

In all associative caches, when a cache miss occurs a decision must be made regarding which block will be evicted and replaced. Current caches typically employ Least Recently Used (LRU) or approximations of LRU policies in deciding on which victim block to evict from cache in order to make room for a missing block. If the victim block is clean, it is simply discarded, however, if the current victim block is dirty, it must be written to memory before the new block may be written to cache. While many variations of LRU and other policies such as Least Frequently Used (LFU) [47] have been studied, much of the focus of this research has been on minimizing overall miss ratio at the caches.

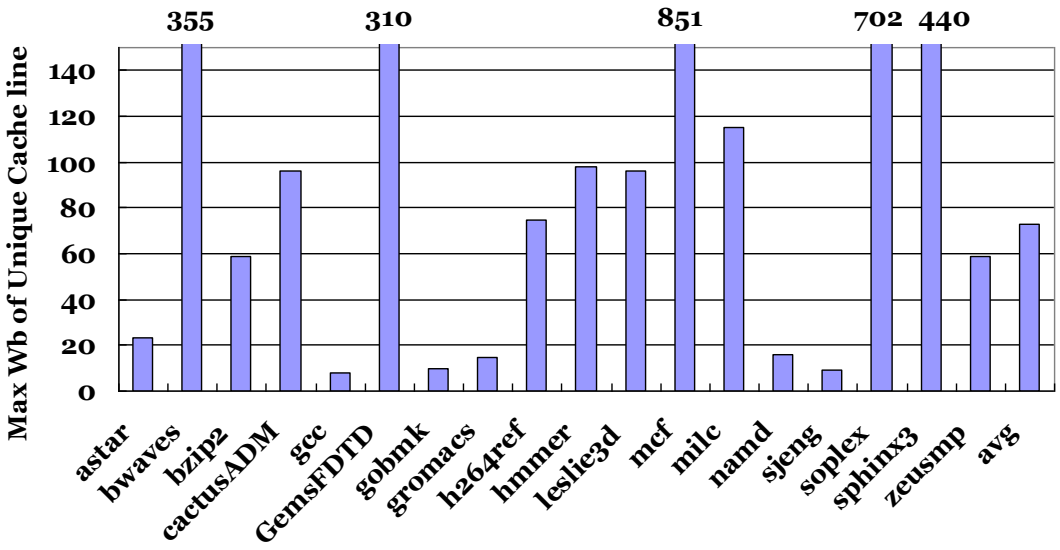


Figure 2.1: Number of total evictions per unique dirty line from a typical, 16-way, 2MB, last-level cache (LLC) for SPEC CPU2006 benchmarks.

As a result of this miss ratio focus, LRU and it's variants often lead to frequent re-writebacks of dirty cache lines to main memory, as illustrated in Figure 2.1. The

figure shows the total number of dirty evictions from a processor's LLC divided by the total number unique dirty lines, giving the average number of times each dirty line is re-written to the main memory for applications in the SPEC CPU2006 benchmark suite. As the figure shows, more than half the benchmarks re-evict the same dirty lines more than 10 times *on average*, two benchmarks re-evict cache lines hundreds of times on average. Given the PCM cell's high write cost in terms of energy, latency and wearout, re-writing the same cache lines repeatedly is highly undesirable. We propose to shift the focus of LLC design to account for the costs of writes to PCM based main memory, while maintaining low miss rates.

2.1.2 Last-level cache based writeback reduction

In this Chapter, we introduce new cache replacement policies and cache organizations that reduce the writeback data volume while minimizing the impact of cache miss rates. Figure 2.1 shows that many dirty cache lines written back to main memory will later be modified again, effectively ping-ponging from memory to cache and back many times during the course of the application. This Chapter examines techniques to favor retention of frequently re-written dirty lines, over lines which are clean. We observe that modern LLCs are both large relative to application footprints and highly associative, yielding an opportunity to reduce writebacks while minimally increasing misses to the LLC. Furthermore, as the techniques developed only imply changes to the LLC, the effect on the total memory system access latency is very small. The primary contributions of our design are as follows:

- We proposes a set of new cache replacement policies which favor eviction of clean data over dirty data. These policies are simple to implement and imply very low hardware overhead.

- We proposes a new cache organization which partitions sets to further favor

the retention of frequently re-written dirty lines. This cache organization is a natural fit for the banking typically found in LLCs, implying very little extra hardware overhead.

- We show these replacement policies and the new cache organization improve PCM lifetime by 49%-66% over a memory system employing randomized wear-leveling techniques [63].
- We also show our design reduce PCM energy consumption by 21%-31% on average over traditional LLC cache design.

2.2 Design and implementation

Based upon the observation that dirty blocks are often re-written to main-memory many times during a program's execution, we propose low-complexity, low-overhead techniques to modify the LLC with the goal of retaining dirty blocks which will be re-written frequently while maintaining low miss-rates. Our techniques address two aspects of cache design: cache replacement policies and cache organization. The proposed cache replacement policies favor replacement of clean blocks, keeping dirty blocks in the LLC longer, effectively reducing redundant writebacks. Moreover, we propose a new cache organization—the *partitioned-cache*, which aims to further bias replacement to favor those particular dirty blocks in a set which are re-written most frequently. The partitioned-cache accomplishes this by sub-dividing cache sets into partitions and adaptively determining which partition to select blocks to evict from depending on the relative writeback performance of those partitions. These techniques not only prolong the lifetime of PCM-based main memory, but also save the power and energy since PCM device has much higher power consumption and latency for write relative to read. This section discusses cache replacement policy based writeback reduction as well as cache organization based writeback reduction.

2.2.1 Cache replacement policy

PCM-based main memory has limited write endurance and different read and write cost in terms of time and energy. As a result, the cache replacement algorithm of the LLC used with PCM-based main memory should consider not only the miss-rate but also the cost, in terms of energy and lifetime, for replacing dirty blocks. Therefore, we propose two cache replacement policies for the last-level of cache which extend conventional LRU by integrating write costs to PCM main memory with cache locality. Our policies reduce the write traffic of last-level cache to PCM main memory, while maintaining a low miss rate. The first policy, *Protect-0* maximally reduces the writeback of last-level cache. The other policy, *Protect-N*, seeks to balance write traffic reduction and the last-level cache's misses.

2.2.1.1 Protect-0

There are two costs for cache replacement on the last-level cache. One is the read cost of fetching a requested cache block from main memory. The read cost may be minimized through cache locality, and is the focus of the LRU replacement algorithm. PCM-based main memory introduces a second, write cost, when evicting a dirty cache block from the last-level cache. PCM-based main memory has higher write access latency relative to read, and limited write endurance; we therefore propose the *Protect-0* cache replacement algorithm to maximally reduce the writeback cost.

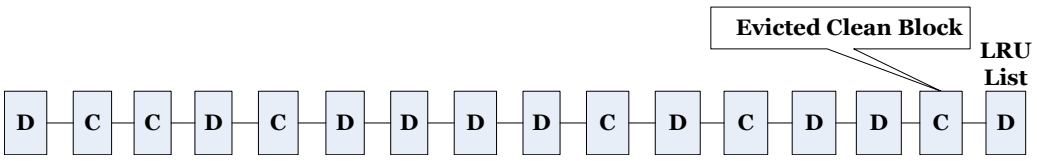


Figure 2.2: MRU-LRU list for one set in a cache using the Protect-0 replacement algorithm.

Protect-0, shown in Figure 2.2, is a modified form of the LRU algorithm. When a replacement is required in the last-level cache, Protect-0 searches the LRU list to find an LRU clean block to replace. In the figure, the LRU clean block is the block next to the LRU block, so it is chosen for the next eviction by Protect-0. If there's no clean block in the LRU chain, Protect-0 replaces the dirty LRU block.

Protect-0 attempts to maximally reduce write traffic to PCM main memory. As long as there are clean blocks in LRU chain, Protect-0 will always select a clean block for replacement and no additional writes to PCM main memory are generated. Protect-0 fosters the retention of dirty cache blocks in the cache longer than clean blocks. The longer dirty blocks remain in the cache, the more likely they are to receive subsequent writes, reducing the total number of write-backs over the course of the program by reducing the frequency with which dirty blocks ping-pong between the main memory and the cache. The disadvantage of Protect-0 is it violates the principle of temporal locality for clean blocks and therefore, may increase misses in the LLC in the event that the replaced clean block is referenced again in the near future and the preserved dirty block is not.

2.2.1.2 Protect-N

To minimize the increased misses caused by Protect-0, we propose another policy called *Protect-N*. Protect-N balances dirty replacements against increased misses. Previous studies have shown, in highly associative caches when sets are sorted in most recently used (MRU) to LRU order, the vast majority of hits typically occur in the first few MRU ways of the cache and the remaining ways provide only marginal decreases in miss rate. Protect-N leverages this observation by dividing the LRU list into two parts: the protected region and the non-protected region. The protected region contains the N MRU blocks for which temporal locality is preserved regardless

of clean/dirty status. The non-protected region includes the rest of the cache blocks (W-N for W-way cache) in the LRU list. Protect-N only applies clean-first policy to the non-protected cache blocks of the LRU chain. By protecting the N MRU blocks, it is expected that locality will be preserved while reducing the writebacks by evicting clean blocks from the remaining (LRU) blocks.

Determining the window size of N is very important to minimize misses while preserving dirty data. A large protected window size will reduce the possibility of finding a clean block for replacement, leading to more dirty replacements in PCM-based main memory. A small protected window size increases the chances of replacing a clean block that is likely to be referenced soon, leading to an increased miss rate of last-level cache. Considering both two kinds of replacement cost of last-level cache, we experimentally determined the best value of N to be 4 for 16-way, 2 for 8-way and 1 for 4-way caches.

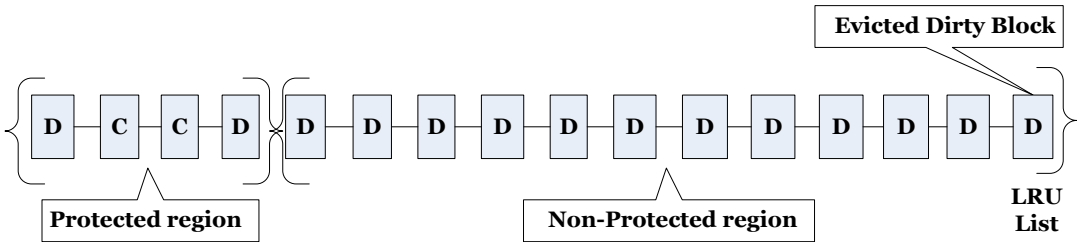


Figure 2.3: MRU-LRU list for one set in a cache using the Protect-N replacement algorithm (N=4).

The Protect-N algorithm always protects the first N MRU blocks regardless of their clean/dirty status and only applies the clean-first policy to the non-protected blocks. As shown in Figure 2.3, if Protect-N fails to find a clean block from the non-protected region of LRU list, it will simply replace the LRU block. In the figure,

there are no clean blocks in the non-protected region, therefore Protect-N (N=4 in this case) will choose the LRU block for replacement. Since the LRU block is dirty, a writeback request must be generated to the lower-level PCM-based main memory. Protect-N may not perform as well as Protect-0 at reducing write traffic, however attempts to balance the miss rate of last-level cache against writebacks to the main memory.

2.2.2 Cache organization

We observe that dirty cache blocks which are evicted and re-fetched repeatedly over the course of a program's execution often will have significant intervening references between consecutive writes. This is often the case in applications which stream clean data through the cache. It also can occur in applications which frequently allocate and free temporary data space on the heap and stack. In these instances Protect-N is unable to retain critical dirty blocks in the cache until their subsequent reuse.

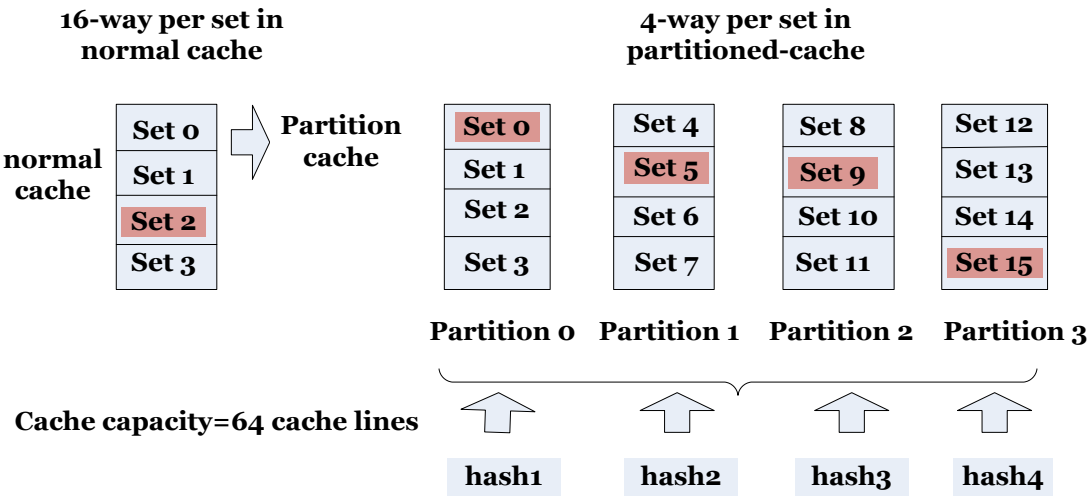


Figure 2.4: The organization of partitioned-cache.

To address the applications where Protect-N is insufficient to reduce re-writes of dirty data, we propose a new adaptive form of cache organization—*partitioned-cache*, which aims to preserve dirty cache blocks longer than Protect-N while introducing fewer misses than Protect-0. Figure 2.4 shows the difference between a normal cache and the partitioned-cache. As the figure shows, the partitioned-cache divides a Protect-N W-way cache into M partitions, each maintaining a smaller (N/M) protected region. In the figure, the partitioned-cache separates the original 16-way set into four 4-way sets, and applies Protect-1 within each of them.

As in a typical cache, the referenced block address in the partitioned-cache is divided into three components: the tag, the set index, and the offset, however in the partitioned-cache the least significant bits of the tag are used as a partition index. Upon receiving a reference, the set index determines which set and the tag is used to match the block within the set. All partitions of the set are searched in parallel. If the reference is a hit, the LRU list of the corresponding set of partition is updated. In the event of a miss, the partition index indicates a preferred partition within the set, to search for a clean block for eviction. In operation this is similar to Protect-N operating only within the preferred partition within the set. In the event that no unprotected clean block is found in the preferred partition of the set, the partition within the cache which has the most clean blocks is adaptively chosen to receive the miss. If no partitions have unprotected clean blocks then the preferred partition's LRU dirty block is evicted.

For the address shown in Figure 2.4, set 2 of Partition 1 will be considered first. If Protect-1 fails to find a proper clean block in set 2 of Partition 1, it will further search other partitions. Finally, if we cannot replace a proper clean block among all the partitions, the LRU block of mapped set within the preferred partition, set 2 of Partition 1, is evicted.

The benefit of the partitioned-cache lies in its longer term preservation of dirty blocks in the set partitions which contain the most dirty blocks. The partition cache steers replacement traffic away from those partitions which receive the most dirty blocks, allowing them to stay in the cache longer than the partitions which contain some clean blocks. As a result, the cache access traffic is shifted to the partition that has a lower writeback rate (writebacks per cache access), further reducing the writeback traffic to PCM-based main memory.

We note that, in comparison with the Protect-N cache, the partitioned-cache has little additional design complexity and latency overhead. Initial tag matching is done in parallel among a set's partitions as in a traditional cache of the same associativity, and hence incurs no extra latency on hits. The logic overhead of adaptive partition selection upon a miss may be placed in the cache's pipeline after the missing block's fetch from main memory has been initiated and prior to it's return, hence off the critical path. We also note that our policies are only imposed on the LLC and not the rest of the cache hierarchy and the resultant LLC does not retain the inclusion property.

2.3 Evaluation

The primary goal of this work is to improve PCM main memory lifetime, and energy through a reduction in the number of writebacks to main memory from the LLC while maintaining a low impact on system performance. To this end, in this section we evaluate our modified cache replacement algorithm and cache organization in their direct impact on PCM lifetime and energy consumption. We then examine the cache performance in terms of writebacks and misses, which cause the corresponding changes on PCM memory lifetime and energy. We note that the techniques which require substantial changes to the main memory system and do not focus on the

Table 2.1: Baseline cache configurations

System	One core	Eight cores
L1 cache (Private)	64KB, 2-way, LRU, 64Bytes block	64KB L1, 2-way, LRU, 64Bytes block
L2 cache (Shared)	2MB, 16-way, LRU, 64Bytes block	8MB, 16-way, LRU, 64Bytes block

LLC [61, 65, 63], are orthogonal and complimentary to our techniques. As such they are not evaluated in this work for the sake of brevity.

2.3.1 Methodology

The proposed cache replacement policy and cache organization were evaluated with both single- and multi-core configurations. Our baseline processor configurations for an 8-core CMP and a single core system are shown in Table 2.1. In both models, each core has its own private L1 caches, each 64KB. In the CMP model, a shared 8MB, 16-way L2 cache forms the last-level cache (LLC), upon which our modified cache replacement policies and cache organization are used. The L1 and L2 caches in both models have the same block size of 64Bytes.

For single core simulation, we use 18 applications from SPEC CPU2006 benchmark suite [74] and collected the memory system reference traces as the input of our simulator. The memory system traces were run through an in-house cache simulator to simplify and speed LLC cache organization development and evaluation.

For multi-core simulation, we use the M5 architecture simulator [5] to generate the simulation results and evaluate our modified cache replacement policies and cache organization. We chose PARSEC 2.1 benchmark suite [21, 4] as workloads which contains a suite of multi-threaded, CMP oriented applications, thus is suitable for

the evaluation of a CMP machine.

For the partitioned-cache, we always ensure the way-complexity is equal with normal cache organization (i.e. partitioned-cache with 4-way, 4-part is compared with 16-way normal cache). Except where otherwise noted, in the following experiments we vary both the L2 cache's replacement policy (Protect-0, Protect-N) and cache organization (traditional cache, partitioned-cache). For simplicity, in all figures, P-0, P-4, P-1-4part represent our designs of Protect-0, Protect-4, Protect-1-4partition respectively. Our design only impacts the L2 cache, the LLC of the system, the L1 still uses the traditional LRU replacement algorithm and normal cache organization. Dirty cache blocks at the end of simulation are included in all the writeback counts.

2.3.2 Single-threaded simulation

In this section we present the results from a single-core, single-threaded evaluation of our proposed cache replacement policies and cache organization.

2.3.2.1 Lifetime

The effective lifetime of PCM is ultimately limited by the maximum number of writes to a given cell. In this section we evaluate the improvement in lifetime by examining the relative reduction in re-writes to the cell which has the maximum number of writes for each of our techniques versus LRU. We show the impact of our design on a memory system that employs randomized wear-leveling similar to that proposed in [71, 63, 52].

As we see in figure 2.5, our design provides a 49% gain (on average) over LRU when both applying on a randomized wear-leveling PCM system. This is because wear-leveling techniques can only distribute the writes to PCM more evenly, while the number of total writebacks can't be reduced at all. As our design effectively reduces writesbacks from LLC to PCM main memory, it's natural to achieve better

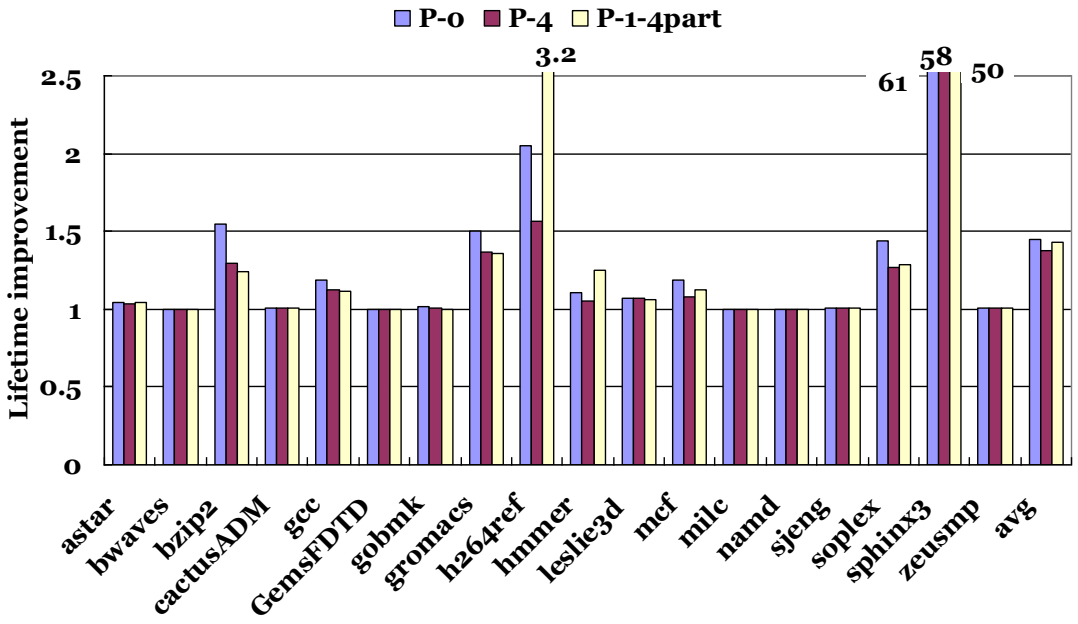


Figure 2.5: Lifetime improvement of SPEC2006 benchmarks on randomized wear-leveling PCM system

performance when combined with other wear-leveling techniques. Importantly, our design accomplishes the writeback reduction without the additional complexity of a DRAM cache in front of PCM.

2.3.2.2 Energy consumption

Unlike previously proposed randomized wear-leveling techniques, our designs also reduce the energy consumption relative to conventional LRU cache. From Table 1.1, we see write-power is approximately 10x read-power and the write-latency is approximately 6x read-latency for PCM devices. We calculate the energy consumption using the formula: Energy = Misses*R_power*R_latency + Writebacks*W_power*W_latency. As our design can effectively reduce write traffic to PCM, we expect them to reduce energy consumption as well. Figure 2.6 shows the normalized energy consumption of our designs against LRU baseline cache. On average, our Protect-0, Protect-4

and Protect-1-4partition reduce the energy consumption by 19.2%, 16.3% and 21% respectively.

