I-CASH: Intelligently Coupled Array of SSD and HDD

Jin Ren and Qing Yang
Dept. of Electrical, Computer, and Biomedical Engineering
University of Rhode Island, Kingston, RI 02881
(rjin,qyang)@ele.uri.edu

Abstract

*This paper presents a new disk I/O architecture composed of an array of a flash memory SSD (solid state disk) and a hard disk drive (HDD) that are intelligently coupled by a special algorithm. We call this architecture **I-CASH**: **I**ntelligently **C**oupled **A**rray of **S**SD and **H**DD. The SSD stores seldom-changed and mostly read **reference data blocks** whereas the HDD stores a log of **deltas** between currently accessed I/O blocks and their corresponding reference blocks in the SSD so that random writes are not performed in SSD during online I/O operations. High speed delta compression and similarity detection algorithms are developed to control the pair of SSD and HDD. The idea is to exploit the fast read performance of SSDs and the high speed computation of modern multi-core CPUs to replace and substitute, to a great extent, the mechanical operations of HDDs. At the same time, we avoid runtime SSD writes that are slow and wearing. An experimental prototype I-CASH has been implemented and is used to evaluate I-CASH performance as compared to existing SSD/HDD I/O architectures. Numerical results on standard benchmarks show that I-CASH reduces the average I/O response time by an order of magnitude compared to existing disk I/O architectures such as RAID and SSD/HDD storage hierarchy, and provides up to 2.8 speedup over state-of-the-art pure SSD storage. Furthermore, I-CASH reduces random writes to SSD implying reduced wearing and prolonged life time of the SSD.*

1. Introduction

While storage capacity and CPU processing power have experienced rapid growth in the past, improvement in data bandwidth and access times of disk I/O systems have not kept pace. As a result, there is an ever widening speed gap between CPUs and disk I/O systems. Disk arrays can improve overall I/O throughput [39] but random access latency is still very high because of mechanical operations involved. Large buffers and deep cache hierarchy can improve latency but the access time reduction they provide has been very limited because of poor data locality at the disk I/O level [22, 52, 54].

This paper presents a new disk I/O architecture that exploits the advancement of flash memory SSD (solid state disks) and multi-core processors. The new disk I/O architecture is referred to as ***I-CASH: I**ntelligently **C**oupled **A**rray of **S**SD and **H**DD*. The main idea of our I-CASH architecture is very simple. Each storage element in the I-CASH consists of an SSD and an HDD that are coupled by an intelligent algorithm. The SSD stores seldom changed and mostly read data called *reference* blocks and the HDD stores a log of *deltas* (or patches) of data blocks of active I/Os with respect to reference data blocks stored in the SSD. The intelligent algorithm performs similarity detection, delta derivations upon I/O writes, combining delta with reference blocks upon I/O reads, and other necessary functions for interfacing the storage to the host OS. I-CASH tries to take full advantages of three different technologies: 1) fast read performance of SSD, 2) high computing power of multi-core processor, and 3) reliable/durable/sequential write performance of HDD. Because of strong regularity and content locality that exist in data blocks [19, 29, 50] , a hard disk block can contain a log of potentially large number of small deltas with respect to reference blocks. As a result, one HDD operation accomplishes multiple I/Os and hence I-CASH improves disk I/O performance greatly by trading high speed computation of multi core CPUs for low access latency of HDD. In addition, random writes in flash SSD are minimized giving rise to longer life time of SSD.

A preliminary prototype I-CASH has been built on the Linux operating system. Standard benchmarks have been run on the prototype to measure the disk I/O performance of I-CASH as compared to existing disk I/O architectures such as RAID0, Fusion-io [16], and LRU SSD cache on top of a disk. Numerical results show that I-CASH reduces the number of HDD operations drastically. The overall I/O speedups over RAID0 range from 1.2 to 7.5. Running our prototype on top of the state-of-the-art SSD storage, I-CASH provides up to 2.8 speedup for certain workloads while using one-tenth of SSD space.

The rest of the paper is organized as follows. Next section gives the background and related work. Section 3 describes the I-CASH architecture and design issues. We discuss our prototype implementation and evaluation methodology in Section 4 followed by numerical results and performance evaluations in Section 5. Section 6 concludes the paper.

2. Background and Motivations

2.1 Flash Memory SSD Technology

Recent developments of flash memory SSDs have been very promising with rapid increase in capacity and

decrease in cost [24, 40]. Because SSD is on a semi-conductor chip, it provides great advantages in terms of high speed random reads, low power consumption, compact size, and shock resistance. Researchers in both academia and industry have been very enthusiastic in adopting this technology [1, 5, 14, 24, 30, 51]. However, most existing research in SSD focused on either using SSD in the similar way to hard disk drives (HDD) with various management algorithms at files system level [23] and device level [5] , or using SSD as an additional cache in the storage hierarchy [24, 30, 36, 40]. We believe that both approaches have limitations because of the physical properties of SSDs.

To understand why simple adoption of SSD has limitations, let us take a close look at the physical properties of a NAND-gate flash memory that is widely used in SSDs. A typical NAND-gate array flash memory chip consists of a number of blocks each of which contains a number of pages. Block with size ranging from tens of KB to MB is the smallest erasable units whereas pages with size ranging from 512B to 16KB are the smallest programmable units. When a write operation is performed, it needs to first find a free page to write. If there is no free page available, an erase operation is necessary to make free pages. Read and write operations usually take tens of microseconds, whereas an erase operation takes 1.5 to 3 milliseconds. Besides performance consideration, the lifetime of the flash memory is limited by the number of erase operations performed on a block. Typically, a block can be erased for only 10k times (for multi level cell: MLC) or 100k times (for single level cell: SLC). After that, the block becomes bad [25]. To make the lifetime of a flash memory longer, wear leveling is typically done by distributing erase operations evenly across all blocks. Recent studies have shown that manufacturers' specifications about SSD lifetime are conservative [6, 18, 34], but reliablity is still one of the major concerns when using SSD as a high data throughput storage device.

To tackle the write issues in SSD, researchers have recently proposed techniques such as hiding erasure latencies and wear leveling at file system level [23], using a log disk as a disk cache to cache data blocks to be written [46], and leveraging phase change random access memory (PRAM) to implement log region [47]. However, none of the existing works on SSD has made attempt to exploit the content locality that exists in disk I/O accesses as discussed next.

2.2 Content Locality

Researchers in computer systems have long observed the strong regularity and content locality that exist in memory pages. Memory pages contain data structures, numbers, pointers, and programs that process data in a predefined way. Such strong regularity and content-locality have been successfully exploited for in-memory data compression [15, 48]. Large files and collections of files also show strong content locality with large amount of data redundancy that can be eliminated by efficient compression algorithms [29, 41]. Data deduplication reduces storage space and disk accesses by keeping a single copy for identical blocks [11, 13, 26, 28, 49, 55]. Researchers have recently proposed efficient methods of identifying duplicate data during backup process. One example is the technique based on RAM prefetching and bloom-filter [55] with close to 99% hit ratio for index lookups. Another example is ChunkStash [13] that uses flash memory to store chunk metadata to further improve backup throughput. Content locality also been exploited in processor designs for instruction reuse and value prediction [9, 17, 21, 32, 44-45].

Besides duplications or identical blocks that exist in data storage, there are many data blocks that are very similar among each other. Delta encoding has been successfully used to eliminate redundancy of one object relative to another [2, 8] , suggesting that many data blocks can be represented as small patches/deltas with respect to reference blocks. Furthermore, recent research literature has reported strong content locality in many data intensive applications with only 5% to 20% of bits inside a data block being actually changed on a typical block write operation [35, 53]. Effectively exploring such content locality of both identical and similar blocks can maximize disk I/O performance.

In addition to the strong regularity and content locality inherent in block data, virtual machines provide us with additional opportunities for content locality. The emerging cloud computing requires hundreds, even thousands, of virtual machines running on servers and clients [31, 42]. Such widespread use of virtual machines creates a problem of *virtual machine image sprawl* [42] where each virtual machine needs to store the entire stack of software and data as a disk image. These disk images contain a large amount of redundant data as observed previously by researchers [37-38, 43]. Gupta et al have recently presented a powerful Difference Engine [19] that has successfully exploited such content locality to perform memory page compression with substantial performance gains. This strong content locality suggests again the possibility of organizing data differently in data storage to obtain optimal performance.