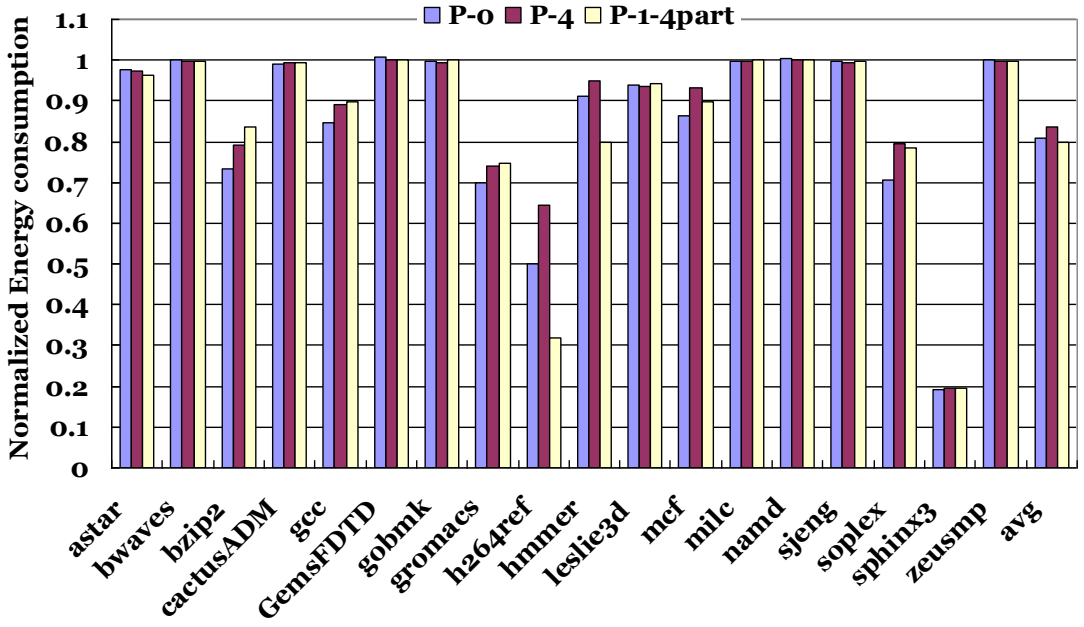


Figure 2.6: Normalized energy consumption for SPEC2006 benchmarks.

2.3.2.3 Write traffic

In this section, we explore how our design achieves improvement on lifetime and energy of PCM memory system. Figure 2.7 shows the impact on writebacks of the LLC cache when applying Protect-0, Protect-4 and Protect-1-4part versus traditional LRU cache for all the benchmarks. In Figure 2.7, we see all designs significantly reduce the write traffic to PCM-based main memory. Our design greatly reduces the total number of writes performed on the PCM device, so that its lifetime is improved effectively.

Generally, Protect-1-4partition performs better than Protect-4 and slightly worse

than Protect-0. It is, however, interesting to note that in several instances, for example h264ref, Protect-1-4partition actually outperforms Protect-0. In these cases Protect-1-4partition is adaptively retaining critical dirty blocks longer than even Protect-0 would allow.

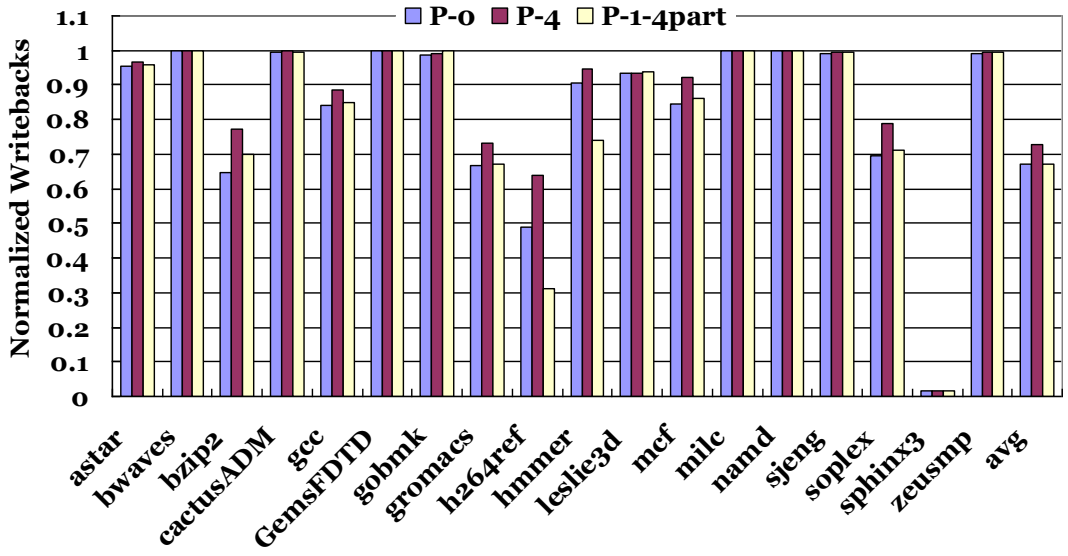


Figure 2.7: Normalized last-level cache writebacks for SPEC2006 benchmarks.

2.3.2.4 Misses

Figure 2.8 shows the corresponding miss impact of our design against traditional LRU baseline. As expected, Protect-0 has the worst effect on misses. Generally, Protect-4 and Protect-1-4partition have a lower impact on misses than Protect-0, and the overall increase is quite small compared with conventional LRU. Therefore, Protect-4 and Protect-1-4partition effectively reduce the additional cost of increasing LLC's misses and achieve a good trade-off between reducing write traffic and preserving locality. Moreover, Protect-1-4partition generates the fewest misses, approaching

LRU. Generally, the partitioned-cache scheme outperforms the corresponding normal cache organization in terms of both writebacks and misses, which demonstrates the benefit of partitioned-cache organization.

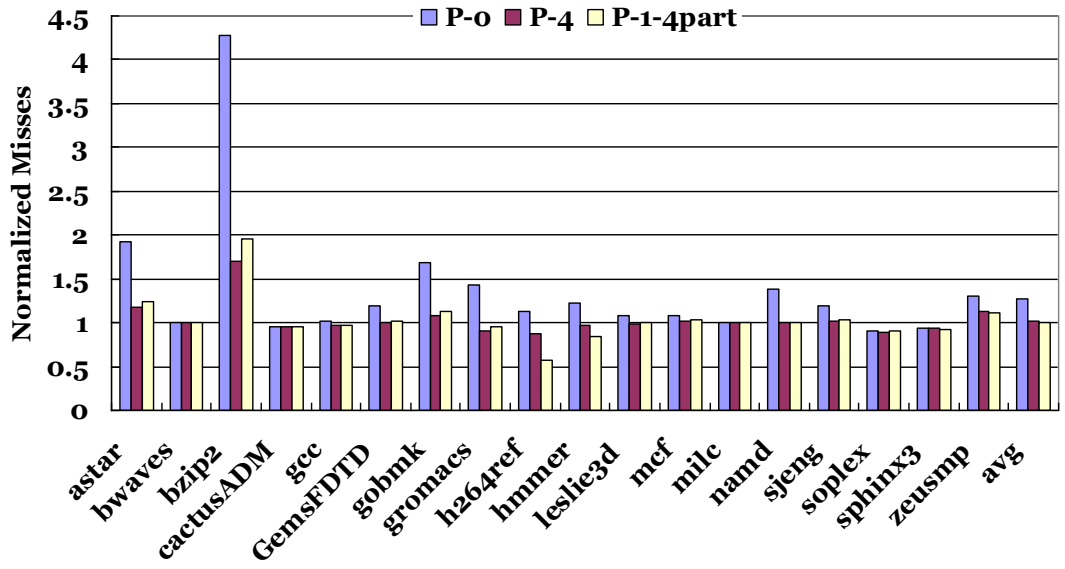


Figure 2.8: Normalized last-level cache misses for SPEC2006 benchmarks.

2.3.3 Multi-threaded simulation

In this section we examine the impact of our proposed cache replacement policies and cache organization on multi-threaded applications on multi-core processors.

Figure 2.9 and 2.10 show the lifetime and energy improvement for our proposed design relative to the baseline randomized wear-leveling system. We see our design also has impressive improvement on lifetime and energy for multi-threaded applications. These improvement come from efficient writeback reduction while at the same time keeping misses comparable to the tradition LRU. Figure 2.11 and 2.12

show the corresponding writeback and misses performance under our design relative to LRU for PARSEC benchmarks. We can see that our design reduces the write-back by 40% while only increases misses by less than 1%. Energy consumption is also reduced greatly because of much less expensive-write performed. Generally, our cache replacement policies and cache organization have similar performance on multi-threaded applications as that of single-threaded ones, both improve lifetime and energy of PCM effectively.

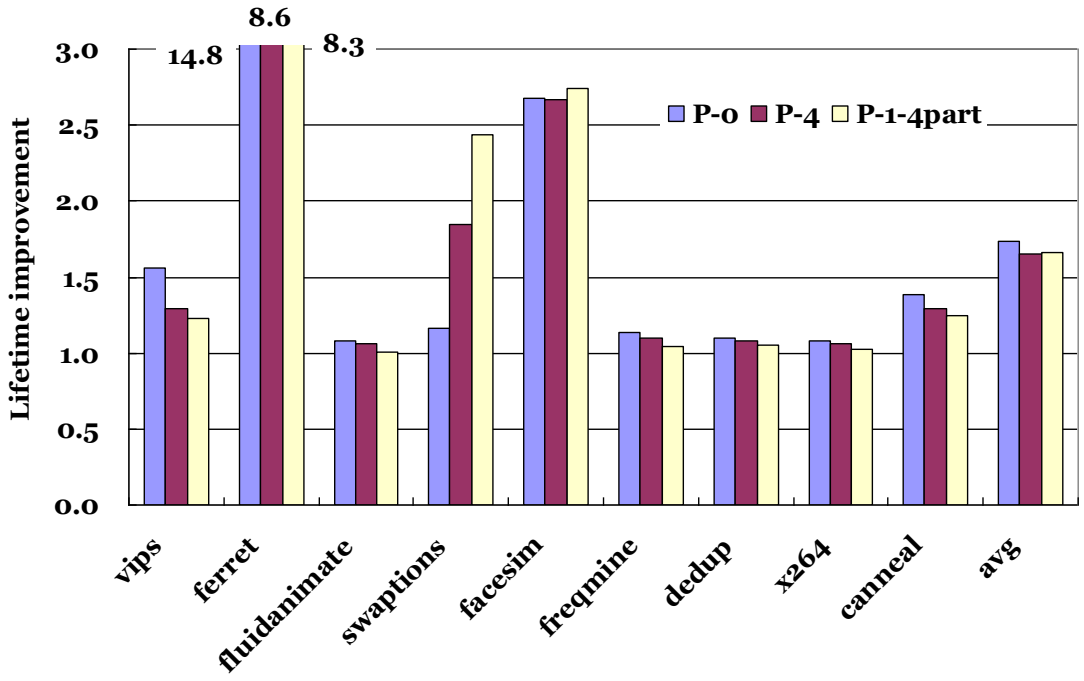


Figure 2.9: Lifetime improvement of PARSEC benchmarks on randomized wear-leveling PCM system

2.4 Analysis

In this section, we analyze the performance of the proposed cache design with respect to wear leveling. We also provide some intuition behind what makes the

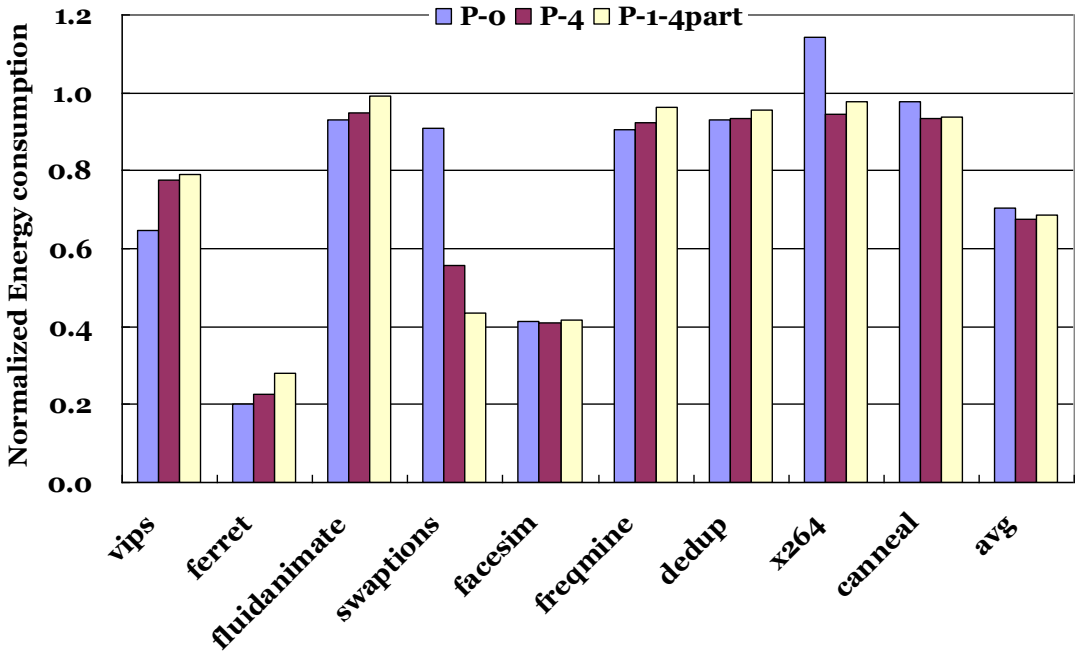


Figure 2.10: Normalized energy consumption for PARSEC benchmarks.

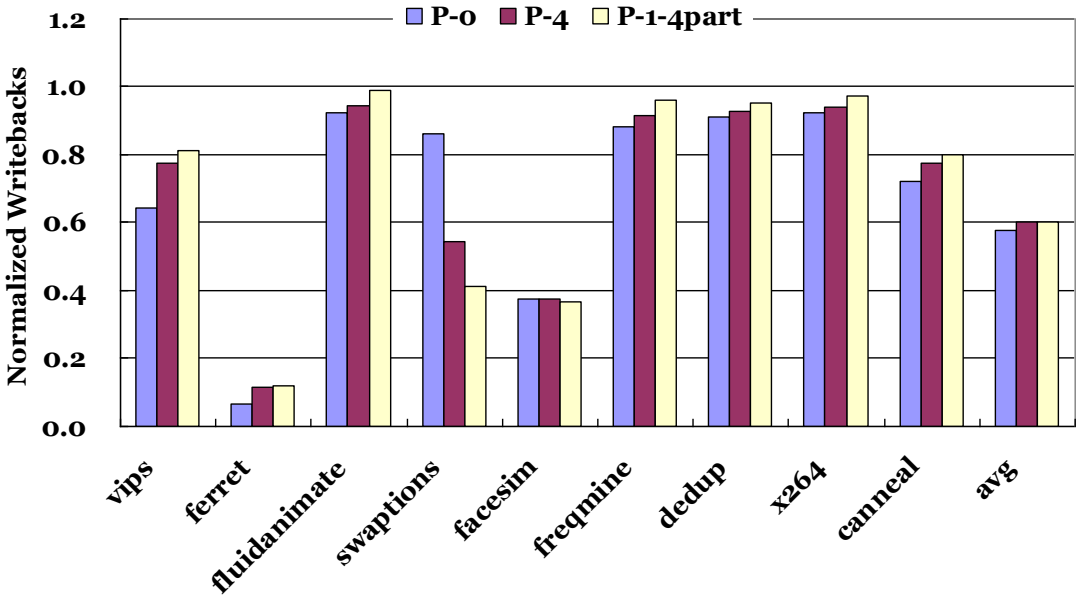


Figure 2.11: Normalized last-level cache writebacks for PARSEC benchmarks.

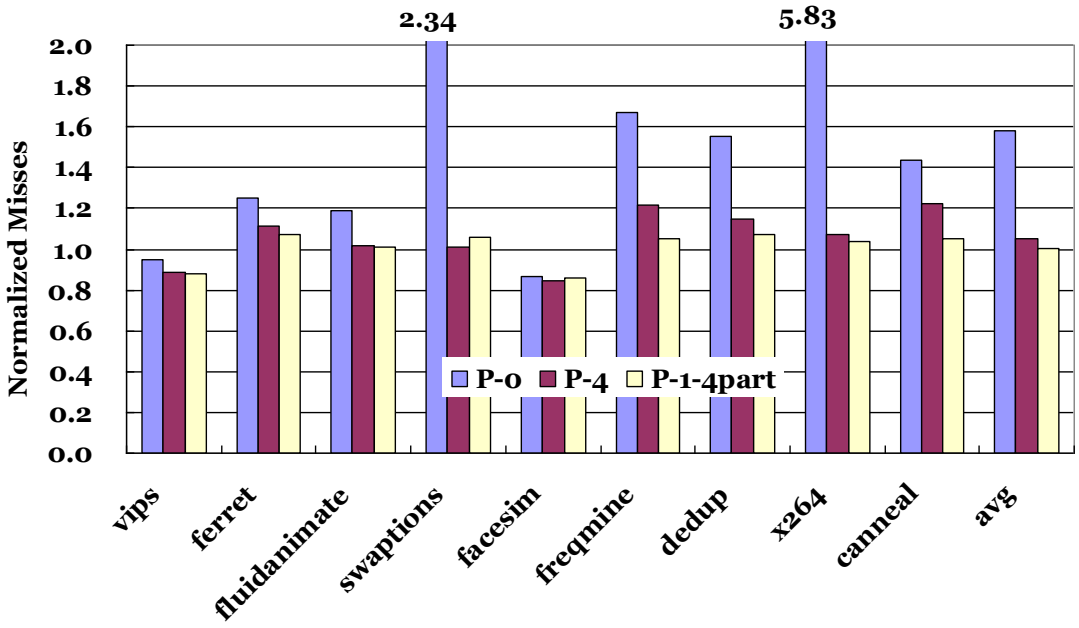


Figure 2.12: Normalized last-level cache misses for PARSEC benchmarks.

partitioned-cache work.

2.4.1 Wear leveling

Wear-leveling is very important for PCM-based main memory, if writes occur more frequently on certain PCM cells the write endurance of those cells is soon hit, potentially leading to the breakdown of the device as a whole. Therefore, a uniform write distribution can effectively prolong the overall life time of a PCM device. To further explore our design's effect on wear-leveling, we apply our design and LRU on a baseline PCM system (without wear-leveling). Figure 2.13 shows a sorted cumulative distribution graph of main memory write addresses versus write counts for systems in which the LLC cache uses the LRU baseline, Protect-4, Protect-1-4partition on the *soplex* benchmark (other benchmarks have similar pattern). The two linear traces representing an ideal uniform write distribution for lru baseline and

our technique respectively. In the figure the x-axis represents the written address tags, sorted according to the number of writes per tag, and the y-axis represents the number of total writes for each tag. As the figure shows, our designs produce a much lower and flatter distribution than the 16-way baseline LRU cache. Moreover, the uniform lines show that our design will perform much better than LRU baseline when combined with wear-leveling algorithms [71, 63, 52]. These results indicate not only do our designs produce fewer writes than baseline LRU cache on this workload, they also produce a much more uniform wear than LRU for its number of writes.

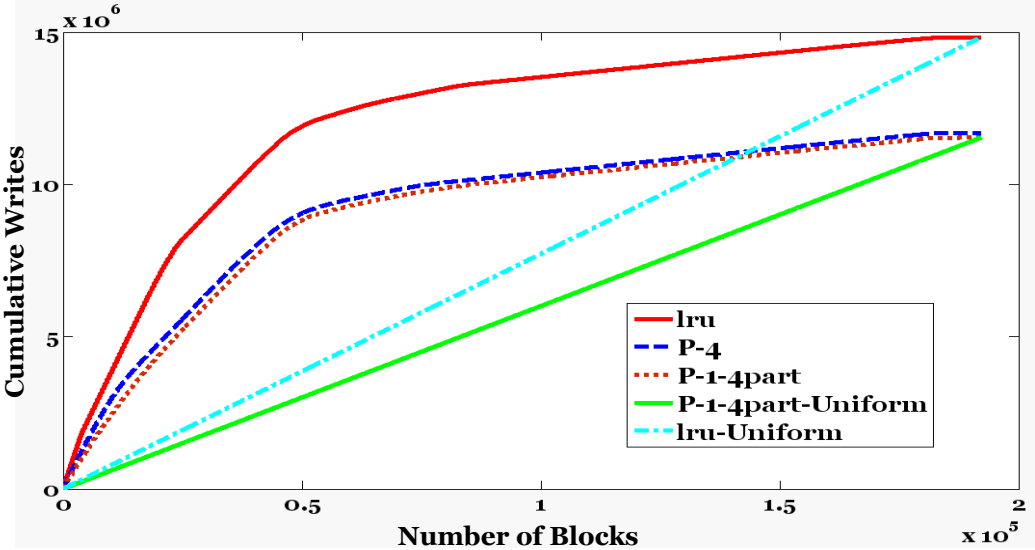


Figure 2.13: Write distribution of soplex

2.4.2 Behavior of the partitioned-cache

To further analyze the behavior of the partitioned-cache, we examine the relationship between writeback rate and access traffic. For the sake of comparison with the partitioned-cache organization, we apply Protect-0 and Protect-1 both on a tra-

ditional 4-way cache, which is manually divided into four partitions according to the most two significant bits of the set index. As shown in Figure 2.14, the accesses are normalized to traditional LRU baseline, and the wbrate is the writeback rate. We observe that Protect-0-4way and Protect-1-4way have comparable accesses among all partitions, while Protect-1-4partition preserves a much higher access rate to the partition with lower writeback rate. Generally, the partitioned-cache holds on to the lines that have higher writeback rate by reducing the number of times they are replaced.

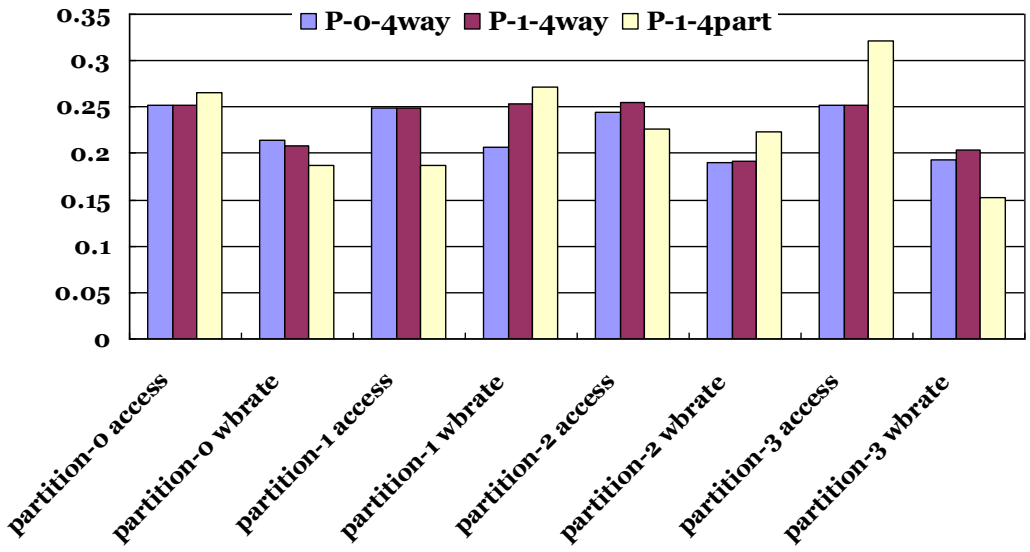


Figure 2.14: Accesses, writeback rate of each partition under hmmer application.