3. I-CASH Architecture

Motivated by the developments of SSDs and multi-core CPUs, coupled with regularity and content locality of disk I/Os, we come up with the I-CASH architecture constructed using a pair of SSD and HDD intelligently coupled by a special algorithm as shown in Figure 1. The idea is turning the traditional thinking by 90°. Instead of having a vertical storage hierarchy with an SSD on top of an HDD, I-CASH arranges SSD and HDD horizontally to store different types of data blocks. The SSD stores

mostly read data called *reference blocks* and the HDD stores a log of deltas called *delta blocks*. A delta in a delta block is derived at run time representing the difference between the data block of an active disk I/O operation and its corresponding reference block stored in the SSD. Upon an I/O write, I-CASH identifies its corresponding reference block in the SSD and computes the delta with respect to the reference block as shown in Figure 1b. Upon an I/O read, the data block is returned by combining the delta with its corresponding reference block as shown in Figure 1c. Since deltas are small due to data blocks' regularity and content locality, we store them in a compact form so that one HDD operation yields many I/Os. The goal here is to convert the majority of I/Os from the traditional seek-rotation-transfer I/O operations on HDD to I/O operations involving mainly SSD reads and computations. The former takes tens of milliseconds whereas the latter takes tens of microseconds. As a result, the SSD in I-CASH is not another level of storage cache but an integral part of the I-CASH architecture. Because of 1) high speed read performance of reference blocks stored in SSDs, 2) potentially large number of small deltas packed in one delta block stored in HDD and cached in the RAM, and 3) high performance CPU coupling the two, I-CASH is expected to improve disk I/O performance greatly. In the following subsections, we will discuss the key design issues of I-CASH.

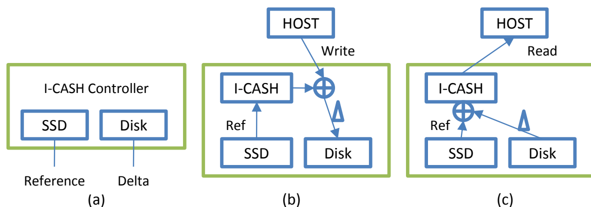

Figure 1. Block diagram of the I-CASH architecture.

3.1 Delta Packing and Unpacking

One critical issue to the success of the I-CASH architecture is whether or not we are able to pack and unpack a batch of deltas in a short time frame so that one HDD operation generates many deltas that can be combined with reference blocks in SSD to satisfy the host's I/O requests. Let LBA_i, LBA_{i+1} … LBA_j, $(j>i)$ be a set of addresses of a sequence of write I/Os from the host in a predefined window. Suppose we derived deltas of these I/Os with respect to their corresponding reference blocks in SSD and packed them in a delta block stored in HDD. The question is: when an I/O request with one of the addresses in the above window, LBA_k $(i <= k <= j)$, appears in subsequent IOs, can we find a set of I/O requests immediately following LBA_k with address LBA_h $(i <= h <= j)$? If we can, how many such I/Os can we find and what is the time frame length containing these I/Os? The number of LBA_h's appeared in the time frame implies potential number of I/Os served by one HDD access. For a given number of such LBA_h's, the length of the time frame containing them determines how long these data blocks need to stay in the DRAM buffer of the I-CASH controller. Therefore, these parameters are very important in our design of the I-CASH architecture. The following examples show how such I/O patterns exist in real applications.

The first case is that all I/O operations that can take advantage of parallel disk arrays can take advantages of I-CASH. RAID was designed to boost I/O performance through parallelism in addition to fault tolerance. To achieve high throughput in RAID system, disk IOs form data stripes across parallel disks with each disk storing one chunk of data in a stripe. With I-CASH, subsequent changes to these data chunks in a stripe can be compressed using the original data of the stripe as reference blocks stored in SSD. The deltas representing such changes on the stripe can be packed together in one delta block. For example, I-CASH can pack deltas of all sequential I/Os into one delta block. Upon read operations of these sequential data blocks, one HDD operation serves all the I/O requests in the sequence. After the HDD operation that is the most time consuming part (in the order of milliseconds), what is left is only operations on semiconductors. The high speed decompression algorithm takes only a few to tens of microseconds to combine the deltas with their corresponding reference blocks that are read from the SSD to satisfy these I/Os.