2.5 Related work

This section describes related work in PCM based memory systems, cache replacement policies and cache organization.

2.5.1 Phase change memory based memory systems

Recently a handful of works have explored the design space of alternative main memory technologies. Li et al. and Dhiman et al. have investigated hybrid DRAM and PCM memory architectures [86, 18], while Sun et al. explored hybrid PCM and Solid State Disk (SSD) storage architectures [77]. In both cases, the authors sought to use one technology to provide buffers for frequently written data, offsetting some of the penalties of the other technology. These works are largely orthogonal and possibly complementary to the work presented here, as we focus upon reducing writebacks from the lower levels in the memory system.

Qureshi et al. propose two techniques called write cancellation and write pausing for improving the read performance of PCM memories [61]. These techniques give preference to reads at PCM and help mitigate the long write times in order to improve read performance. Separately, Qureshi et al. propose sub-block cache writebacks to reduce the writeback volume to PCM [65]. This work has a similar motivation as our work here, but proposes a different solution. While their work incorporates sub-block level dirty bits and only writes dirty words to PCM instead of full cache blocks, our work modifies the cache replacement policy to reduce the number of writebacks.

Lee et al. propose three techniques to improve PCM lifetime [45, 46]. These include elimination of redundant bit writes, row shifting, and segment swapping, all of which are aimed at either reducing the number of writes or leveling wear across the arrays. These techniques should be complimentary with the work presented here, although the benefit of combining them will not be strictly additive as our technique also will level wear somewhat across the arrays.

2.5.2 Cache replacement policies

Cache replacement policies have been studied extensively since caches came into wide use in the early 1980's [28, 27, 35, 2, 47]. These works focused primarily upon achieving the fewest misses, as cache miss rate has the most direct effect on processor performance. These works largely disregard the effect of writebacks on the DRAM because, from the processor's point of view, writebacks occur in the background and do not directly affect miss latency. Furthermore, in DRAM technology, writes are not much more expensive than reads so there was less need to favor them.

In a seminal paper introducing the "Snoopy" cache coherence protocol, Goodman discussed the impact of writebacks on memory system bandwidth [23]. This work focused on shared memory induced writebacks and write-through caches. Mattson's stack algorithm is a useful tool to study associativity and replacement algorithms in caches [22]. Several recent works use Mattson's stack algorithm to improve cache utilization [9], sharing between processors [64, 37], or to improve DRAM utilization [75].

Clean First LRU (CFLRU) policy was proposed for page cache management in solid-state disks (SSDs) for similar reasons of reducing expensive writes to SSDs [58]. While the page replacement policies they propose have some similarities with our *Protect-N* policy, our focus here is on appropriate last-level, on-die cache, block replacement policies for PCM main memory. As such our proposed policies must be hardware implementable with low latency overheads and limited logical complexity.

2.5.3 Cache organization

Cache organization has also been extensively studied. Seznec proposed skew associative caches employing different hash functions enabling different sets to map to different parts (or partitions) of the cache [72, 6]. Powell et al. examined techniques

to adaptively change the associativity and parallel search requirements of highly associative caches to reduce power and energy in the cache [60]. The set balancing cache allowed the associativity of a set in the cache to double based on observed miss behavior of that set [68]. A miss saturation counter is used to guide the expansion of a set into another location in the cache and these locations are serially searched to find a data item. The V-way cache employed a larger number of tag entries (compared to data entries) to allow variable associativity per set [62]. These alternative cache organizations were motivated by an aim of maintaining uniform accesses across the different sets or ways of the cache to approximate higher associativity and fewer misses. We propose an organization of the cache that intentionally skews the traffic (makes it less uniformly spread across the sets) to reduce the writebacks from the last level cache.

3. NVMFS: A HYBRID FILE SYSTEM FOR IMPROVING RANDOM WRITE IN NAND-FLASH SSD*

In this chapter, we propose a hybrid storage system employing non-volatile DIMMs and solid state disks. The hybrid storage system is managed by our file system, NVMFS, which leverages both devices' advantages and compensates their disadvantages. Our design utilizes non-volatile DIMMs to absorb small random write requests and optimizes the write operations on flash SSD. We show that such a hybrid storage system can improve write throughput and garbage collection efficiency of SSD.

3.1 Background

For many years, the performance of persistent storage (such as hard disk drives) has remained far behind that of microprocessors. Although the disk density has improved from GBs to TBs, data access latency has increased by only 9X [17, 16]. Compared with HDDs, SSDs have several benefits. An SSD is a purely electronic device with no mechanical parts, and thus can provide lower access latencies, lower power consumption, lack of noise and shock resistance. However, SSDs also have two serious problems: limited lifetime and relatively poor random write performance. In SSDs, the smallest write unit is one page (such as 4KB) and can only be performed out-of-place, since data blocks have to be erased before new data can be written. Random writes can cause internal fragmentation of SSDs and thus lead to higher frequency of expensive erase operations [11, 7]. Besides performance degradation, the lifetime of SSDs can also be dramatically reduced by random writes.

Flash memory is now being used in other contexts, for example in designing

*Reprinted with permission from "NVMFS: A hybrid file system for improving random write in NAND-flash SSD" by Sheng Qiu and A.L.Narasimha Reddy, 2013. IEEE 29th Symposium on Mass Storage Systems and Technologies (MSST), Copyright 2013 by IEEE

Figure 3.1: Non-volatile DIMMs.

nonvolatile DIMMs [1, 79] as shown in Figure 3.1. These designs combine traditional DRAM, Flash, an intelligent system controller, and an ultracapacitor power source to provide a highly reliable memory subsystem that runs with the latency and endurance of the fastest DRAM, while also having the persistence of Flash (data on DRAM will be automatically backed up to flash memory on power failure). The availability of these nonvolatile DIMMs can simplify and enhance file system design, a topic we explore in this chapter.

In this chapter, we consider a storage system consisting of Nonvolatile DIMMs (as NVRAM) and SSDs. We expect a combination of NVRAM and SSD will provide the higher performance of NVRAM while providing the higher capacity of SSD in one system. We propose a file system NVMFS for such a system that employs both NVRAM and SSD. Our file system exploits the unique characteristics of these devices to simplify and speed up file system operations.

Traditionally, when devices of different performance are used together in a system, two techniques are employed for managing space across the devices. When caching is employed, the higher performance device improves performance transparently to the layers above, with data movement across the devices taken care of at lower layers. When migration is alternately employed, the space of both slower and faster devices becomes visible to the higher layers. Both have their advantages and disadvantages.

In our file system proposed here, we employ both caching and migration at the same time to improve file system operations. When data is migrated, the address of the data is typically updated to reflect the new location whereas in caching, the permanent location of the data remains the same, while the data resides in higher performance memory. For example, in current file systems, when data is brought into the page cache, the permanent location of the file data remains on the disk even though access and updates may be satisfied in the page cache. Data eventually has to be moved to its permanent location, in caching systems. In systems that employ migration, data location is typically updated as data moves from one location to another location to reflect its current location. When clean data needs to be moved to slower devices, data cannot be simply discarded as in caching systems (since data always resides in the slower devices in caching systems), but has to be copied to the slower devices and the metadata has to be updated to reflect the new location of the data. Otherwise, capacity of the devices together cannot be reported to the higher layers as the capacity of the system.

In our system, we employ both these techniques simultaneously, exploiting the nonvolatile nature of the NVRAM to effectively reduce many operations that would be otherwise necessary. We use the higher performance NVRAM as both a cache and permanent space for data. Hot data and metadata can permanently reside in the NVRAM while not-so-hot, but recently accessed data can be cached in the NVRAM at the same time. This flexibility allows us to eliminate many data operations that would be needed in systems that employ either technique alone.

When data is accessed from SSD, initially that data is cached in the NVRAM, and the file system retains pointers to both locations. If this data becomes a candidate for eviction from NVRAM and it hasn't been updated since it is brought into NVRAM, we discard the data from NVRAM and update the metadata to reflect the fact that

data resides now only on the SSD. If the data gets updated after it is brought into NVRAM, we update the metadata to reflect that the data location is the NVRAM and the data on the SSD is no longer valid. Since NVRAM is nonvolatile, we can retain the data in NVRAM much longer and get forced to flush or write this data back to SSD to protect against failures. This allows us to group the dirty data together and write the dirty data together to SSD at a convenient time. Second, this allows us to group data with similar hot-cold behavior into one block while moving it to SSD. We expect this will improve the garbage collection process at SSD in the longer term.

In order to allow this flexibility that we described above, where data can be cached or permanently stored on the NVRAM, we employ two potential addresses for a data block in our file system. The details of this will be described later in section 3.2.

The primary contributions of our design are as following:

- We proposes a new file system – NVMFS, which integrates Nonvolatile DIMMs (as NVRAM) and a commercial SSD as the storage infrastructure.

- NVMFS exploits the strengths of NVRAM and SSD to improve file system performance. In our design, we utilize SSD's larger capacity to hold the majority of file data while absorbing random writes on NVRAM. We explore different write policies on NVRAM and SSD: in-place updates on NVRAM and non-overwrite on SSD. As a result, random writes at file system level are transformed to sequential ones at device level when completed on SSD.

- NVMFS distributes metadata and relatively hot file data on NVRAM while storing other file data on SSD. Unlike normal caching or migration scheme, our design can permanently store hot data on NVRAM while also temporar-

ily caching the recently accessed data. To track the hotness of file data, we implement two LRU lists for dirty and clean file data respectively. Our file system will dynamically adapt the number of pages distributed between dirty and clean LRU lists. When the dirty file data are not hot enough we will collectively flush them (grouped into SSD blocks) to SSD and put them to the end of clean LRU list which may be quickly replaced whenever the space of NVRAM is not enough (we always replace LRU clean pages).

- We show that NVMFS improves IO throughput by an average of 98.9% when segment cleaning is not active, while improving IO throughput by an average of 19.6% when segment cleaning is activated, compared to several existing file systems.

- We also show that the erase operations and erase overhead at SSD are both effectively reduced.

3.2 Design and implementation

NVMFS improves SSD's random write performance by absorbing small random IOs on NVRAM and only performing large sequential writes on SSD. To reduce the overhead of SSD's erase operations, NVMFS groups data with similar update likelihood into the same SSD blocks. The benefits of our design resides on three aspects: (1)reduce write traffic to SSD; (2)transform random writes at file system level to sequential ones at SSD level; (3)group data with similar update likelihood into the same SSD blocks.

3.2.1 Hybrid storage architecture

In NVMFS, the memory system is composed of two parts, one is the traditional DRAM, the other is the Nonvolatile DIMMs. Figure 3.2 shows the hardware archi-

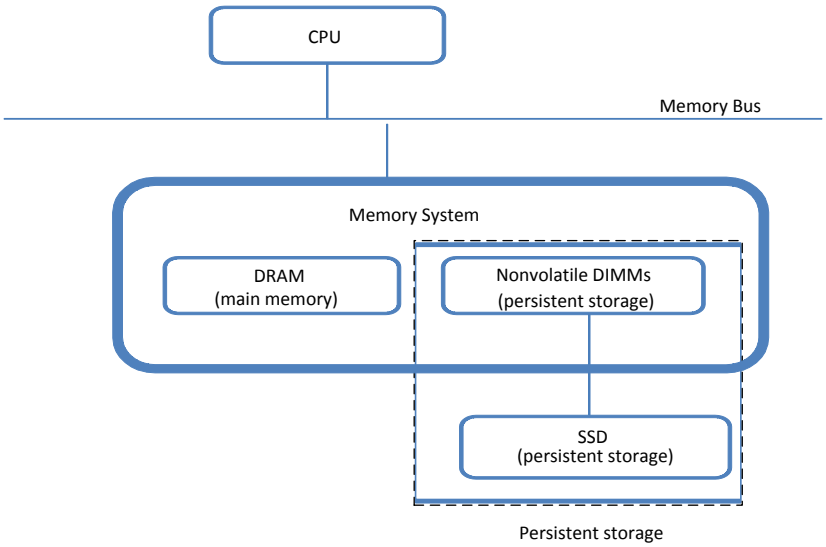


Figure 3.2: Hybrid storage architecture

tecture of our system. We utilize Nonvolatile DIMMs attached to the memory bus, and accessed through virtual addresses as NVRAM. The actual physical addresses to access NVRAM are available through the page mapping table, leveraging the operating system infrastructure. All the page mapping information of NVRAM will be stored on a fixed part of NVRAM. We will detail this later in section 3.2.2. If the requested file data is on NVRAM, we can directly access it through load/store instructions. While if the requested file data is on SSD, we need to first fetch it to NVRAM. It's noted that we bypass page cache in our file system, since CPU can directly access NVRAM which can provide the same performance as DRAM based page cache. To access the file data on SSD, we use logical block addresses (LBAs), which will be translated to the physical block addresses (PBAs) through FTL component of SSD. Therefore, NVMFS has two types of data addresses at file system level – virtual addresses for NVRAM and logical block addresses for SSD. In our design, we can store two valid versions for hot data on NVRAM and SSD respec-

tively. Whenever the data become dirty, we keep the recent data on NVRAM and invalidate the corresponding version on SSD. We will introduce how we manage the data addresses of our file system in section 3.2.2.

The benefit of building such a hybrid storage is that we can exploit each device's advantages while offsetting their disadvantages. Since SSD has poor random write performance and limited write cycles, we absorb random writes on NVRAM and only perform large sequential writes on SSD. For metadata and frequently accessed file data, we permanently store them on NVRAM, while distributing other relatively cold, clean data on SSD. We will detail how NVMFS distributes the file data between NVRAM and SSD in section 3.2.3.

3.2.2 File system layout

The space layout of NVMFS is shown in figure 3.3. The metadata and memory mapping table are stored on NVRAM. The metadata contains the information such as size of NVRAM and SSD, size of page mapping table, etc. The memory mapping table is used to build some in-memory data structures when mounting our file system and is maintained by memory management module during runtime. All the updates to the memory mapping table will be flushed immediately to NVRAM.

In the file system address space, the layout of NVMFS is very simple. The file system metadata which includes super block, inode table and block bitmap are stored on NVRAM while the file data are stored either on NVRAM or SSD based on their usage pattern. The block bitmap indicates whether the corresponding NVRAM or SSD block is free or used. In NVMFS, we always put hot file data on NVRAM and cold file data on SSD. We use LRU list to classify hot and cold data which will be discussed in section 3.2.3. In our current implementation, the total size of virtual memory space for NVRAM addresses is 2^{47} bytes (range: ffff000000000000

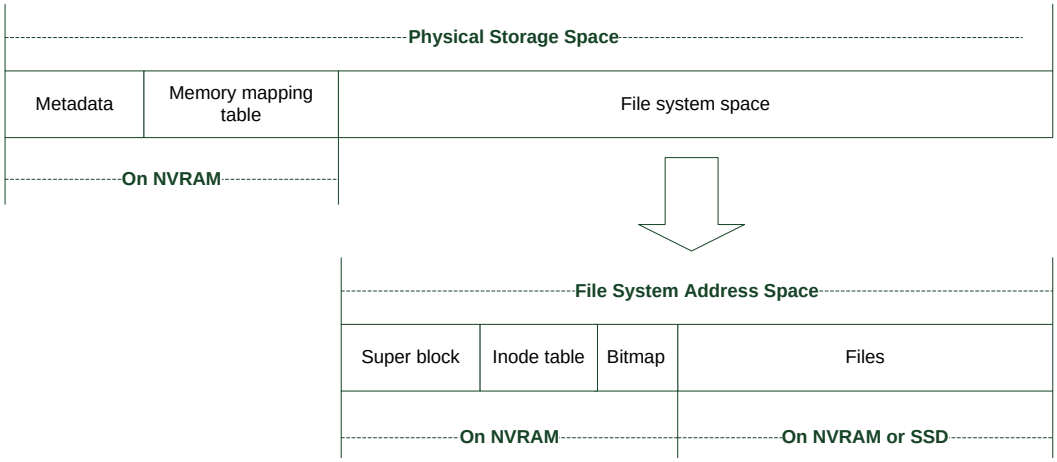


Figure 3.3: Storage space layout

- ffff7fffffffffff), which is unused in original Linux kernel. The space can be larger if we re-organize 64-bit virtual address space. We modified the Linux kernel to let the operating system be aware of the existence of two types of memory devices – DRAM and NVRAM, attached to the memory bus. We also added a set of functions for allocating/deallocating the memory space of NVRAM. This implementation is leveraged from previous work in [85].

In NVMFS, the directory files are stored as ordinary files, while their contents are lists of inode numbers. To address the inode table, we store the pointer to the starting address of the inode table in the super block. Within the inode table, we use a fixed size entry of 128 bytes for each inode, and it is simple to get a file's metadata through its inode number and the start address of the inode table. The inode will store several pieces of information including checksum, owner uid, group id, file mode, blocks count of NVRAM, blocks count of SSD, size of data in bytes, access time, block pointer array and so on. The block pointer array is similar as the direct/indirect block pointers used in EXT2. The difference is that we always allocate indirect blocks on NVRAM so that it is fast to index the requested file data even

when the file is large which requires retrieving indirect blocks. The block address is 64 bits and the NVRAM addresses are distinct from the SSD block addresses. To build our file system, we can use the command like "mount -t NVMFS -o init=4G /dev/sdb1 /mnt/NVMFS". In the example, we attached 4GB Nonvolatile DIMMs as the NVRAM, and inform NVMFS the path of the SSD device, finally mount it to the specified mount point.

3.2.3 Data distribution and write reorganization

The key design of NVMFS relies on two aspects: (a)how to distribute file system data between the two types of devices – NVRAM and SSD; (b)how to group and reorganize data before writing to SSD so that we can always perform large sequential writes on SSD.

File system metadata are small and will be updated frequently, thus it's natural to store them on NVRAM. To efficiently distribute file data, we track the hotness of both clean and dirty file data. We implemented two LRU (Least Recently Used) lists — dirty and clean LRU lists, which are stored as metadata on NVRAM. Considering the expensive write operations of SSD, we prefer to store more dirty data on NVRAM, expecting them to absorb more update/write operations. Whenever the space of NVRAM is not sufficient, we replace file data from clean LRU list. However, we also do not want to hurt the locality of clean data. We balance this by dynamically adjusting the length of dirty and clean LRU lists. For example, if the hit ratio of clean data is 2X more than that of dirty data, we increase the NVRAM pages that are allocated for clean data. In other words, we increase the length of clean LRU list. To achieve this, we maintain two performance counters which keep track of the hits on clean and dirty data (on NVRAM) respectively. We periodically measure and reset the counters. The total number of pages within clean and dirty LRU lists

is fixed, equalling to the number of NVRAM pages.

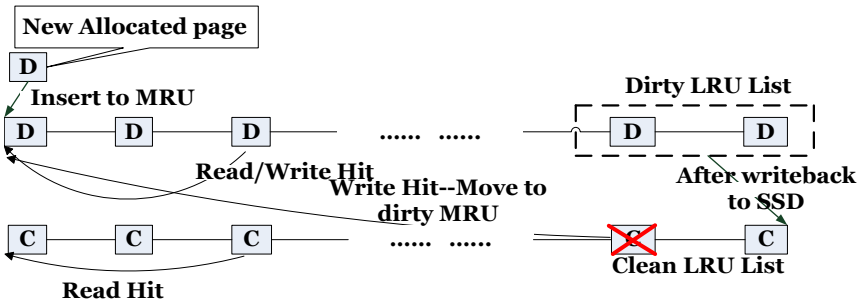


Figure 3.4: Dirty and clean LRU lists

Figure 3.4 shows the clean and dirty LRU lists as well as the related operations. When writing new file data, we allocate space on NVRAM and mark them as dirty, then insert to the MRU (Most Recently Used) position of dirty LRU list. Read/write operations on dirty data will update their position to MRU within dirty LRU list. For clean data, read operations update their position to MRU of clean LRU list, while write operations are a little different since the related data become dirty afterward. As shown in figure 3.4, writes on clean data will migrate the corresponding NVRAM pages from clean LRU list to the MRU of dirty LRU list.

Unlike existing page cache structure which flushes dirty data to the backed secondary storage (such as SSDs) within short period, our file system can store dirty data permanently on NVRAM. NVMFS always keeps the pointer to the most recent data version. We can choose when and which data to flush to SSD dynamically according to the workloads. We begin to flush dirty data to SSD whenever the NVRAM pages within dirty LRU list reaches a high bound (i.e. 80% of dirty LRU list is full). Then we pick dirty data from the end of dirty LRU list and group them into SSD blocks. We allocate the corresponding space on SSD with sequential LBAs

and write these data sequentially to SSD. This process continues until the NVRAM pages within dirty LRU list reaches a low bound (i.e. 50% of dirty LRU list is full). The flushing job is executed by a background kernel thread.

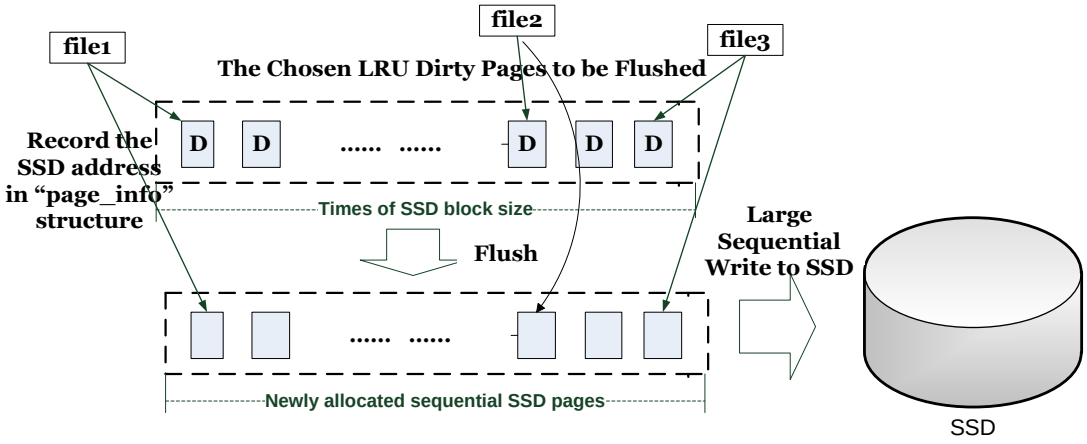


Figure 3.5: Migrate dirty NVRAM pages to SSD

Figure 3.5 shows how our file system migrates dirty data from NVRAM to SSD. The dirty NVRAM pages will become clean after migrating to SSD and will be inserted to the LRU position of clean LRU list. As a result, we have two valid clean data versions on NVRAM and SSD respectively. We can facilitate the subsequent read/write requests since we still have valid data versions on NVRAM. Moreover, we can easily replace those data on NVRAM by only reflecting their positions on SSD. In our file system, the file inode always points to the appropriate data version. For example, if file data have two valid versions on NVRAM and SSD respectively, the inode will point to the data on NVRAM. We have another data structure called "page_info" which records the position of another valid data version on SSD. It is noted that we won't lose file system consistency even if we lose this "page_info"

structure, since file inodes consistently keep the locations of appropriate valid data version. We will discuss file system consistency in section 3.2.5

3.2.4 Non-overwrite on solid state drive

We employ different write policies on NVRAM and SSD. We do in-place update on NVRAM and non-overwrite on SSD, which exploits the devices' characteristics. The space of SSD is managed as extents of 512KB, which is also the minimum flushing unit for migrating data from NVRAM to SSD. Each extent on SSD contains 128 normal 4KB blocks, which is also the block size of our file system. When dirty data are flushed to SSD, we organize them into large blocks (i.e. 512KB) and allocate corresponding number of extents on SSD. As a result, random writes of small IO requests are transformed to large write requests (i.e. 512KB).

To facilitate allocation of extents on SSD, we need to periodically clean up internal fragmentation within the SSD. For example, when clean data are fetched from SSD to NVRAM because of write accesses, the corresponding data on SSD will become invalid, since the updated data now reside on NVRAM. The fetched data can be smaller than one SSD extent or across several SSD extents, as a result, there will be invalid portions within SSD extents. During recycling, we can integrate several partial valid SSD extents into one valid SSD extent and free up the remaining space. This ensures that we can always have free extents available on SSD for allocation, which is similar to the segment cleaning process of log-structured file systems. It's noted that the FTL component of SSD still manages the internal garbage collection of SSD. We will show how NVMFS impacts it in section 3.3.3.

Figure 3.6 shows the space organization of SSD. As we see, each extent is 512KB which contains 128 normal 4KB blocks. Given the logical block number, it's easy to get its extent's index and offset within that extent. To facilitate extent recycling,

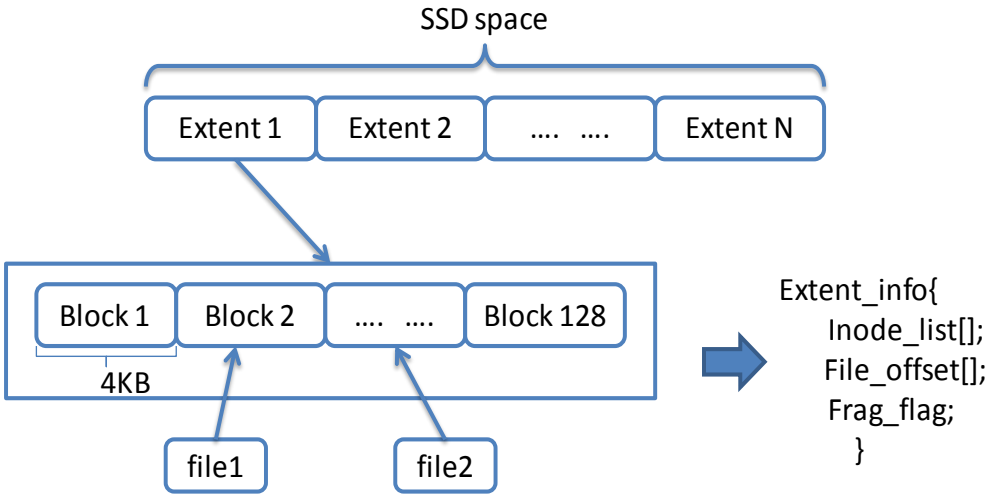


Figure 3.6: Space management on SSD

we need to keep some information for each block within a candidate extent, for example, the inode and file offset each valid block belongs to. We also keep a flag which indicates whether this extent is fragmented (contains invalid blocks). This information is kept as metadata in a fixed space on NVRAM. The space overhead is small, 64 bits (32 bits inode number, 32 bits file offset) for each 4KB block. Whenever extent recycling is invoked, we choose LRU (Least Recently Used) fragmented extents and move the valid data blocks into NVRAM, update their inodes, finally release the extents' space. It's noted that we only free the recycled extent whenever the associated inodes are all updated. In our current design, two conditions have to be satisfied in order to invoke the recycling: (a)the fragmentation ratio of SSD is over a configurable threshold (ideal extent usage/actual extent usage); (b)the number of free SSD extents is fewer than a configurable threshold. The first condition ensures that we do get some free space after recycling whenever the free extents are not sufficient.

3.2.5 File system consistency

File system consistency is always an important issue in file system design. As a hybrid file system, NVMFS stores metadata and hot file data on NVRAM and distributes other relatively cold data on SSD. When the space of NVRAM is not sufficient, LRU dirty data will be migrated from NVRAM to SSD. When the segment cleaning process is activated on SSD, valid data will be migrated from SSD to NVRAM. If system crash happens during the middle of these operations, what is the state of our file system? How do we ensure file system consistency? We will discuss this in detail within this section.

As described in section 3.2.3, NVMFS invokes flushing process whenever the dirty LRU list reaches a high bound (i.e. 80% of dirty LRU list is full). The flushing process chooses 512KB data each round from the end of dirty LRU list and prepares a new SSD extent (512KB), then composes the data as one write request to SSD, finally updates the corresponding metadata. The metadata updating involves inserting the flushed NVRAM pages into clean LRU list and recording the new data positions (on SSD) within "page_info" structure that mentioned in the previous section. It's noted that the inodes (unchanged) still point to valid data on NVRAM until they are replaced from clean LRU list. If system crashes while flushing data to SSD, there is no problem, because inodes still point to valid data versions on NVRAM. We simply drop previous operations and restart migration. If system crashes after data flushing but before we update the metadata, NVMFS is still consistent since inodes point to valid data version on NVRAM. The already flushed data on SSD will be recycled during segment cleaning. If system crashes in the middle of metadata update, the LRU list and "page_info" structure may become inconsistent, NVMFS will reset them. Since the inodes still point to the valid data version on NVRAM,

our file system is consistent. To reconstruct the LRU list, NVRAM scans the inode table, if the inode points to a NVRAM page, we insert it to dirty LRU list while keeping clean LRU list empty. It's noted that if file data have two valid data versions on both NVRAM and SSD, the data version on SSD will be lost since the "page_info" structure are reset now. The corresponding space will be recycled during segment cleaning since no inodes point to those blocks (invalid blocks within extent).

Segment cleaning is another point prone to inconsistency. The cleaning process chooses one candidate extent (512KB) per round and migrates the valid blocks (4KB) to NVRAM, then updates the inodes to point to the new data positions, finally frees the space on SSD. If system crashes during data migration, NVMFS inodes still point to the valid data on SSD. If system crashes during the inodes update, NVMFS maintains consistency by adopting transaction mechanism (inodes update and space freeing on SSD are one transaction) similar to other log-structured file systems.

Another issue is that NVMFS stores metadata and hot data permanently on NVRAM which creates a new challenge: unsure write ordering. The write ordering problem is caused by CPU caches that stand between CPUs and memories [15]. 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. Therefore 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 use a combination of the instructions MFENCE and CLFLUSH to ensure modification of the critical information, including "metadata", "superblock", "inode table", "bitmap" and "directory files", are in consistent ordering. This implementation leverages previous work in [85].

3.3 Evaluation

To evaluate our design, we have implemented a prototype of NVMFS in Linux. In this section, we present the performance of our file system in three aspects: (1)reduced write traffic to SSD; (2)reduced SSD erase operations and erase overhead; (3)improved throughput on file read and write operations.

3.3.1 Methodology

We use several benchmarks including IOZONE [82], Postmark [38], FIO [32] and Filebench [19] to evaluate the performance of our file system. The workloads we choose all have different characteristics. IOZONE creates a single large file and performs random writes on it. For Postmark, the write operations are in terms of appending instead of overwriting. FIO performs random updates on randomly opened files chosen from thousands of files. Filebench does mixed read and write on thousands of files which simulate a file server.

In the experimental environment, the test machine is a commodity PC system equipped with a 2.8GHz Intel Core i5 CPU, 8GB of main memory. We also attached 4GB Nonvolatile DIMMs [1, 79] as the NVRAM. The NAND flash SSD we used is Intel's X25-E 64GB SSD. The operating system used is Ubuntu 10.04 with a 2.6.33 kernel.

In all benchmarks, we compare the performance of NVMFS to that of other existing file systems, including EXT3, Btrfs, Nilfs2 and Conquest (also a hybrid file system) [80]. The first three file systems are not designed for hybrid storage architecture. Therefore we configure 4GB DRAM-based page cache for them. While for Conquest file system, we implemented it according to the description in [80]. In [80], it is said that a larger threshold for small files will keep more files on NV-memory and achieve better performance. In our implementation of Conquest, we

define small file as one with file size less than 128KB. The reason we made this decision is that we found that with a threshold larger than 128KB, we will not be able to allocate all the small files on NVRAM under our workload. The reason we choose these file systems is as following. EXT3 is a popular file system used in Linux operating system. Btrfs [8] implements some optimizations for SSDs (we mount btrfs with "nodatacow" option to get the best available performance). Nilfs2 [56] is a log structure file system which is designed for NAND flash. Finally, Conquest is also a hybrid file system which utilizes both NVRAM and HDD/SSD as the storage.

3.3.2 Reduced i/o traffic to solid state drive

In this section, we calculated how much IO data are written to SSD while running different workloads for our NVMFS and other file systems. As explained in section 2.2, our NVMFS persistently keeps metadata and hot file data on NVRAM without writing to SSD. However, other file systems have to periodically flush dirty data from page cache to SSD in order to keep consistency. Therefore, NVMFS is expected to reduce write traffic to SSD.

Figure 3.7 shows the write traffic to SSD (number of sectors) across different workloads. For all the workloads, the IO request size is 4KB. We can see our file system has less write traffic to SSD across all the workloads. For Filebench workload, the reduction is about 50% compared to other file systems. To further explore this, we calculated how many IO requests are satisfied by memory (NVRAM or Page Cache) under different file systems. Figure 3.8 shows the hit ratio across different workloads, which is the number of IO requests satisfied by memory divided by the total number of IO requests. We can see our file system has higher hit ratio across all the workloads except Postmark. As a result, other file systems have to evict more data from page cache to SSD due to cache replacements. For Postmark workload,

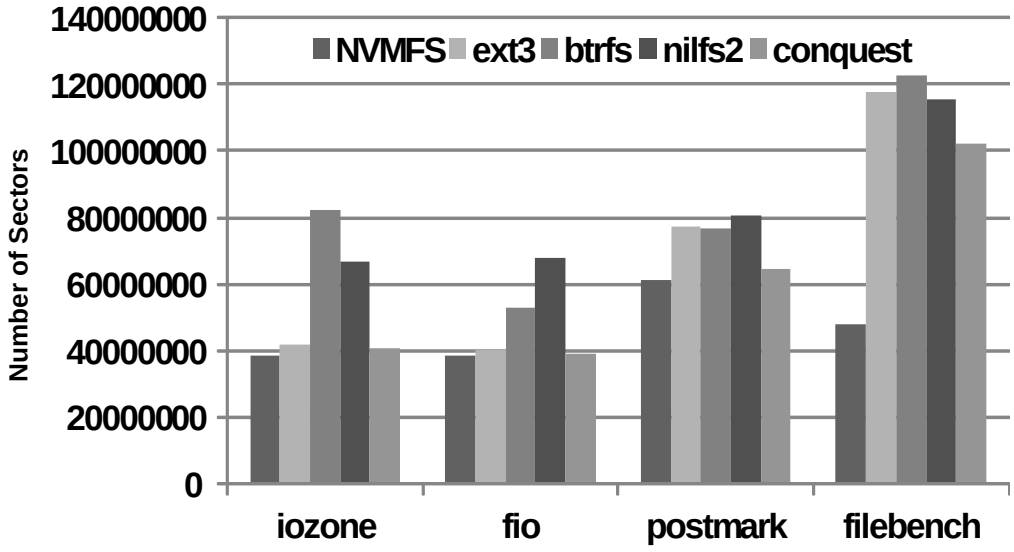


Figure 3.7: Write traffic to SSD under different workloads and file systems

although our hit ratio is slightly lower than ext3 and btrfs, we still write less data to SSD. This is because we permanently store metadata on NVRAM, which saves many writes to SSD.

Conquest only stores small files on NVRAM while keeping large files on disk. Therefore, for Conquest, IO requests to large files still go through page cache and need to be synchronized with SSD. However, our file system can store the frequently accessed portion of large files on NVRAM permanently. Figure 3.9 shows the write traffic to SSD while running the original and our modified IOZONE workloads. Both the workloads create a large file, then perform random writes on it. The difference is that the modified IOZONE writes randomly only to the first 3GB of the large file. We can see our file system further reduced write traffic by keeping parts of that large file on NVRAM. Because our file system will permanently store the frequently accessed portion of the large file on NVRAM, while Conquest and other file systems need to periodically flush dirty data from page cache to SSD for consistency, NVMFS

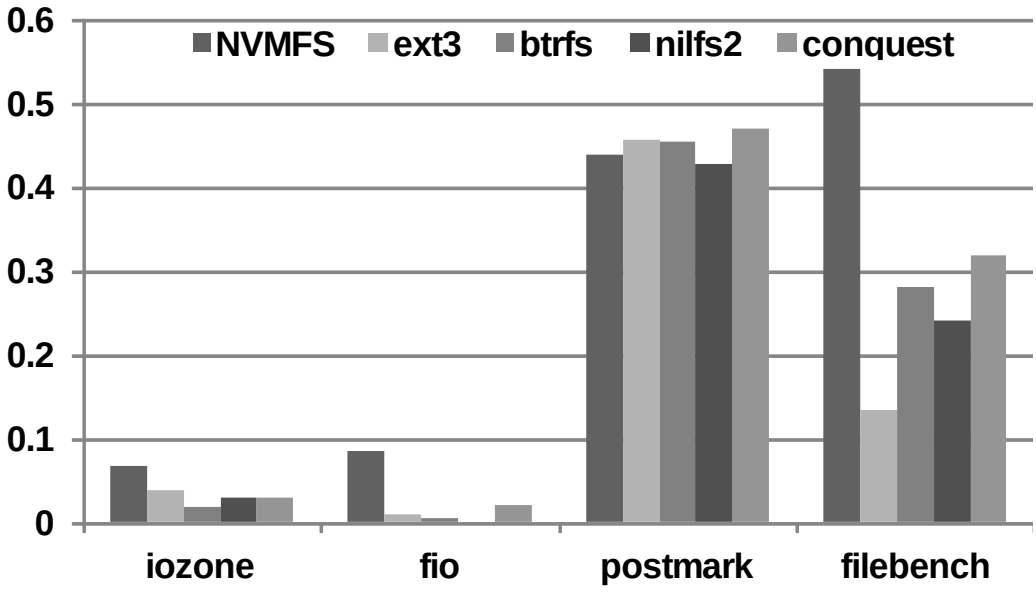


Figure 3.8: Hit ratio on memory

achieves better performance.

3.3.3 Reduced erase operations and overhead on solid state drive

The erase operations on SSD are quite expensive which greatly impact both lifetime and performance. The overhead of erase operations are usually determined by the number of valid pages that are copied during the GC (Garbage Collection).

To evaluate the impact on SSD's erase operations, we collected I/O traces issued by the file systems using blktrace [31] while running our workloads described in section 3.3.1, and the traces were run on an FTL simulator, which we implemented, with two FTL schemes -(a)FAST [48] as a representative hybrid FTL scheme and (b)page-level FTL [39]. In both schemes, we configure a large block 24GB NAND flash memory with 4KB page, 256 KB block, and 10% over-provisioned capacity. Figure 3.11 and 3.12 show the total number of erases and corresponding erase cost for the workload processed by each file system.

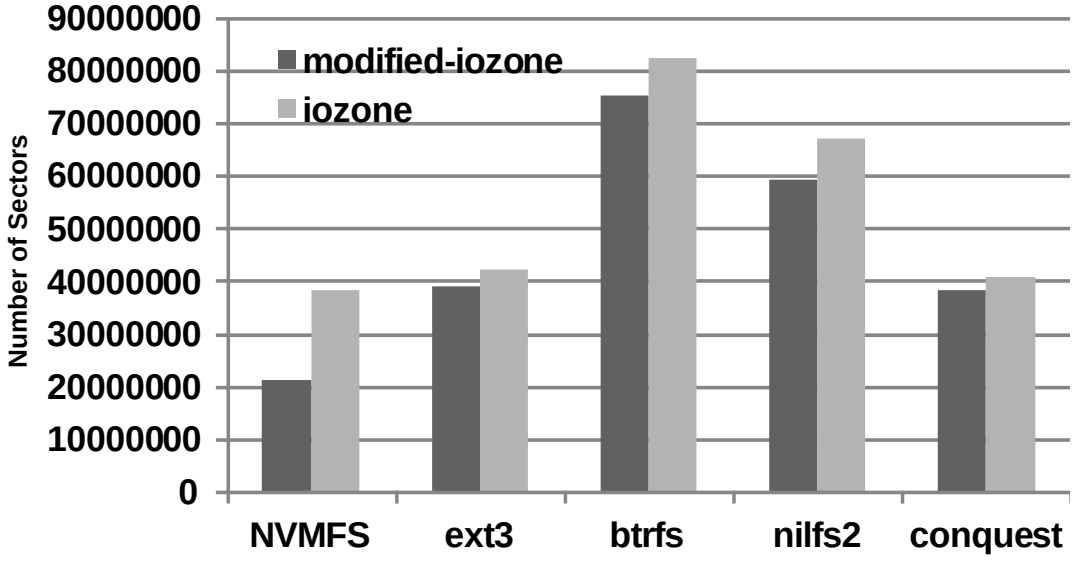


Figure 3.9: Write traffic to SSD under modified iozone

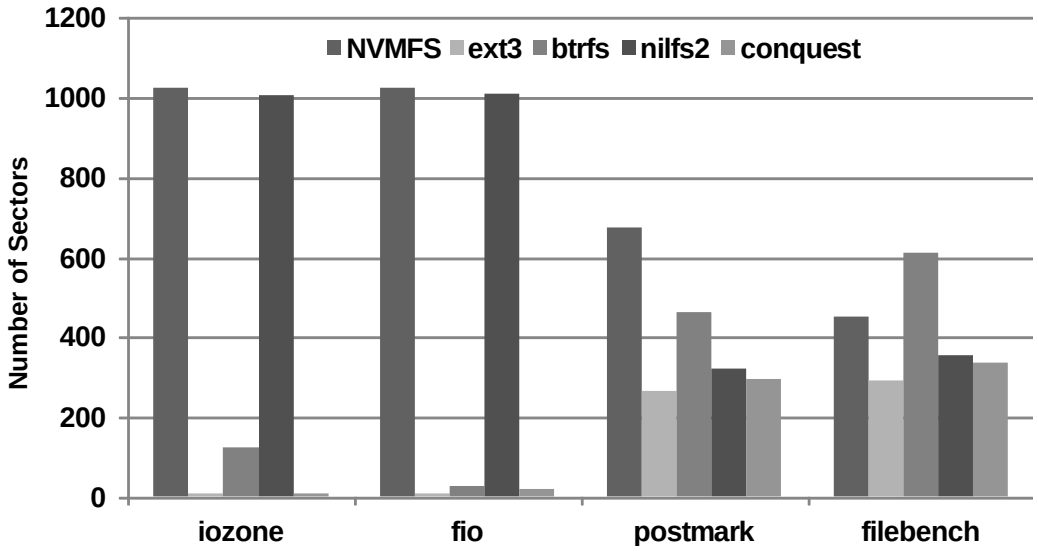
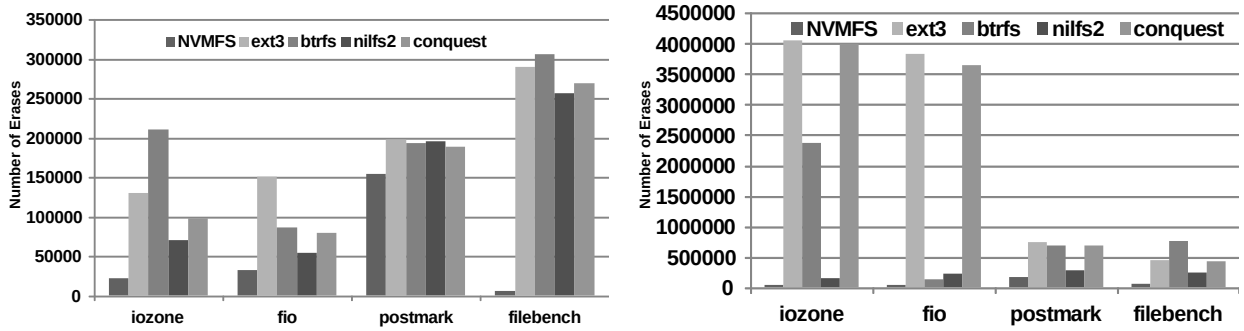
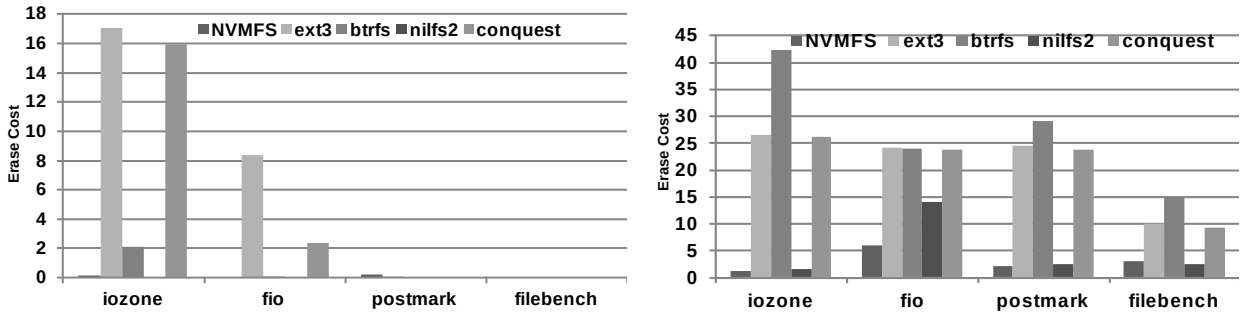


Figure 3.10: Average I/O request size issued to SSD under different workloads and file systems



(a)erase count for page-level FTL (b)erase count for FAST FTL

Figure 3.11: Erase count for page-level and FAST FTL



(a)erase cost for page-level FTL (b)erase cost for FAST FTL

Figure 3.12: Erase cost for page-level and FAST FTL

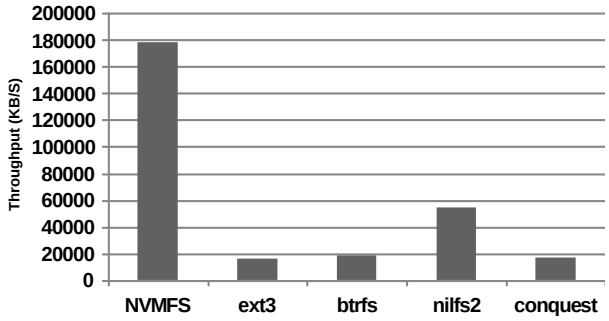
We can see that NVMFS has fewer number of erases under all situations. Our benefits come from two aspects. For Filebench workload, we saved 50% write traffic to SSD as described in section 3.3.2, thus, we see much fewer erases on SSD. For IOZONE and FIO workloads, we transformed random writes to sequential ones at SSD level, which is similar to log structured file system (such as nilfs2). This design also helps reduce erase overhead. As shown in figure 3.10, for IOZONE and FIO workloads, our file system and nilfs2 transform random writes to sequential ones at SSD level, thus we observed larger IO request size compared to other file systems. Postmark and Filebench workloads have some locality in accesses. As a result, OS scheduler can merge adjacent IO requests into large ones before issuing to SSD, thus, we see relatively large IO request size across all the tested file systems.