The second case is the widespread use of virtual machines. As virtual machines are being created, disk images for the virtual machines are made to store software stack and data. The difference between data blocks of a virtual machine image and the data blocks of the native machine are very small and therefore it makes sense to store only the difference/delta between the two instead of storing the entire image. The pairing between a delta and its reference block is clear and should be the data block of the native machine and its exact image of the virtual machine. At the time when virtual machines are created, I-CASH compares each data block of a virtual machine image with the corresponding block of the native machine, derives deltas representing the differences of the image blocks from the native machine blocks, and packs the deltas into delta blocks to be stored in HDD. Future I/Os are served by combining deltas with their corresponding reference blocks in SSD, which mainly involves SSD reads and computations with minimal HDD operations.

The third case is the temporal locality and partial determinism behavior of general non sequential IOs observed by prior researchers [4]. Prior experiments have shown that strong temporal locality exists in disk I/Os and besides sequential accesses to a portion of files, fragments of block access sequence repeat frequently. In

many applications such as office, developer workstations, version control servers, and web servers, there are a large number of read IOs that occur repeatedly and only 4.5-22.3% of the file system data were accessed over a week. Such repetitive and determinism behavior can be exploited to take full advantages of I-CASH architecture.

3.2 Possible Implementations of I-CASH

I-CASH can be implemented in several different ways. The first and the most efficient implementation is to embed the I-CASH architecture inside the controller board of an HDD or an HBA card (host bus adaptor). The controller board will have added NAND-gate flash SSD, an embedded processor, and a small DRAM buffer in addition to the existing disk control hardware and interface. Figure 2(a) shows the block diagram for the implementation of the I-CASH inside the controller. Host system is connected to the controller using a standard interface such as PCIe, SCSI, SATA, SAS, PATA, iSCSI, FC, and etc. An SSD in form of flash memory chip or SSD drive is used to store reference blocks. The embedded processing element performs the I-CASH logic such as delta derivation, similarity detection, combining delta with reference blocks, managing reference blocks, managing metadata, etc. The RAM cache stores temporarily deltas and data blocks for active I/O operations. The controller is connected to an HDD in any of the conventional interfaces.

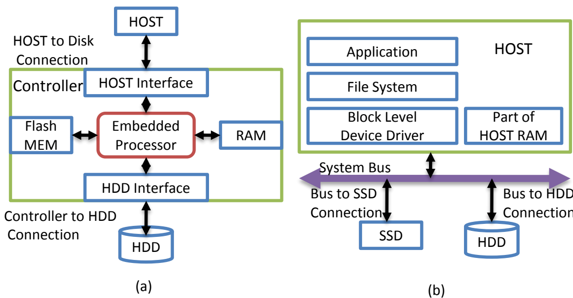

Figure 2. I-CASH implementations.

While the above hardware implementations inside a disk controller or HBA card can provide great performance benefits, they require purpose-built hardware. Another possible implementation is a software approach using commodity hardware. Figure 2(b) shows the block diagram of one software implementation of I-CASH in which a software module at the block device level controls standard off-the-shelf SSD and HDD. The software is running entirely on the host CPU with a part of the system RAM as a cache to temporarily buffer delta and data blocks. The software solution is easy to implement without requiring changes on the hardware but it consumes system resources such as CPU, RAM, and system bus for the necessary functionality of I-CASH. In addition, software implementation is OS dependent and requires different designs and implementations for different operating systems. However, with the rapid advances in multi core chip multiprocessors (CMP), computation power in today's servers and workstations is abundant. Trading such high performance and low cost computation for better I/O performance is a cost-effective and attractive solution to many applications.

3.3 Data Reliability and Recovery

I-CASH uses a DRAM buffer to store temporarily data blocks and delta blocks that are accessed by host I/O requests. In addition, data and changes are stored separately and a sector contains many deltas. As a result, data reliability and recovery become a concern. This issue can be addressed in two ways besides depending on lower level redundancy. First, dirty delta and metadata are flushed to disk periodically. There is a tradeoff here. For reliability purposes, we would like to perform write to HDD as soon as possible whereas for performance purposes we would like to pack as many deltas in one block as possible. The flush interval is a tunable parameter based on the number of dirty delta blocks in the system. The second approach is to employ a simple log structure similar to prior researches [4, 20, 46]. For data recovery after a failure, I-CASH can recover data by combining reference blocks with deltas unrolled from the delta logs in the HDD.