To explore the erase cost, we calculated the average number of pages (valid pages) copied during GC. Figure 3.12 shows the erase cost which is the average number of pages migrated during GC. For page-level FTL, both NVMFS and nilfs2 have very few valid pages within the erase blocks for most cases. For hybrid FTL scheme, NVMFS also performs better than other file systems.

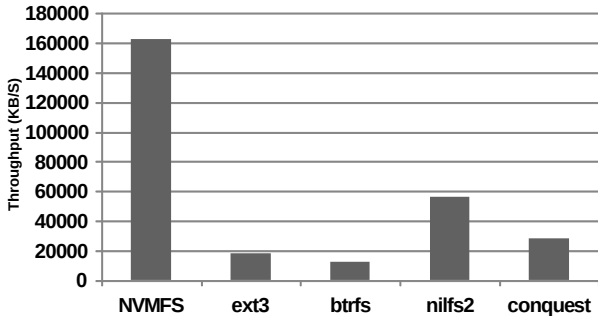
3.3.4 Improved i/o throughput

In this section, we evaluate the performance of our file system in terms of IO throughput. We use the workloads described in section 3.3.1. For our file system and nilfs2, we measure the performance under both high (over 85%) and medium disk utilizations (50%-70%) to evaluate the impact of segment cleaning overhead. The segment cleaning is activated only under high disk utilization. For other file systems that do in-place update on SSD, there is little difference for varied disk utilizations.

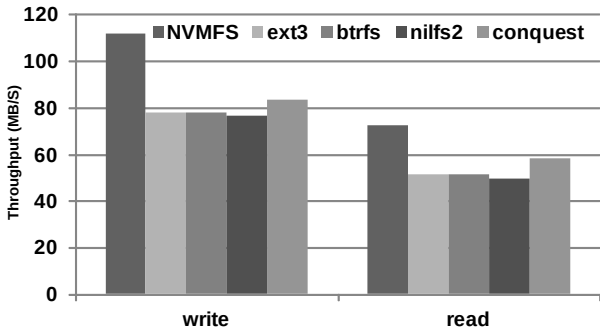
Figure 3.13 shows the IO throughput while the segment cleaning is not activated with our file system and nilfs2. We can see our file system performs much better



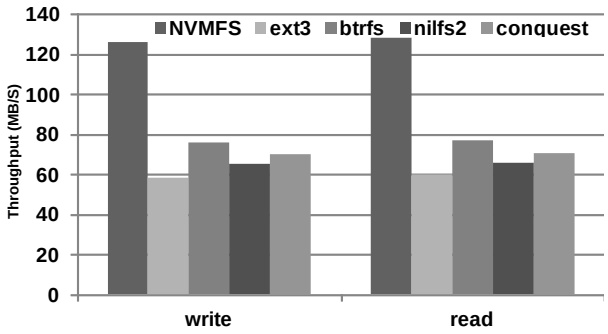
(a)Throughput of IOZONE



(b)Throughput of FIO

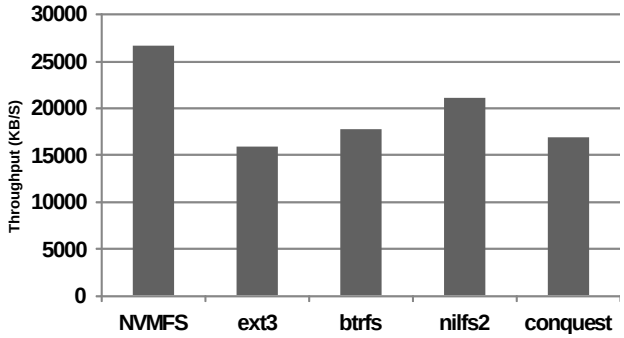


(c)Throughput of Postmark

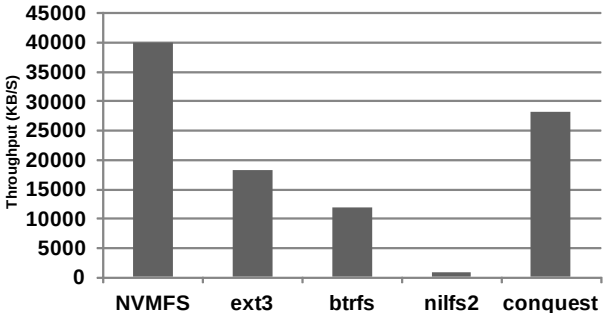


(d)Throughput of Filebench

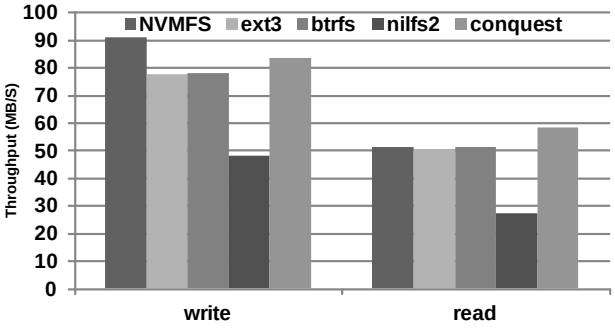
Figure 3.13: I/O throughput under different workloads for 50% - 70% disk utilization



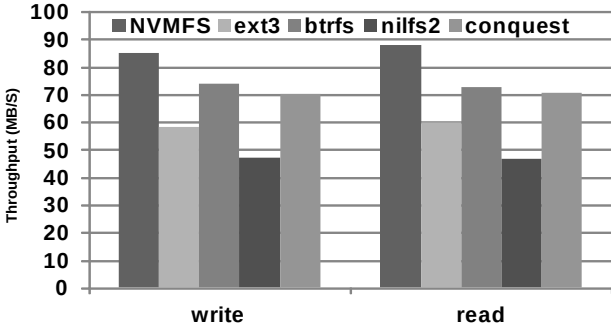
(a)Throughput of IOZONE



(b)Throughput of FIO

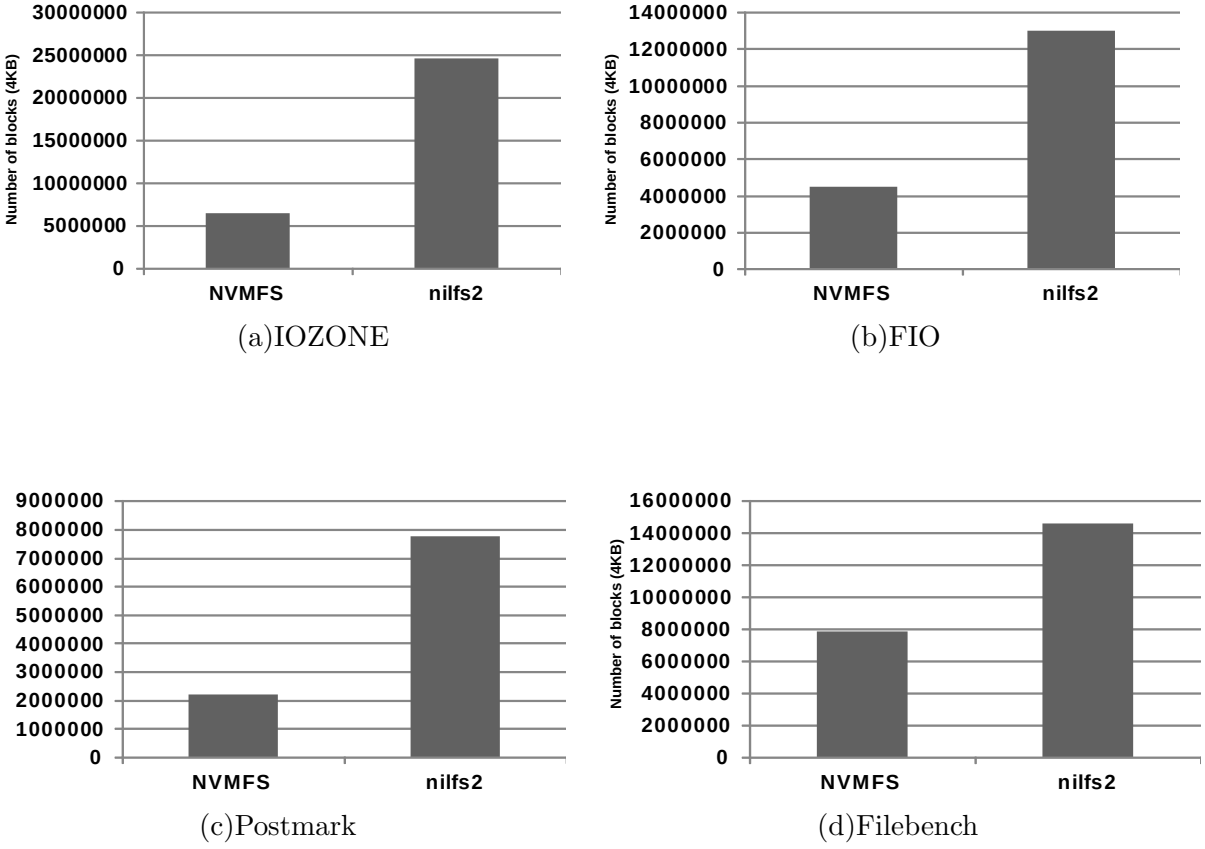


(c)Throughput of Postmark



(d)Throughput of Filebench

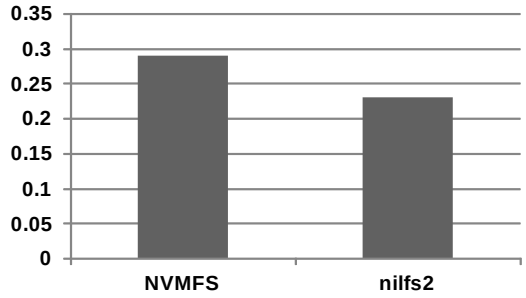
Figure 3.14: I/O throughput under different workloads for over 85% disk utilization



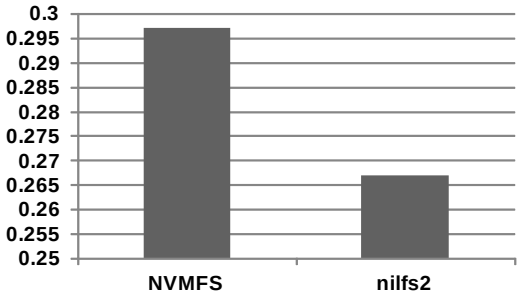
(a)IOZONE (b)FIO

(c)Postmark (d)Filebench

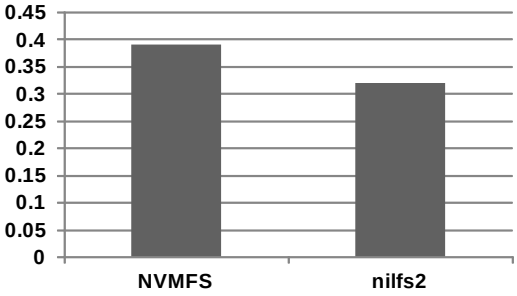
Figure 3.15: Total number of recycled blocks while running different workloads under NVMFS and nilfs2



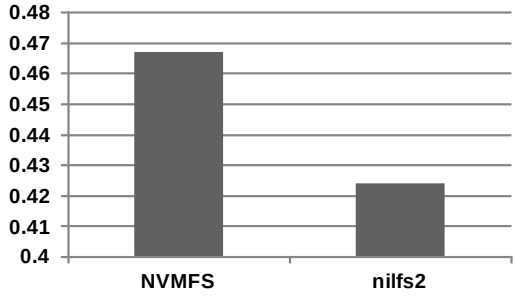
(a)IOZONE



(b)FIO



(c)Postmark



(d)Filebench

Figure 3.16: Cleaning efficiency while running different workloads under NVMFS and nilfs2

than all the other file systems across all the workloads. Compared with in-place update file systems, NVMFS transforms small random writes to large sequential writes on SSD, therefore improves the write bandwidth significantly. Compared with log-structured file system such as nilfs2, NVMFS stores hot data on NVRAM and better groups dirty data before flushing to SSD, as a result, the erase overhead of SSD is reduced. We also noticed that Conquest, another hybrid file system, did not perform well under IOZONE and FIO workloads. For IOZONE benchmark, Conquest permanently stored the single large file on SSD and used page cache to temporarily buffer the write accesses, similar to EXT3. Compared with our file system and nilfs2, Conquest still performed random writes on SSD. For FIO benchmark, Conquest put small files (smaller than 128KB) on NVRAM, however, for large files, random writes were still performed on SSD.

To evaluate the impact of segment cleaning on our file system and nilfs2, we also measured the performance under high disk utilization (over 85%). Figure 3.14 shows the throughput when disk utilization is over 85% for all the tested file systems and workloads. We can see obvious performance reduction for both NVMFS and nilfs2, while other file systems have little change compared with that under 50%-70% disk utilization. Compared with nilfs2, our file system performs much better across all the workloads, especially under FIO workload. To further explore this, we calculated the number of blocks (4KB) recycled and the cleaning efficiency while running different workloads under NVMFS and nilfs2. For cleaning efficiency, we measure it using the formula "1 - (moved_valid_blocks / total_recycled_blocks)".

Figure 3.15 and 3.16 show the total number of recycled blocks and the cleaning efficiency respectively while running different workloads under NVMFS and nilfs2. We can see for all the workloads NVMFS recycled much fewer blocks compared with nilfs2, which means we generate much fewer background IOs. This is because

NVMFS absorbs many small IOs on NVRAM and avoids many writes to SSD which relieves the pressure of segment cleaning on SSD. As a result, forefront IO workloads are less impacted by NVMFS compared to nilfs2. Another benefit of NVMFS is that when we move valid blocks from SSD to NVRAM due to recycling, we can free the corresponding space on SSD once the data reside on NVRAM. However, nilfs2 has to wait until the valid blocks are written back to new segments on SSD, otherwise they may lose consistency for power failure. As shown in figure 3.16, we also see NVMFS has higher cleaning efficiency relative to nilfs2. This is benefit from our grouping policy on dirty data before flushing to SSD.

3.4 Related work

A number of projects have previously built hybrid storage systems based on non-volatile memory devices [44, 36, 59, 80]. [44] and [59] proposed using a NVRAM as storage for file system metadata while storing file data on flash devices. FRASH [36] harbors the in-memory data and the on-disk structures of the file system on a number of byte-addressable NVRAMs. Compared with these works, our file system explores different write policies on NVRAM and SSD. We do in-place updates on NVRAM and non-overwrite updates on SSD.

Rio [14] and Conquest [80] use a battery-backed RAM in the storage system to improve the performance or provide protections. Rio uses the battery-backed RAM to avoid flushing dirty data to disk. Conquest uses the nonvolatile memory to store the file system metadata and small files. WSP [54] proposes to use flush-on-fail technique, which leverages the residual energy of the system, to flush registers and caches to NVRAM in the presence of a power failure. Our work here explores nonvolatile DIMMs to provide a highly reliable NVRAM that runs with the latency and endurance of the fastest DRAM, while also having the persistence of Flash. In

the eNVy storage system [84], the flash memory with a battery-backed RAM buffer is attached to the memory bus to implement a non-volatile memory device. Our work assumes that nonvolatile memory is large enough for both data and metadata and uses dynamic metrics to determine what data is retained in NVRAM. Moreover, our file system transforms random writes to sequential ones at SSD level which can effectively reduce SSD's erase overhead and improve SSD's lifetime.

The current SSDs implement log-structured like file systems [69] on SSDs to accommodate the erase, write operations of the SSDs. Garbage collection and the write amplification resulting from these operations are of significant interest as the lifetime of SSDs is determined by the number of program/erase cycles [29]. Several techniques have been recently proposed to improve the lifetime of the SSDs, for example [13, 26]. The recent work SFS [10] proposed to collect data hotness statistics at file block level and group data accordingly. However, they were restricted to exploit this information within a relatively short time slice, since all the dirty data within page cache have to be flushed to persistent storage in a short time. Our work here exploits the NVRAM to first reduce the writes going to the SSD and second in grouping similar pages into one block write to SSD to improve garbage collection efficiency.

Several recent studies have looked at issues in managing space across different devices in storage systems [12, 24]. These studies have considered matching workload patterns to device characteristics and studied the impact of storage system organizations in hybrid systems employing SSDs and magnetic disks. Our hybrid storage system here employs NVRAM and SSD. Another set of research work proposed different algorithms for managing the buffer or cache for SSD [83, 42, 33, 73]. They all intended to temporally buffer the writes on the cache and reduce the writes to SSD. Our work differs from them since our file system can permanently store the data on

NVRAM, thus further reducing writes to SSD.

Much research has been focused on FTL design to improve performance and to extend lifetime of SSDs [43, 25, 49, 48]. Three types of FTL schemes are proposed including block-level mapping, page-level mapping and hybrid mapping that trades-off the first two. The block-level FTL maps a logical block number to a physical block number and the logical page offset within that block is fixed. This scheme can store the entire mapping table in memory since it is small. However, such coarse-grained mapping results in a higher garbage collection overhead. In contrast, a page-level FTL manages a fine-grained page-level mapping table, thus has lower garbage collection overhead. While page-level FTL requires a large mapping table on RAM which cost more on hardware. To overcome such technical challenges, hybrid FTL schemes [43, 49] extend the block-level FTL. These schemes logically partition flash blocks into data blocks and log blocks. The majority of data blocks are using block-level mapping while the log blocks are using page-level mapping. Our work can reduce the erase overhead during GC (Garbage Collection) which benefits various FTL schemes.

4. SPACE MANAGEMENT OF SSD BASED SECONDARY DISK CACHES

NAND-flash based solid state disks provide us better performance than traditional hard disks, however, their prices are also higher than hard disks. Moreover, compared with hard disks, SSDs have smaller capacity and limited lifetime, which slow down their deployment as the persistent storage in enterprise storage systems. To explore SSD devices' performance benefit, we can add a new tier into the storage hierarchy - secondary disk cache, which cache data in front of high-capacity hard disks. For large storage servers, we might employ multiple SSDs at the cache layer. How to manage the space of the secondary disk caches is an issue we need to solve. A straightforward way is to build a RAID volume composed of all the SSDs that are used as disk caches. In this chapter, we extend an existing secondary disk cache design to port to the multi-device case and manage the cache space efficiently to improve performance. Compared with RAID based cache management, our scheme maintains the data reliability as well as improving the cache performance.

4.1 Background

High performance storage systems are currently in high demand for data-intensive computing. The speed of reading and writing data to the storage system might directly affect the execution time of the applications and the performance of the whole system. However, most storage systems, are still using traditional hard disk drives (HDDs). Although solid state disk (SSD) has become a mature technology which demonstrates better performance than hard disk, it's still not cost-effective to replace all the hard disks, especially for systems that require large amount of storage space. For example, San Diego Supercomputer Center (SDCS) has built a large flash-based cluster, called Gordon, which adopts 256TB of flash memory as its storage [12].

However, this design is backed by a $20 million funding from the National Science Foundation (NSF) and may not be a cost-effective solution for employing SSDs. Moreover, data accesses normally exhibit locality within the storage systems which gives us opportunity to cache frequent used data on high-performance storage devices and build a cost-effective, hierarchical, tiered storage system.

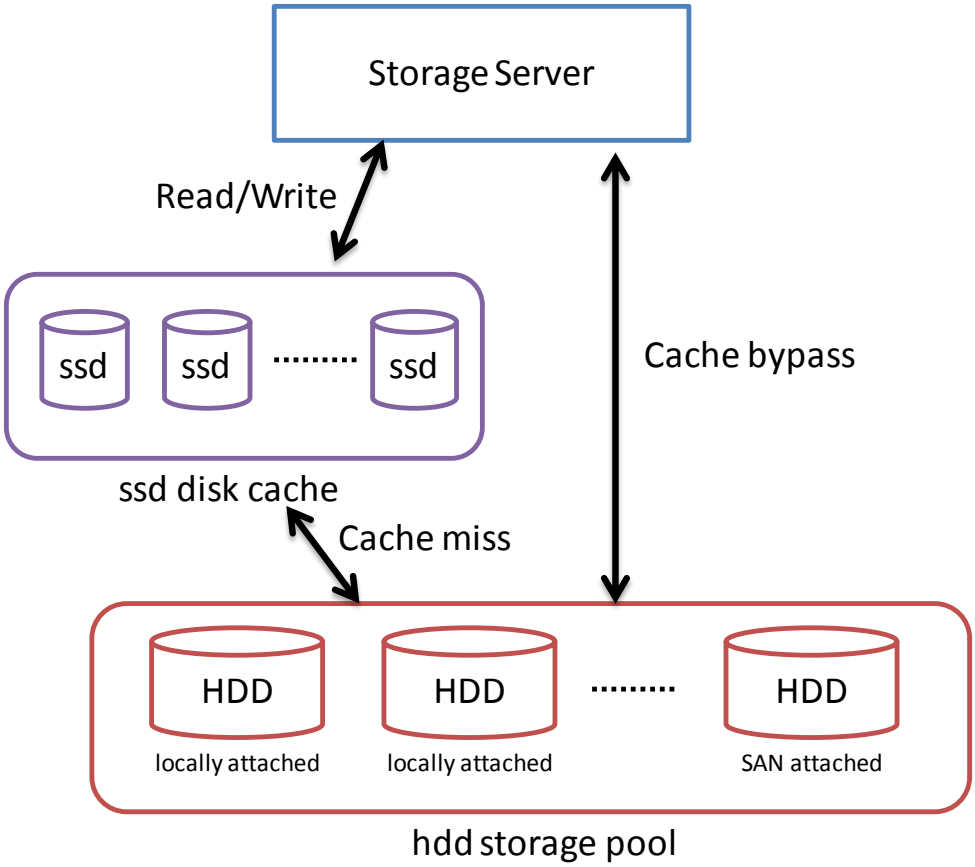


Figure 4.1: SSD cache based disk storage system.

Considering SSDs' performance and cost are in between of DRAM memory and HDDs, a straightforward way to employ SSDs is to add a new tier into the storage hierarchy — secondary disk cache. Figure 4.1 shows the storage hierarchy of a server

which integrates secondary disk caches built by multiple SSDs. As shown in figure 4.1, normally the read and write requests are first forwarded to the SSD-cache layer, if they hit in the cache, we directly return the data or write the updates to the SSD caches. Otherwise we redirect the desired requests to the target HDDs. Under some conditions, we might want to bypass the SSD-cache layer. For example, when there are too much dirty data at the cache layer, it's better to write the updates directly to the HDDs so that we can reduce the data synchronization pressure at the cache layer.

There are several secondary disk cache designs proposed previously [41, 87, 70]. In this chapter, we focus on a newly released cache design — Bcache [40], which is specially designed for flash based SSDs.

Bcache partitions the storage on the SSD device into buckets. All buckets on the SSD device have the same size (128KB - 4MB), which is expected to be the native erase block size of the SSD (or multiple times of the erase block size). Within a bucket, bcache writes data sequentially following a log-style and only once. Once the bucket is fully written, it's sealed and won't allow further overwrites. When bcache needs to reclaim space on SSD, it reclaims a bucket and sends the device a discard command covering the entire bucket before it writes any incoming data into this bucket. This allows bcache to cooperate with SSD's internal garbage collection mechanism to better utilize SSD's space. Bcache implements a very fast index. On every IO request to the cache layer, bcache can quickly determine whether it's a hit or not and where the requested date is located if it's a hit. The data structure used to implement the index is a B+ tree with each node storing more than one key set. Keys are ordered within key sets but all key sets must be scanned from the newest set towards the oldest when searching for a key. Keys are never removed from a node until such a node is either split or reclaimed by the garbage collection

mechanism. Only leaf nodes point to data while all internal nodes point to other B+ tree nodes, which is the same as a traditional B+ tree. There are two main types of write policies supported by bcache: write-back and write-through. In write-back mode, the dirty flag on the key is set to indicate that the data is not available (updated) on the HDDs. All read accesses to that data have to be satisfied by SSD caches. Any new writes to that data replace the previous version by inserting a new data version on SSD caches. At some point, the dirty data must be written to the storage device. Normally this is done by a background write-back process. Bcache has several tunable parameters that control write-back. In write-through mode, a write is not considered completed until it reaches the target HDDs. Therefore, all the data are clean in bcache if it is configured as write-through mode.

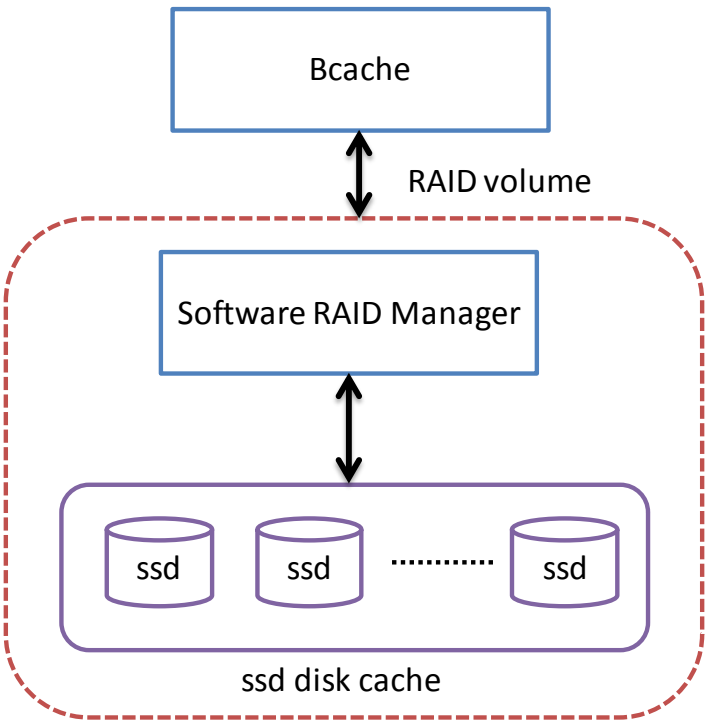


Figure 4.2: Bcache based on RAID in a multi-device environment.

The design of bcache has incorporated several optimizations considering flash-SSD's special characteristics, for example, the bucket-level space reclaiming, log-style data filling within the bucket. However, bcache's current implementation cannot be directly ported to a multi-device environment. In order to achieve this, we have to build the multiple SSD devices as a RAID volume and export it to the bcache as a single logical device. As shown in figure 4.2, a software RAID manager is sitting below the bcache layer and responsible for managing the cache space of the multiple SSD devices. The RAID volume exported to bcache is viewed and handled as a single device. The benefit of this approach is that it is simple to deploy since it does not require any adaptation at the bcache layer. However, the RAID system is initially designed for persistent storage and is not necessarily optimized for a cache environment. Normally we have no choice on selecting what kind of data should be protected, for example, either protect none (i.e. RAID0) or protect all (i.e. RAID1). Most storage systems apply full protection to the data stored at the HDDs to avoid any data loss during failures. For a disk cache, this might not be necessary since any data that are marked as clean should have their identical copies stored at the HDDs. Based on this observation, we claim it's better to remove the cache-unaware RAID layer and manage the multiple SSD devices directly inside the bcache. In our design, we extend existing bcache module to be multi-device aware and selectively protect the stored data at the disk cache layer. We assume that the SSD caches are used in write-back mode which provides us greater potential for improving performance. We might have both clean and dirty data on the SSD caches. For the clean data, we keep only one copy since there are identical data versions on the HDDs. During a SSD cache failure, we can still obtain the clean data from HDDs. However, for the dirty data, we employ full-redundancy and keep additional data copies on different SSD devices. When an SSD fails we can still obtain the dirty data from other SSDs

which store the duplicated copies. It is noted that we cannot recover the dirty data from the HDDs since they are out of date.

The primary contributions of our design are as following:

- We extend the existing bcache module to be multi-device aware and selectively protect the data stored at the cache layer.
- We balance the write traffic to each SSD device which prevents any SSD from wearing-out much earlier than others.
- We show that our design improves the space utilization efficiency of the SSDs, as a result, both hit ratio and system throughput are increased.

4.2 Design and implementation

In this section, we describe the extensions we have added to the existing bcache module to better manage the cache space in a multi device environment. We allocate the incoming data across the SSD devices in a round robin fashion. This approach helps us balance the overall writes to each SSD, so that the SSD devices wear out at a uniform rate. As a result, no single SSD will wear out much earlier than the others.

4.2.1 Multi-device aware caching

In order to make the bcache module aware of the multiple SSD devices of the disk cache space, we extend the existing data structure of the so called "cache set". For current implementation of bcache, the "cache set" only consists of one cache device. In our extension, we define a cache set to be a collection of one or more cache devices. Within the "cache set" data structure, we have a cache array which stores the pointers to each individual cache device descriptor. Each cache device has a data structure called "cache" to maintain all the information related with this SSD.

As shown in figure 4.3, our extensions enable the bcache to manage the multiple SSD devices directly without importing the RAID software.

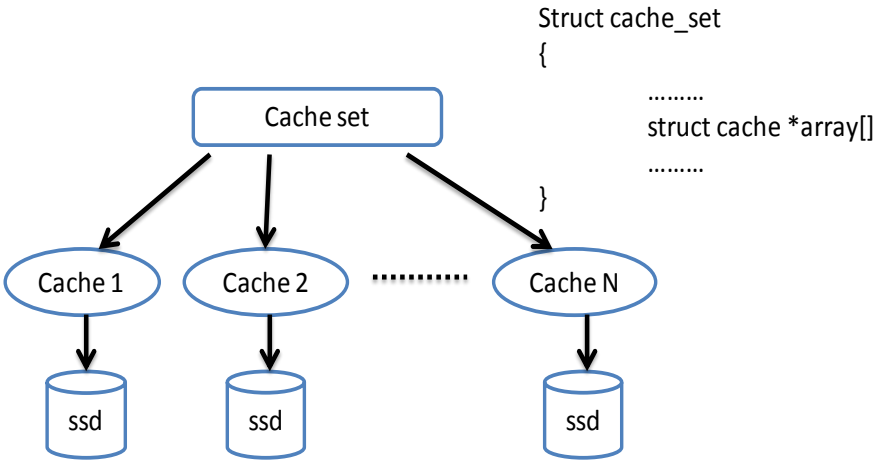


Figure 4.3: Extensions made to the cache set structure of bcache.

When we build the bcache caching layer, we need to format each individual SSD device according to the desired pattern. This is done by writing a super-block to the specific SSD device. The super-block will be read later while registering the caching devices to the operating system. To explicitly manage multiple SSD devices, we also changed the super-block's information and defined a device id for each registered SSD. The device id is increased sequentially whenever there are new SSD devices added to the cache layer and reported to the bcache module. Through the device id, we can manage and differentiate all the registered SSD devices. To refer to a specific SSD device, we can simply obtain its device description by indexing into the device pointer array (in "cache set" data structure) using the device id. When we want to detach the bcache cache devices, we need to unregister the "cache set" as well as each individual SSD device.

4.2.2 Flexible data redundancy

When we build a RAID volume to manage the multiple devices, we have no flexibility on choosing what kind of data will be protected. Normally there is only one option — either none of the stored data are protected or all of the stored data are protected. For cache storage, clean data already have their identical copies stored on HDDs, it's unnecessary to keep another copy at the cache layer which will potentially waste the cache space. However, for dirty data, it's desired to store additional copies at the cache layer among different SSDs. We cannot recover the dirty data from the HDDs since the data versions on HDDs are out of date.

We changed the existing data insertion logic of bcache module and considered additional situations for choosing what kind of data to duplicate. Currently we only consider a simple case, which chooses to duplicate dirty incoming data and write them to different SSD devices. So that even if one SSD device fails, we can still recover all the dirty data on that SSD from the other SSD devices. When the dirty data are written back to the HDDs, the original data and the duplicated copies will become both clean. We mark one of them as invalid to recycle the used space. For clean data, they can be obtained from the HDDs whenever they are requested in future. The benefit of doing this is that we can improve the cache space utilization while ensuring the data reliability.

4.2.3 Balance the writes among solid state drives

When there are multiple SSD devices within the cache layer, we need to consider the aging problem of each individual SSD. SSD has limited write endurance and will lose data afterward. Normally we can have two device replacement modes: (a)incrementally replace the device that is near the failure point; (b)replace all the devices at once if any device is nearly worn out. The former one assumes that each

device has different wear-out speed and might invoke frequent replacement activity from time to time. The latter one assumes each device is following a similar wear-out pattern and each replacement event will involve all the storage devices.

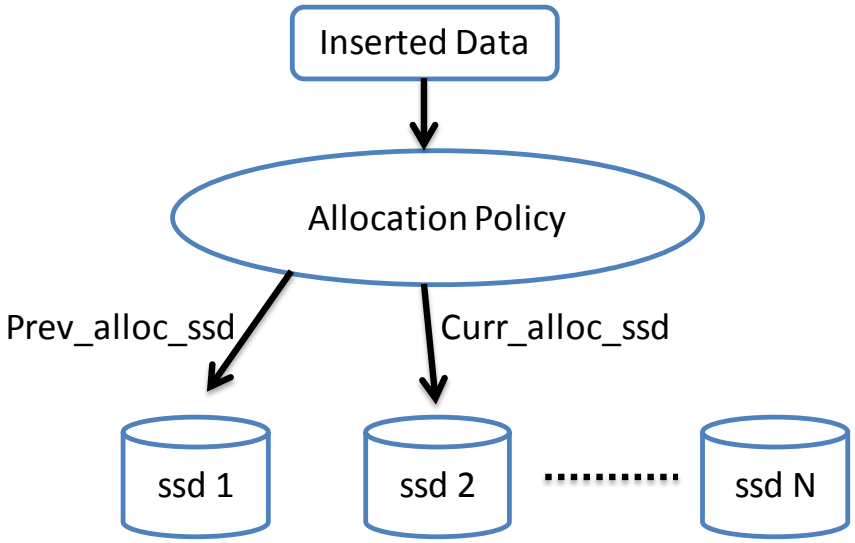


Figure 4.4: Insert data to SSDs based on round-robin style.

In our design, we assume using the second device replacement mode at the SSD cache layer. To ensure each SSD device gets aged at similar speed, we need to balance the write traffic among all the SSDs. We achieve this by explicitly inserting the incoming data to SSDs in a round-robin style. As shown in figure 4.4, we have two pointers to track the previous and current devices where data are allocated, which will be updated accordingly. This approach ensures that the write traffic is distributed more or less uniformly among all the SSD devices.

4.3 Evaluation

To evaluate our design, we have added our extensions to the current implementation of bcache module in Linux. In this section, we present the performance of

our design by comparing with the original bcache module which is built on a RAID1 volume.

4.3.1 Methodology

We use two workloads to evaluate our design. The first workload is write intensive with read/write ratio equal to 1:1. The second workload is read intensive with read/write ratio equal to 9:1. Both workloads employ 4 concurrent threads, which issue 4KB IO requests to one large file. The file is initially stored on the HDD. Moreover, both workloads follow a Zipfian distribution [81] with theta equal to 0.5. The reasons we choose these two workloads are that (a)the Zipfian distribution can demonstrate sufficient data locality for evaluating the cache system; (b)the different configurations of read/write ratios can let us know the performance of our design under different IO request patterns. It is noted that we wait enough time to warm up the SSD caches before starting our performance measurements.

In the experimental environment, the test machine is a storage server equipped with two Intel MLC 240GB SSDs and one WD 2TB HDD. The operating system used is Fedora 16 with a 3.10 Linux kernel. In all workloads, we compare the performance of our design with the original bcache module that is built on top of a RAID1 volume (two SSDs). For our design, we manage the two SSDs directly at the cache layer without the additional RAID1 layer.

4.3.2 Results

We measure the performance using two metrics: the overall throughput of the storage system and the cache hit ratio of SSDs. Figure 4.5 and figure 4.6 show the two metrics for our design and the bcache-RAID1.

We can see that our design improved the throughput by 20% to 30% compared to the original bcache module that is built on RAID1. It's noted that the improve-

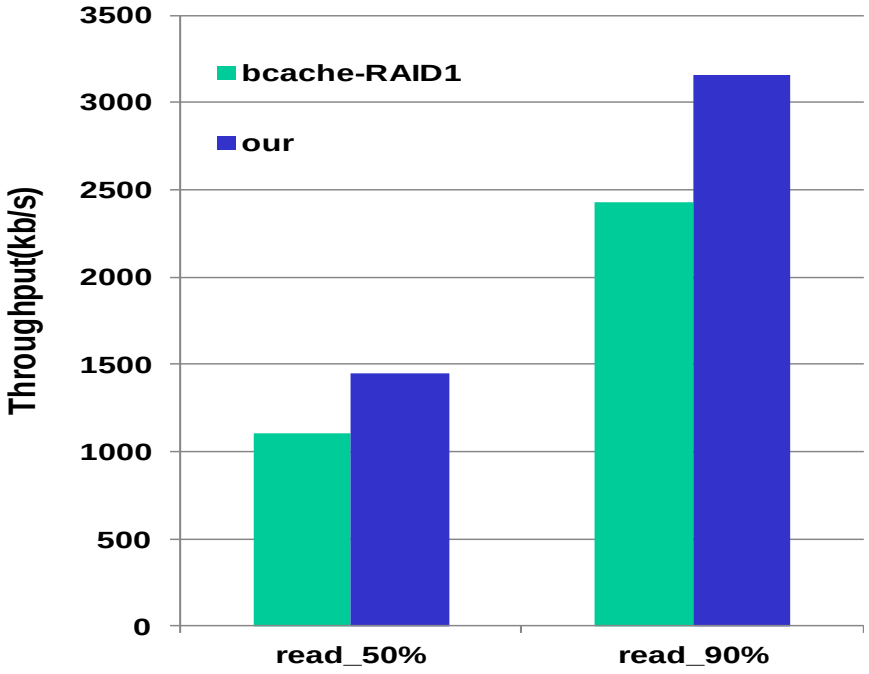


Figure 4.5: The throughput of the storage system while running different workloads.

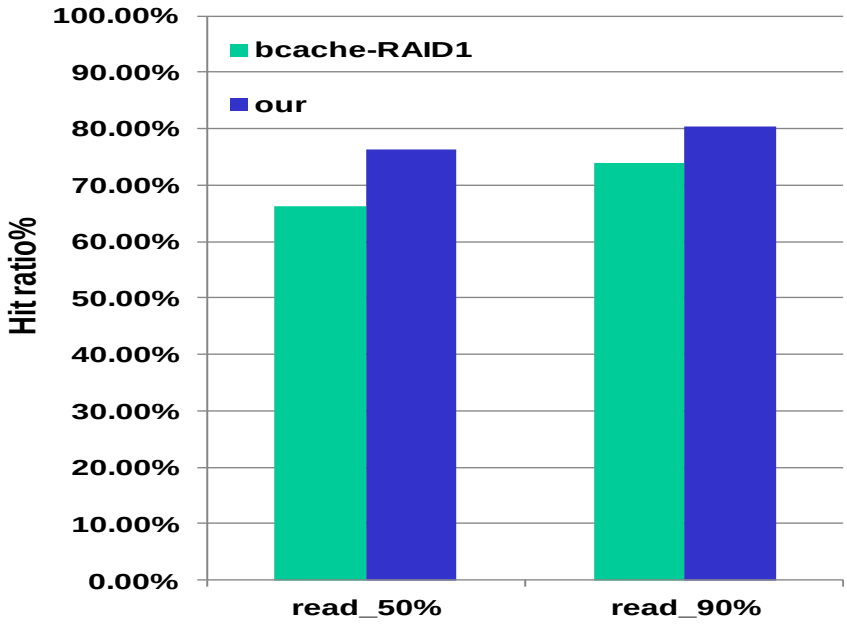


Figure 4.6: The hit ratio of SSD caches while running different workloads.

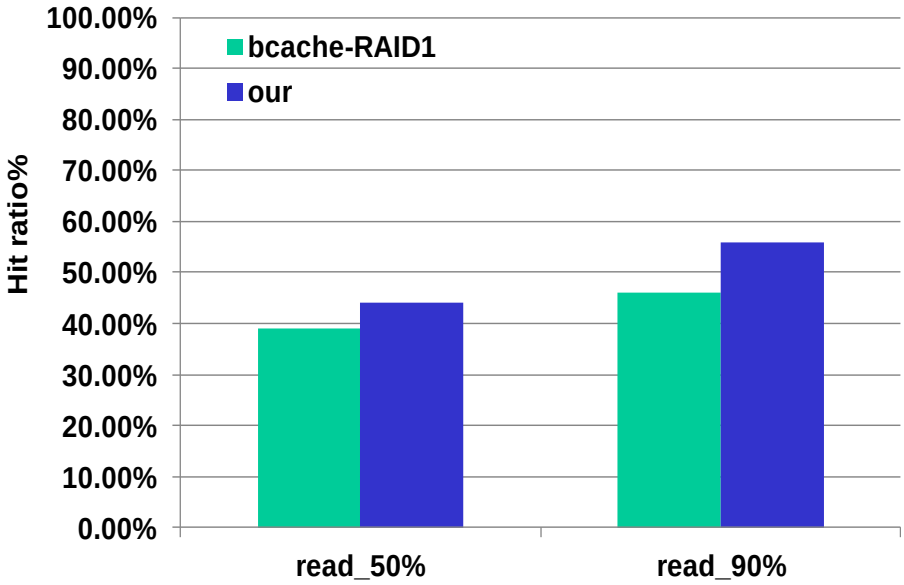


Figure 4.7: The hit ratio of SSD caches after reducing the cache size.

ment is greater under the read-intensive workload. This is because for read-intensive workload, lots of the data on SSD caches are marked as clean which won't be duplicated. Therefore we can avoid more duplicate copies which allows us to cache more data on SSDs. As a result, we also achieved better hit ratio than the unchanged bcache-RAID1 configuration which is shown in figure 4.6.

To further investigate the impact of SSD cache size on hit ratio, we reduced the total cache size by 30% for both our design and the bcache-RAID1 configuration (we run this setup on a different machine). We run the same workloads as before. Figure 4.7 shows the cache hit ratio after reducing the cache size. We cam see that the hit ratio is lower than the result shown in figure 4.6 since the cache size is reduced for both our design and bcache-RAID1. However, our design still improved the cache hit ratio compared to the bcache-RAID1 scheme.

Our design extended the existing bcache module to the multi-device case. Compared with bcache-RAID1 configuration, we can achieve better performance. More-

over, we still ensure the data reliability of the whole storage system by duplicating the dirty data on SSD caches.

5. EXPLOITING SUPERPAGES IN A NONVOLATILE MEMORY FILE SYSTEM*

Emerging nonvolatile memory technologies (sometimes referred as Storage Class Memory (SCM)), are poised to close the enormous performance gap between persistent storage and main memory. The SCM devices can be attached directly to memory bus and accessed like normal DRAM. It becomes then possible to exploit memory management hardware resources to improve file system performance. However, in this case, SCM may share critical system resources such as the TLB, page table with DRAM which can potentially impact SCM's performance.

In this chapter, we propose to solve this problem by employing superpages to reduce the pressure on memory management resources such as the TLB. As a result, the file system performance is further improved. We also analyze the space utilization efficiency of superpages. We improve space efficiency of the file system by allocating normal pages (4KB) for small files while allocating super pages (2MB on x86) for large files. We show that it is possible to achieve better performance without loss of space utilization efficiency of nonvolatile memory.

5.1 Background

For decades, modern file systems are designed on the assumption that the underlying storage devices are block-based, such as disk or flash-based SSD. The recent development of nonvolatile memory technologies such as phase change memory (PCM) are poised to revolutionize storage in computer systems. These technologies collectively are termed Storage Class Memory (SCM). The SCM devices are attached

*Reprinted with permission from "Exploiting superpages in a nonvolatile memory file system" by Sheng Qiu and A.L.Narasimha Reddy, 2012. IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST), Copyright 2012 by IEEE

directly to memory bus and are byte-addressable. SCM can offer comparable access latency to DRAM and are orders of magnitude faster than traditional disks. Processor can access persistent data through memory load/store instructions. Figure 5.1 shows the potential system hierarchy while building SCM as the persistent storage. As shown in Figure 5.1, the DRAM and SCM can sit in parallel and be accessed through memory bus. It becomes possible to leverage the memory management module to simplify and accelerate file system operations such as space management and file block addressing. Previous work – SCMFS [85] has exploited memory management hardware to improve file system performance. However, this approach also added more work for memory hardware resources such as TLB and MMU which caused increased data TLB misses. In this chapter, we show that it is possible to obtain better file system performance by reducing pressures on such resources.

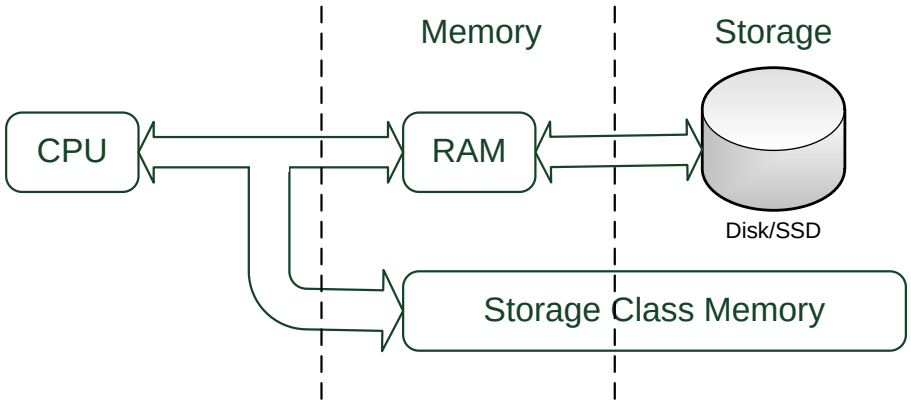


Figure 5.1: Storage class memory

In this chapter, we propose to solve the problem of increased TLB misses by employing superpages. As a result, the performance of our file system is further improved. Compared with normal pages (usually 4K), the super pages (2MB on

x86) are able to enlarge the coverage of TLB because a TLB entry for super pages covers more memory than normal 4KB pages. As a result, we can effectively reduce the TLB misses when the size of TLB is limited and fixed.

We also analyze the space utilization efficiency of superpages. We improve space efficiency of the file system by allocating normal pages for small files and metadata while allocating super pages for large files. We show that it is possible to achieve better performance without loss of space utilization efficiency of nonvolatile memory.

The primary contributions of our design are as following: (a) we analyze the impact of TLB misses while designing a nonvolatile memory file system, (b) we propose to solve this problem by employing superpages for large files while utilizing normal pages for small files and metadata, and (c) we show that it is possible to achieve better performance without loss of space utilization efficiency of nonvolatile memory.

5.2 Design and implementation

To accelerate the memory access speed, modern processors cache the virtual to physical address mappings from the page tables in TLB. Expensive performance penalties are incurred whenever we get TLB misses. To enlarge the coverage of the TLB, most hardware and operating systems support superpages. In this section, we describe how to efficiently employ superpages within our file system.

5.2.1 Preservation for superpage

A superpage is a memory page of larger size than an ordinary page. To allocate a superpage, we are required to have a contiguous memory space which is usually multiple sizes of a normal page. Therefore it is not guaranteed to be able to obtain a superpage successfully even though there is still sufficient physical memory. In our implementation, we solve this problem by preserving a contiguous, configurable size

of SCM for allocating superpages. We divide the physical space of SCM into two regions, one for normal page allocation and the other for superpages. The boundary between normal and super page region is configurable during file system mounting. Figure 5.2 shows the layout of the physical space of SCM.

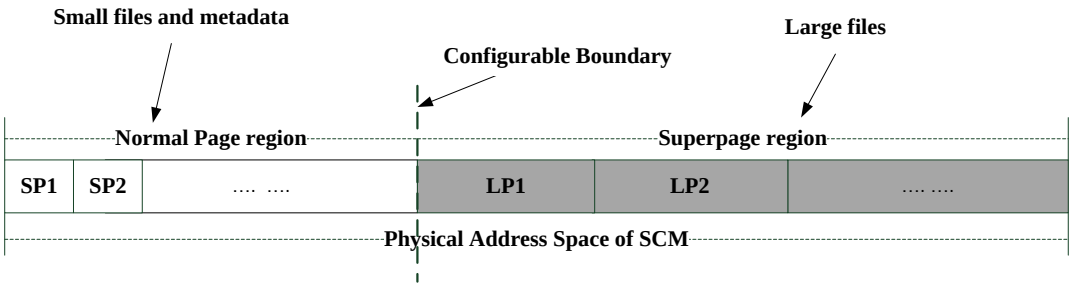


Figure 5.2: Physical space of SCM

5.2.2 Space utilization

In a file system, we may have a number of small and large files as well as metadata that need to be stored. The space utilization efficiency is very important especially for SCM devices considering their expensive cost. We want to utilize superpages for storing data which may potentially reduce the pressure on TLB and improve file system performance. However, allocating the whole file system data with superpages will generate lots of internal fragmentation, especially for small files and metadata. In such a case, we may have a low space utilization efficiency on the SCM device.

To achieve better performance without loss of space utilization efficiency, we propose using both normal and super pages within our file system. As shown in Figure 5.2, small files and metadata are mapped to normal pages while large files are stored within super pages. As a result, we solve the internal fragmentation issue for

small size data while TLB misses are reduced effectively whenever accessing large files on super pages.

One potential problem is that it is not easy to decide how the file size will grow during creation. Therefore, initially, we always allocate normal pages for file data. Whenever the file size become larger than a configurable threshold, we begin to migrate this file to super pages. After migration, the original file data (on normal pages) will be freed and the corresponding inode metadata will be updated. To minimize the impact on SCMFS's performance, we use a background kernel thread to handle the migration. This kernel thread will pick up those files that are not being written currently to do migration. It is noted that read request can still be handled by the original file (on normal pages) during migration, while write request has to wait until the migration process finishes. Since most large files in real system are multi-media or read-heavily files, which usually keep a relatively stable size once written. Therefore the migration between normal and super pages will not be frequent.

5.2.3 Modifications to kernel

To support superpages within our file system, we made several modifications to the original Linux kernel 2.6.33. We first add a memory zone "ZONE_STORAGE" into the kernel. We put all the address range of DRAM space which we used to emulate SCM into the new zone "ZONE_STORAGE". Then we add a set of memory allocation/deallocation functions for super pages. Generally, there are four main functions used by our file system. The function nvmalloc_superpage() allocates designated number of super pages from "ZONE_STORAGE" while nvfree_superpage() is the corresponding function for deallocation. Another two functions are nvmalloc_expand_superpage() and nvmalloc_shrink_superpage(). The former one is used when the file size increases and the mapped super pages are not enough, while the

latter one is used to recycle the allocated but unused super pages.

5.3 Evaluation

In this section, we evaluate the performance of enhanced SCMFS with superpage support. We implemented superpages within SCMFS on a linux kernel of version 2.6.33.

5.3.1 Methodology

To evaluate superpage performance of SCMFS, we use multiple benchmarks. The first benchmark, IOZONE [82] 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 IOZONE to evaluate the performance of accessing file data. The second benchmark we use is postmark [38], which creates a lot of files and performs read/write operations on them. The file size can be configured within one specific range. We use this benchmark to evaluate superpage's impact when accessing both small and large files in SCMFS. In all experiments, we track the number of allocated superpages and the actual file data size. We see that our approach keeps the internal fragmentation within 1% on average. 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 configured 4GB of the memory as the type "ZONE_STORAGE", and used it as Storage Class Memory.

In all the benchmarks, we compare the performance of SCMFS with/without superpage supported to that of other existing file systems, including ramfs, tmpfs and ext2. Since ext2 is designed for a traditional storage device, we run it on ramdisk which emulates a disk drive by using normal RAM in main memory. It is noted that tmpfs, ramfs and ramdisk are not designed for persistent memory, and none of them can be directly used on storage class memory.

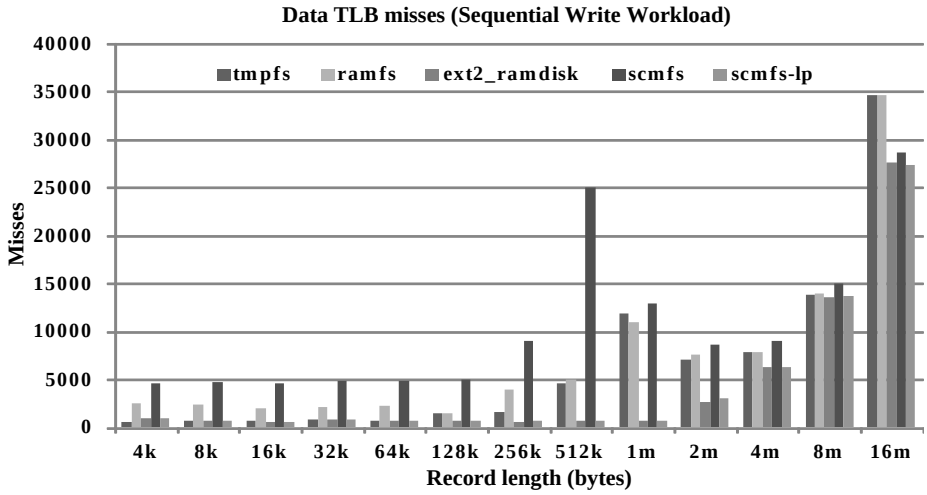


Figure 5.3: TLB misses – iozone sequential write workload

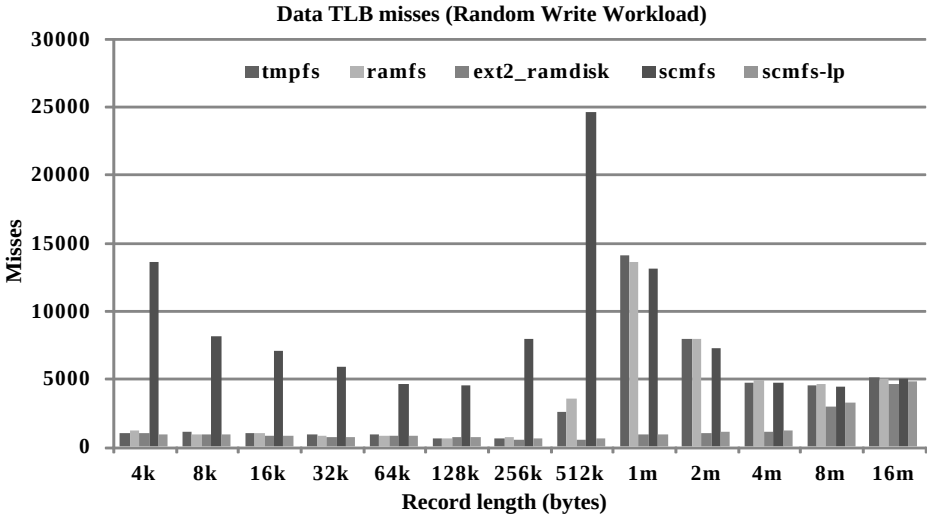


Figure 5.4: TLB misses – iozone random write workload

5.3.2 Iozone results

We run IOZONE with sequential and random workloads for both read and write. To obtain the performance of TLB, we used the performance counters in the modern processors through the PAPI library [53]. Figure 5.3 and 5.4 show the data TLB misses of all file systems while running IOZONE's sequential and random write workloads. We can see SCMFS with superpage support (bar scmfs-lp) effectively reduced the data TLB misses compared with original SCMFS. When the request size become larger (more than 2MB), the variance of data TLB misses tends to be smaller among all file systems. This is because the number of TLB entries for superpages is limited which may not cache all the superpages when request sizes is larger.

Figure 5.5 and 5.6 show the corresponding throughput for IOZONE's sequential and random write workloads. We can see that employing superpages within SCMFS improves throughput performance significantly. It is noted that in Figure 5.5, ext2 on ramdisk performs much better than other file systems when request size is within 128kb–512kb. This is because within that range, ext2 has much lower L2 data cache misses compared to other file systems.

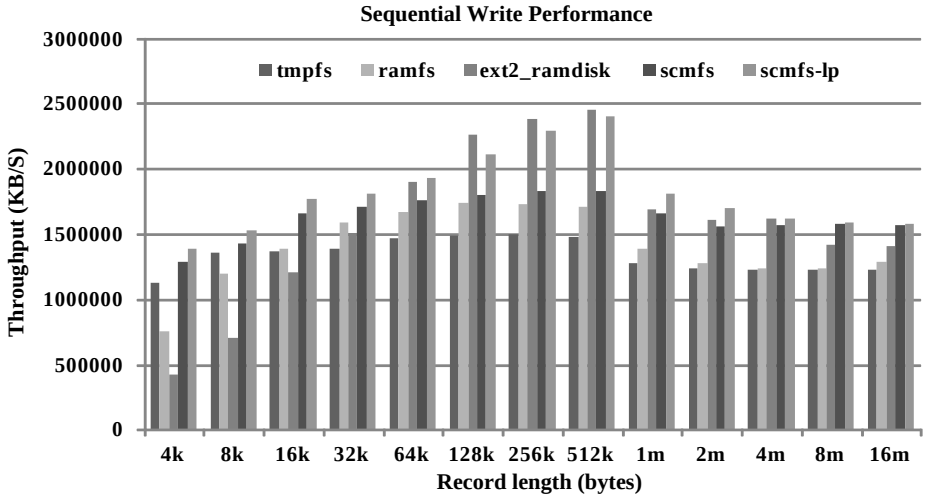


Figure 5.5: Throughput – iozone sequential write workload

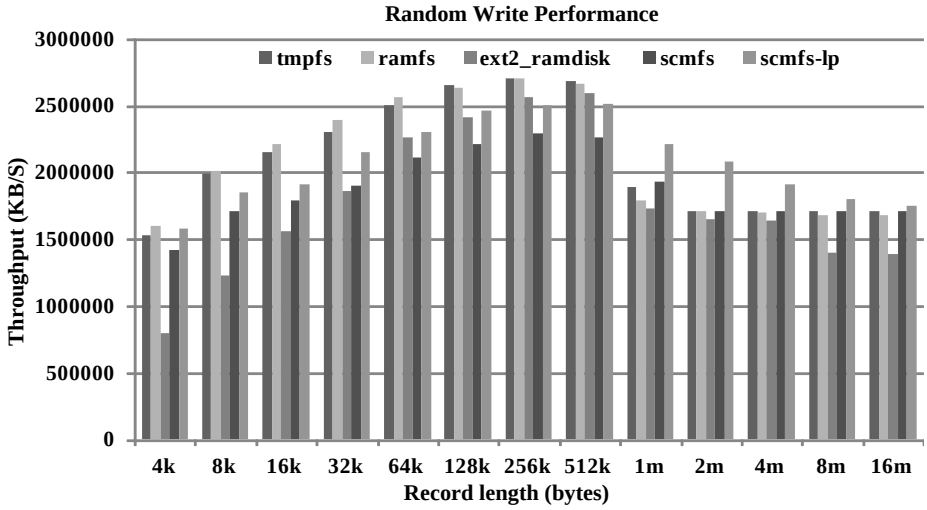


Figure 5.6: Throughput – iozone random write workload

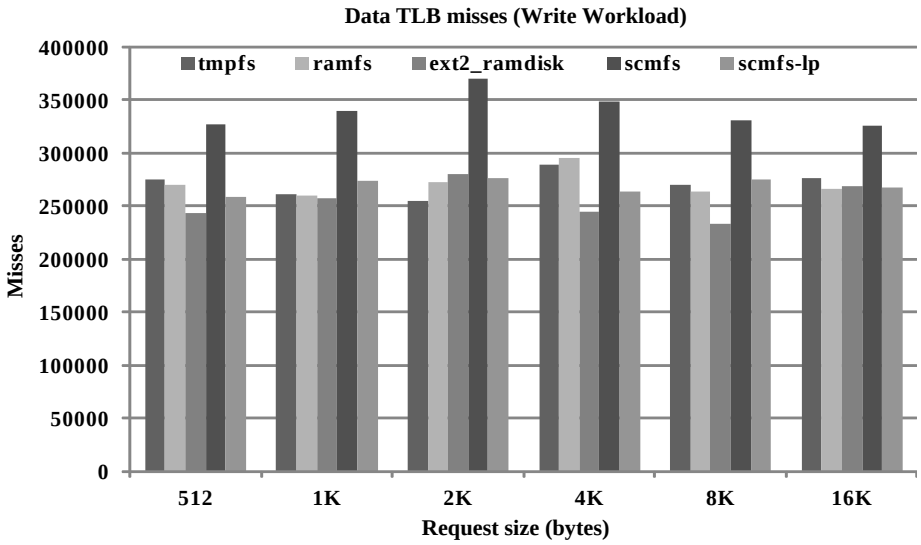


Figure 5.7: TLB misses – postmark's write workload

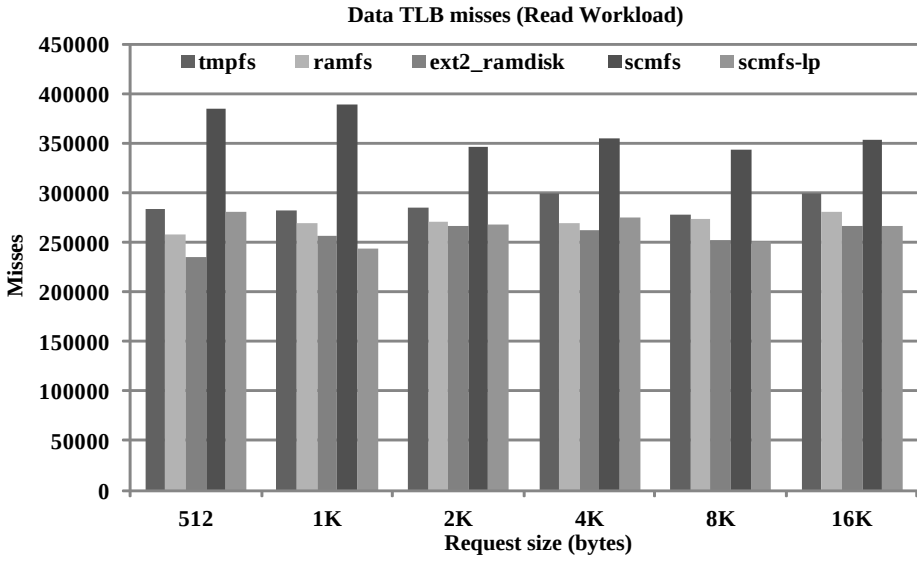


Figure 5.8: TLB misses – postmark's read workload

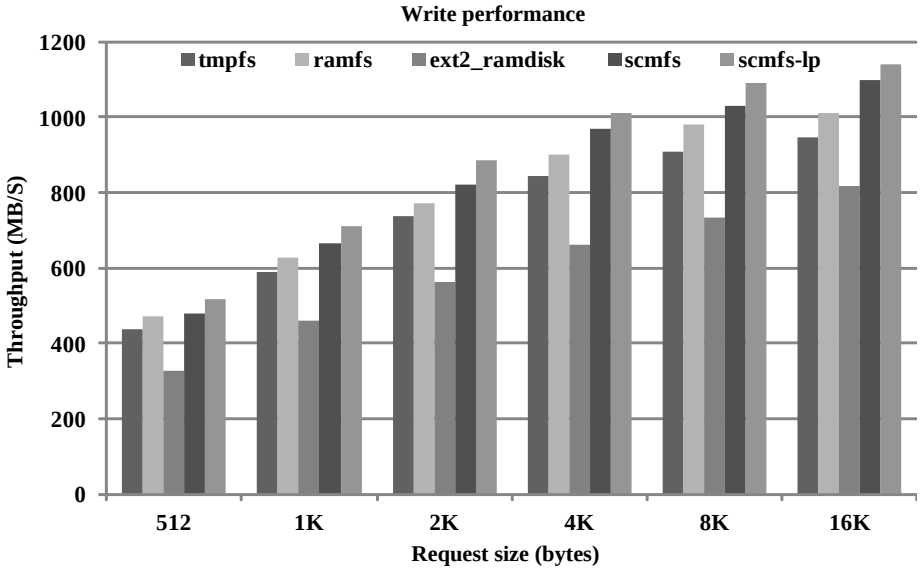


Figure 5.9: Throughput of postmark's write workload

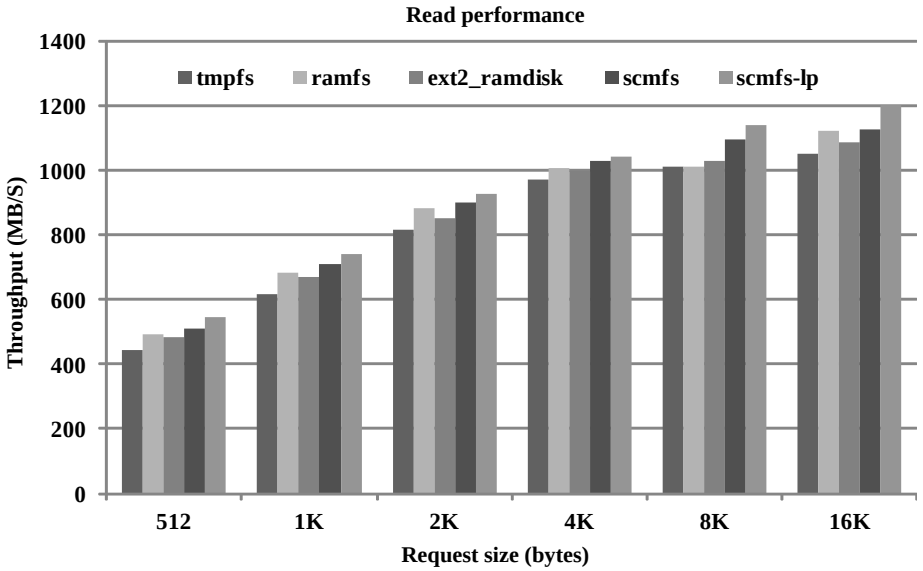


Figure 5.10: Throughput of postmark's read workload

5.3.3 Postmark results

In this section, we evaluate the impact of superpages by running postmark benchmark. We use postmark to generate both read intensive and write intensive workloads. In our experiment, postmark created a number of small and large files and performed read, append and delete transactions. We again used the PAPI library to investigate the detailed performance of TLB.

Figure 5.7 and 5.8 show the data TLB misses of postmark for all file systems. We can see that utilizing superpages effectively reduces data TLB misses of SCMFS which is consistent with IOZONE results. The throughput performance is shown in Figure 5.9 and 5.10. We again achieved better performance while employing superpages within SCMFS.

5.4 Related work

A number of recent works proposed hybrid file systems via byte-addressable NVRAM and HDDs [50, 80]. In [50], Miller et al. proposed using a byte-addressable NVRAM file system which used NVRAM as storage for file system metadata, a write buffer, and storage for the front parts of files. In the Conquest file system [80], the byte-addressable NVRAM layer holds metadata, small files and executable files while the large files reside on HDDs. Hybrid file systems for byte-addressable NVRAM and NAND Flash are proposed to address NAND-Flash file system specific issues using byte-addressable NVRAM [44, 36, 59]. They include mount latency, recovery overhead against unexpected system failure, and the overhead in accessing page metadata for a NAND Flash device. All of these previous works assume the NVRAM is small and stores only metadata and/or small files, while our file system is designed for purely nonvolatile memory based persistent storage which expects the NVRAM to be large enough to hold the whole file system data.

BPFS [15] file system designed for non-volatile byte-addressable memory, 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. DFS[34] is another file system designed for flash storage. 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.

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 [45, 46]. Some of them use a DRAM device as a cache of PCM in the hierarchy. Page placement policies are proposed in [66] for a memory controller within a PCM-DRAM hybrid memory system. Several wear-leveling schemes to protect PCM devices from normal applications and even malicious attacks have been proposed [63, 52, 71, 88]. Since our work focuses on the file system layer, all the hardware techniques can be integrated with our file system to provide better performance or stronger protection.

The importance of TLB performance and support for superpages has been described in [55, 78, 20, 76]. Impact of TLB misses on application performance prompted proposals for effective superpage management [55]. The architectural and operating system support required to exploit medium-sized superpages (e.g., 64KB) is presented in [78]. Our approach focuses on employing superpages within a non-volatile memory file system. We propose to utilize both normal pages and super pages to achieve better performance of file system without loss of space utilization efficiency of the SCM device.

6. CONCLUSION

In this dissertation, we explored several new nonvolatile memory technologies, such as flash-SSD, PCM. We have presented novel approaches for integrating these devices into existing storage hierarchy of a computer system.

In chapter 2, we analyzed the characteristics of PCM and proposed a PCM-based main memory, which can provide us higher density and lower power consumption compared to traditional DRAM-based main memory. To realize these benefits, we have to first solve the expensive write problem of PCM in terms of both performance and lifetime. We proposed a new CPU cache design for the last-level cache, which can potentially reduce the write traffic to the PCM main memory. The results showed that, our design effectively improved the lifetime of PCM as well as the energy efficiency. Moreover, the overhead of our design is negligible and we do not incur additional misses at the last-level CPU cache for most of the workloads and configurations.

In chapter 3, we considered the poor performance of random writes on flash SSDs. We proposed to build a hybrid storage system which includes a small nonvolatile memory and a SSD. We designed a new file system, NVMFS, to manage the hybrid storage space. Our design satisfied most of the random write requests on the fast nonvolatile memory and only performed large, optimized writes on flash SSD. As a result, we reduced the number of small random I/Os to SSD significantly which improved the write bandwidth/throughput of SSD. The experimental results show that we also reduced the garbage collection overhead and the write amplification inside SSDs.

In chapter 4, we considered the problem of managing space on SSDs when they

are employed as caches in front of hard disks. The existing Bcache implementation relies on RAID software to manage the multiple devices if we employ more than one SSD at the cache layer. We proposed a new way to manage the SSDs directly at the cache layer and applied different protection policies for the cached clean and dirty data. The experimental results show that our approach improved the hit ratio of SSD caches as well as the throughput of the storage system.

In chapter 5, we considered the problem of managing space in a storage class memory. The SCM devices can be attached directly to memory bus and accessed like normal DRAM, which provides us the opportunity of exploiting memory management hardware resources to improve file system performance. However, in this case, SCM may share critical system resources such as the TLB, page table with DRAM which can potentially impact SCM's performance. We proposed to employ superpages to reduce the pressure on memory management resources. The experimental results show that our design significantly reduced the data TLB misses and further improved the performance of file system.

REFERENCES

[1] AgigA NVDIMM Technology. AgigA DDR3 NVDIMM. [EB/OL], 2013. `http://www.agigatech.com/ddr3.php`.

[2] Hussein Al-Zoubi, Aleksandar Milenkovic, and Milena Milenkovic. Performance evaluation of cache replacement policies for the spec cpu2000 benchmark suite. In *Proceedings of the 42nd annual Southeast regional conference*, ACM-SE 42, pages 267–272, New York, NY, USA, 2004. ACM.

[3] F. Bedeschi, R. Fackenthal, C. Resta, E.M. Donze, M. Jagasivamani, E.C. Buda, F. Pellizzer, D.W. Chow, A. Cabrini, G. Calvi, R. Faravelli, A. Fantini, G. Torelli, D. Mills, R. Gastaldi, and G. Casagrande. A Bipolar-Selected Phase Change Memory Featuring Multi-Level Cell Storage. *Solid-State Circuits, IEEE Journal of*, 44(1):217–227, 2009.

[4] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, PACT '08, pages 72–81, New York, NY, USA, 2008. ACM.

[5] Nathan L. Binkert, Ronald G. Dreslinski, Lisa R. Hsu, Kevin T. Lim, Ali G. Saidi, and Steven K. Reinhardt. The M5 Simulator: Modeling Networked Systems. *IEEE Micro*, 26:52–60, 2006.

[6] F. Bodin and A. Seznec. Skewed Associativity Enhances Performance Predictability. In *The 22nd Annual International Symposium on Computer Architecture (ISCA)*, pages 265 – 274, 1995.

[7] Luc Bouganim, Björn THór Jónsson, and Philippe Bonnet. uflip: Understanding flash io patterns. In *CIDR*, 2009.

[8] Btree Wiki. Btrfs: A Linux File System. [EB/OL], 2012. `https://btrfs.wiki.kernel.org/`.

[9] Calin Caçaval and David A. Padua. Estimating Cache Misses and Locality Using Stack Distances. In *The 17th Annual International Conference on Supercomputing (ICS)*, pages 150–159, 2003.

[10] Hyunjin Cho Changwoo Min, Kangnyeon Kim and Young Ik Eom Sang-Won Lee. Sfs: Random write considered harmful in solid state drives. In *Proceedings of the 10th USENIX conference on File and storage technologies*, 2012.

[11] Feng Chen, David A. Koufaty, and Xiaodong Zhang. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. In *Proceedings of the eleventh international joint conference on Measurement and modeling of computer systems*, SIGMETRICS '09, pages 181–192, 2009.

[12] Feng Chen, David A. Koufaty, and Xiaodong Zhang. Hystor: making the best use of solid state drives in high performance storage systems. In *Proceedings of the international conference on Supercomputing*, pages 22–32, 2011.

[13] Feng Chen, Tian Luo, and Xiaodong Zhang. Caftl: a content-aware flash translation layer enhancing the lifespan of flash memory based solid state drives. In *Proceedings of the 9th USENIX conference on File and stroage technologies*, FAST'11, pages 6–6, 2011.

[14] Peter M. Chen, Wee Teck Ng, Subhachandra Chandra, Christopher Aycock, Gurushankar Rajamani, and David Lowell. The rio file cache: surviving operating

system crashes. In *Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*, ASPLOS-VII, pages 74–83, 1996.

[15] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. Better i/o through byte-addressable, persistent memory. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 133–146, 2009.

[16] Data Sheet of IBM 3340 Disk Storage Unit. IBM 3340 Disk Storage Unit. [EB/OL], 2012. `http://www-03.ibm.com/ibm/history/exhibits/storage/storage3340.html`.

[17] Data Sheet of Sata Hard Disk Drives. Wd Velociraptor: Sata Hard Drives. [EB/OL], 2012. `http://www.wdc.com/wdproducts/library/SpecSheet/ENG/2879-701284.pdf`.

[18] Gaurav Dhiman, Raid Ayoub, and Tajana Rosing. Pdram: a hybrid pram and dram main memory system. In *Proceedings of the 46th Annual Design Automation Conference*, DAC '09, pages 664–469, New York, NY, USA, 2009. ACM.

[19] Filebench Wiki. Filebench: A File System and Storage Benchmark that Allows to Generate a Large Variety of Workloads. [EB/OL], 2013. `http://sourceforge.net/apps/mediawiki/filebench/index.php`.

[20] N. Ganapathy and C. Schimmel. General purpose operating system support for multiple page sizes. In *Proc. of the USENIX Annual Technical Conference*, pages 91–104, 1998.

[21] Mark Gebhart, Joel Hestness, Ehsan Fatehi, Paul Gratz, and Stephen W. Keckler. Running PARSEC 2.1 on M5. Technical report, The University of Texas at Austin, Department of Computer Science, 2009.

[22] J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation Techniques for Storage Hierarchies. *IBM Systems Journal*, 9(2):78–117, 1970.

[23] James R. Goodman. Using Cache Memory to Reduce Processor-Memory Traffic. In *The 10th Annual International Symposium on Computer Architecture (ISCA)*, pages 124–131, 1983.

[24] Jorge Guerra, Himabindu Pucha, Joseph Glider, Wendy Belluomini, and Raju Rangaswami. Cost effective storage using extent based dynamic tiering. In *Proceedings of the 9th USENIX conference on File and stroage technologies*, FAST'11, pages 20–20, 2011.

[25] Aayush Gupta, Youngjae Kim, and Bhuvan Urgaonkar. Dftl: a flash translation layer employing demand-based selective caching of page-level address mappings. In *Proceedings of the 14th international conference on Architectural support for programming languages and operating systems*, ASPLOS '09, pages 229–240, 2009.

[26] Aayush Gupta, Raghav Pisolkar, Bhuvan Urgaonkar, and Anand Sivasubramaniam. Leveraging value locality in optimizing nand flash-based ssds. In *Proceedings of the 9th USENIX conference on File and stroage technologies*, FAST'11, pages 7–7, 2011.

[27] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.

[28] M.D. Hill and A.J. Smith. Evaluating Associativity in CPU Caches. *IEEE Transactions on Computers*, 38(12):1612 –1630, 1989.

[29] Xiao-Yu Hu, Evangelos Eleftheriou, Robert Haas, Ilias Iliadis, and Roman Pletka. Write amplification analysis in flash-based solid state drives. In *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, SYSTOR '09, pages 10:1–10:9, 2009.

[30] International Technology Roadmap for Semiconductors (ITRS) Working Group. International Technology Roadmap for Semiconductors (ITRS). [EB/OL], 2009. `http://www.itrs.net/Links/2009ITRS/Home2009.htm`.

[31] Jens Axboe. A Block Layer IO Tracing Mechanism. [EB/OL], 2012. `http://linux.die.net/man/8/blktrace`.

[32] Jens Axboe. Flexible IO Workload. [EB/OL], 2012. `http://freecode.com/projects/fio`.

[33] Heeseung Jo, Jeong-Uk Kang, Seon-Yeong Park, Jin-Soo Kim, and Joonwon Lee. FAB: Flash-aware Buffer Management Policy for Portable Media Players. *Consumer Electronics, IEEE Transactions on*, 52(2):485 – 493, 2006.

[34] William K. Josephson, Lars Ailo Bongo, David Flynn, and Kai Li. Dfs: A file system for virtualized flash storage. In *Proc. of the USENIX Conference on File and Storage Technologies*, volume 6, pages 85–100, 2010.

[35] Norman P. Jouppi. Cache Write Policies and Performance. In *The 20th Annual International Symposium on Computer Architecture (ISCA)*, pages 191–201, 1993.

[36] Jaemin Jung, Youjip Won, Eunki Kim, Hyungjong Shin, and Byeonggil Jeon. FRASH: Exploiting Storage Class Memory in Hybrid File System for Hierarchi-

cal Storage. *Trans. Storage*, 6(1):3:1–3:25, 2010.

[37] D. Kaseridis, J. Stuecheli, Jian Chen, and L.K. John. A Bandwidth-Aware Memory-Subsystem Resource Management using Non-invasive Resource Profilers for Large CMP Systems. In *The 16th IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 1 –11, 9-14 2010.

[38] Jeffrey Katcher. PostMark: A New File System Benchmark. Technical Report TR3022. Network Applicance Inc. October 1997.

[39] Atsuo Kawaguchi, Shingo Nishioka, and Hiroshi Motoda. A flash-memory based file system. In *Proceedings of the USENIX 1995 Technical Conference Proceedings*, TCON'95, pages 13–13, 1995.

[40] Kent Overstreet. Bcache: A Linux Kernel Block Layer Cache. [EB/OL], 2009. `http://bcache.evilpiepirate.org/`.

[41] Taeho Kgil and Trevor Mudge. Flashcache: a nand flash memory file cache for low power web servers. In *Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, CASES '06, pages 103–112, New York, NY, USA, 2006. ACM.

[42] Hyojun Kim and Seongjun Ahn. Bplru: a buffer management scheme for improving random writes in flash storage. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, FAST'08, pages 16:1–16:14, Berkeley, CA, USA, 2008. USENIX Association.

[43] Jesung Kim, Jong Min Kim, S.H. Noh, Sang Lyul Min, and Yookun Cho. A Space-efficient Flash Translation Layer for CompactFlash Systems . *Consumer Electronics, IEEE Transactions on*, 48(2):366 –375, 2002.

[44] Jin Kyu Kim, Hyung Gyu Lee, Shinho Choi, and Kyoung Il Bahng. A pram and nand flash hybrid architecture for high-performance embedded storage subsystems. In *Proceedings of the 8th ACM international conference on Embedded software*, EMSOFT '08, pages 31–40, 2008.

[45] B.C. Lee, Ping Zhou, Jun Yang, Youtao Zhang, Bo Zhao, E. Ipek, O. Mutlu, and D. Burger. Phase-Change Technology and the Future of Main Memory. *Micro, IEEE*, 30(1):143–143, 2010.

[46] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. Architecting Phase Change Memory as a Scalable Dram Alternative. In *The 36th Annual International Symposium on Computer Architecture (ISCA)*, pages 2–13, 2009.

[47] Donghee Lee, Jongmoo Choi, Jong-Hun Kim, Sam H. Noh, Sang Lyul Min, Yookun Cho, and Chong Sang Kim. On the Existence of a Spectrum of Policies that Subsumes the Least Recently Used (LRU) and Least Frequently Used (LFU) Policies. In *The ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, pages 134–143, 1999.

[48] Sang-Won Lee, Dong-Joo Park, Tae-Sun Chung, Dong-Ho Lee, Sangwon Park, and Ha-Joo Song. A Log Buffer-based Flash Translation Layer Using Fully-associative Sector Translation. *ACM Trans. Embed. Comput. Syst.*, 6(3), 2007.

[49] Sungjin Lee, Dongkun Shin, Young-Jin Kim, and Jihong Kim. LAST: Locality-aware Sector Translation for NAND Flash Memory-based Storage Systems. *SIGOPS Oper. Syst. Rev.*, 42(6):36–42, October 2008.

[50] E.L. Miller, S.A. Brandt, and D.D. Long. Hermes: High-performance reliable mramenabled storage. In *Proc. of the 8th IEEE Workshop on Hot Topics in Operating Systems (HotOS)*, pages 83–87, 2011.

[51] Daniel Molka, Daniel Hackenberg, Robert Schone, and Matthias S. Muller. Memory Performance and Cache Coherency Effects on an Intel Nehalem Multiprocessor System. In *The 2009 18th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 261–270, 2009.

[52] M.Qureshi, Andre Seznec, Luis Lastras, and Michele Franceschini. Practical and secure pcm systems by online detection of malicious write streams. In *The IEEE 17th International Symposium on High Performance Computer Architecture (HPCA)*, pages 478–489, 2011.

[53] Philip J. Mucci, Shirley Browne, Christine Deane, and George Ho. Papi: A portable interface to hardware performance counters. In *Proc. of the Department of Defense HPCMP Users Group Conference*, pages 7–10, 1999.

[54] Dushyanth Narayanan and Orion Hodson. Whole-system persistence. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 401–410, 2012.

[55] Juan Navarro, Sitaram Iyer, Peter Druschel, and Alan Cox. Practical, transparent operating system support for superpages. In *Proc. of the 5th USENIX OSDI*, 2002.

[56] Nilfs2 Wiki. NILFS2: New Implementation of a Log-structured File System version 2. [EB/OL], 2012. `http://en.wikipedia.org/wiki/NILFS`.

[57] Numonyx. The Basics of Phase Change Memory (PCM) Technology: A New Class of Non-volatile Memory. [EB/OL], 2008. `http://www.numonyx.com/Documents/WhitePapers/PCM_Basics_WP.pdf`.

[58] Seon-yeong Park, Dawoon Jung, Jeong-uk Kang, Jin-soo Kim, and Joonwon Lee. CFLRU: a Replacement Algorithm for Flash Memory. In *The 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, pages 234–241, 2006.

[59] Youngwoo Park, Seung-Ho Lim, Chul Lee, and Kyu Ho Park. Pffs: a scalable flash memory file system for the hybrid architecture of phase-change ram and nand flash. In *Proceedings of the 2008 ACM symposium on Applied computing*, SAC '08, pages 1498–1503, 2008.

[60] M.D. Powell, A. Agarwal, T.N. Vijaykumar, B. Falsafi, and K. Roy. Reducing Set-Associative Cache Energy via Way-prediction and Selective Direct-mapping. In *In the 34th ACM/IEEE International Symposium on Microarchitecture (Micro)*, pages 54 – 65, 2001.

[61] M.K. Qureshi, M.M. Franceschini, and L.A. Lastras-Montano. Improving Read Performance of Phase Change Memories Via Write Cancellation and Write Pausing. In *The IEEE 16th International Symposium on High Performance Computer Architecture (HPCA)*, pages 1 –11, 2010.

[62] M.K. Qureshi, D. Thompson, and Y.N. Patt. The v-way cache: demand-based associativity via global replacement. In *Computer Architecture, 2005. ISCA '05. Proceedings. 32nd International Symposium on*, pages 544–555, 2005.

[63] Moinuddin K. Qureshi, John Karidis, Michele Franceschini, Vijayalakshmi Srinivasan, Luis Lastras, and Bulent Abali. Enhancing lifetime and security of pcm-based main memory with start-gap wear leveling. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 14–23, New York, NY, USA, 2009. ACM.

[64] Moinuddin K. Qureshi, Daniel N. Lynch, Onur Mutlu, and Yale N. Patt. A Case for MLP-Aware Cache Replacement. In *The 33rd Annual International Symposium on Computer Architecture (ISCA)*, pages 167–178, 2006.

[65] Moinuddin K. Qureshi, Vijayalakshmi Srinivasan, and Jude A. Rivers. Scalable High Performance Main Memory System using Phase-Change Memory Technology. In *The 36th Annual International Symposium on Computer Architecture(ISCA)*, pages 24–33, 2009.

[66] Luiz E. Ramos, Eugene Gorbatov, and Ricardo Bianchini. Page placement in hybrid memory systems. In *Proceedings of the international conference on Supercomputing*, ICS '11, pages 85–95, New York, NY, USA, 2011. ACM.

[67] S. Raoux, G.W. Burr, M.J. Breitwisch, C.T. Rettner, Y.C. Chen, R.M. Shelby, M. Salinga, D. Krebs, S.-H. Chen, H. L Lung, and C.H. Lam. Phase-Change Random Access Memory: A Scalable Technology. *IBM Journal of Research and Development*, 52(4.5):465–479, 2008.

[68] D. Rolan, B.B. Fraguela, and R. Doallo. Adaptive line placement with the Set Balancing Cache. In *The 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 529 –540, 2009.

[69] Mendel Rosenblum and John K. Ousterhout. The Design and Implementation of a Log-structured File System. *ACM Trans. Comput. Syst.*, 10(1):26–52, February 1992.

[70] Mohit Saxena, Michael M. Swift, and Yiying Zhang. Flashtier: a lightweight, consistent and durable storage cache. In *Proceedings of the 7th ACM european conference on Computer Systems*, EuroSys '12, pages 267–280, New York, NY, USA, 2012. ACM.

[71] Nak Hee Seong, Dong Hyuk Woo, and Hsien-Hsin S. Lee. Security refresh: prevent malicious wear-out and increase durability for phase-change memory with dynamically randomized address mapping. In *Proceedings of the 37th annual international symposium on Computer architecture*, ISCA '10, pages 383–394, New York, NY, USA, 2010. ACM.

[72] A. Seznec. A Case for Two-way Skewed-Associative Caches. In *The 20th Annual International Symposium on Computer Architecture (ISCA)*, pages 169–178, 1993.

[73] Gokul Soundararajan, Vijayan Prabhakaran, Mahesh Balakrishnan, and Ted Wobber. Extending ssd lifetimes with disk-based write caches. In *Proceedings of the 8th USENIX conference on File and storage technologies*, pages 8–8, 2010.

[74] Standards Performance Evaluation Corporation (SPEC). SPEC CPU2006 Benchmark Suite. [EB/OL], 2006. `http://www.spec.org/cpu2006/`.

[75] Jeffrey Stuecheli, Dimitris Kaseridis, David Daly, Hillery C. Hunter, and Lizy K. John. The Virtual Write Queue: Coordinating DRAM and Last-Level Cache Policies. In *The 37th Annual International Symposium on Computer Architecture (ISCA)*, 2010.

[76] I. Subramanian, C. Mather, K. Peterson, and B. Raghunath. Implementation of multiple page size support in hp-ux. In *Proc. of the USENIX Annual Technical Conference*, pages 105–118, 1998.

[77] Guangyu Sun, Yongsoo Joo, Yibo Chen, Dimin Niu, Yuan Xie, Yiran Chen, and Hai Li. A Hybrid Solid-State Storage Architecture for the Performance, Energy Consumption, and Lifetime Improvement. In *The 16th IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 1–12, 2010.

[78] Madhusudhan. Talluri and Mark D. Hill. Surpassing the tlb performance of superpages with less operating system support. In *Proc. of the 6th ASPLOS*, 1994.

[79] Viking Nonvolatile DIMM technology. Viking Techonology of Non-Volatile DIMM. [EB/OL], 2013. `http://www.vikingtechnology.com/non-volatile-dimm`.

[80] An-I Andy Wang, Geoff Kuenning, Peter Reiher, and Gerald Popek. The Conquest File System: Better Performance Through a Disk/Persistent-RAM Hybrid Design. *Trans. Storage*, 2(3):309–348, August 2006.

[81] Wikipedia. Zipf's Law. [EB/OL], 2009. `https://en.wikipedia.org/wiki/Zipf's_law`.

[82] William D. Norcott. IOzone Filesystem Benchmark. [EB/OL], 2012. `http://www.iozone.org/`.

[83] Guanying Wu, Xubin He, and Ben Eckart. An Adaptive Write Buffer Management Scheme for Flash-based SSDs. *Trans. Storage*, 8(1):1:1–1:24, February 2012.

[84] Michael Wu and Willy Zwaenepoel. envy: a non-volatile, main memory storage system. In *Proceedings of the sixth international conference on Architectural support for programming languages and operating systems*, ASPLOS-VI, pages 86–97, 1994.

[85] Xiaojian Wu and A. L. Narasimha Reddy. Scmfs: a file system for storage class memory. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 39:1–39:11, 2011.

[86] Xiaoxia Wu, Jian Li, Lixin Zhang, Evan Speight, Ram Rajamony, and Yuan Xie. Hybrid Cache Architecture with Disparate Memory Technologies. In *The 36th Annual International Symposium on Computer Architecture (ISCA)*, pages 34–45, 2009.

[87] Jingpei Yang, Ned Plasson, Greg Gillis, and Nisha Talagala. Hec: improving endurance of high performance flash-based cache devices. In *Proceedings of the 6th International Systems and Storage Conference*, SYSTOR '13, pages 10:1–10:11, New York, NY, USA, 2013. ACM.

[88] Ping Zhou, Bo Zhao, Jun Yang, and Youtao Zhang. A durable and energy efficient main memory using phase change memory technology. In *Proceedings of the 36th annual international symposium on Computer architecture*, ISCA '09, pages 14–23, New York, NY, USA, 2009. ACM.