4. Prototype and Evaluation Methodology

4.1 Prototype Implementation

We have developed a proof-of-concept prototype of I-CASH using Kernel Virtual Machine (KVM). The prototype represents the realization of our I-CASH design using a software module and off-the-shelf hardware components. The functions that the prototype has implemented include identifying reference blocks, deriving deltas for write I/Os, serving read I/Os by combining deltas with reference blocks, and managing interactions between SSD and HDD. The current prototype carries out the necessary computations using the host CPU and uses a part of system RAM as the DRAM buffer of the I-CASH (The program is available at ele.uri.edu/hpcl).

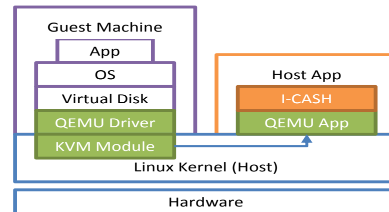

Figure 3. Prototype implementation of I-CASH.

The software module is implemented in the virtual machine monitor as shown in Figure 3. The I/O function

of the KVM depends on QEMU [3] that is able to emulate many virtual devices including virtual disk drive. The QEMU driver in a guest virtual machine captures disk I/O requests and passes them to the KVM kernel module. The KVM kernel module then forwards the requests to QEMU application and returns the results to the virtual machine after the requests are complete. The I/O requests captured by the QEMU driver are block-level requests of the guest virtual machine. Each of these requests contains the virtual disk address and data length. I-CASH is implemented within the QEMU application module and is therefore able to catch the virtual disk address and the length of an I/O request. The most significant byte of the 64-bit virtual disk address is used as the identifier of the virtual machine so that the requests from different virtual machines can be managed in one queue.

4.2 Program Structure and Data Layouts

In order to select reference blocks, we need to determine and keep track of both access frequency and content signature of a data block. For this purpose, each block is divided into *S* sub-blocks. A sub-signature is calculated for each of the *S* sub-blocks. A special two dimensional array, called Heatmap, is maintained in our design. The Heatmap has *S* rows and *Vs* columns, where *Vs* is the total number of possible signature values for a sub-block. For example, if the sub-signature is 8 bits, Vs = 256. Each entry in the Heatmap keeps a popularity value that is defined as the number of accesses of the sub-block matching the corresponding signature value. As an example, consider Figure 4 that shows the 8×256 Heatmap. In this example, each data block is divided into 8 sub-blocks and has 8 corresponding signature values. When a block is accessed with sub-block signatures being 55, 00, and so on as shown in Figure 4, the popularity value corresponding to column number 55 of the 1st row is incremented. Similarly, column number 0 of second row is also incremented. In this way, Heatmap keeps popularity values of all sub-signatures of sub-blocks.

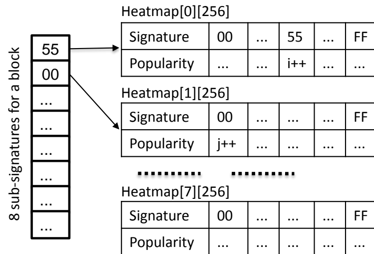

Figure 4.Sub-signatures and the Heatmap.

To illustrate how Heatmap is organized and maintained as I/O requests are issued, consider a simple example where each cache block is divided into 2 sub-blocks and each sub-signature has only four possible values, i.e. Vs = 4. The Heatmap of this example is shown in Table 1 for a sequence of I/O requests accessing data blocks at addresses *LBA1, LBA2, LBA3*, and *LBA4*, respectively. Assume that all possible contents of sub-blocks are *A, B, C*, and *D* and their corresponding signatures are *a, b, c*, and *d,* respectively. The Heatmap in this case contains 2 rows corresponding to 2 sub-blocks of each data block and 4 columns corresponding to 4 possible signature values. As shown in this table, all entries of the Heatmap are initialized to {(0, 0, 0, 0), (0, 0, 0, 0)}. Whenever a block is accessed, the popularities of corresponding sub-signatures in the Heatmap are incremented. For instance, the first block has logical block address (*LBA*) of *LBA1* with content (*A, B*) and signatures (*a, b*). As a result of the I/O request, two popularity values in the Heatmap are incremented corresponding to the two sub-signatures, and the Heatmap becomes {(1, 0, 0, 0), (0, 1, 0, 0)} as shown in Table 1. After 4 requests, the Heatmap becomes {(2, 1, 1, 0), (0, 1, 0, 3)}.

Table 1. The buildup of heatmap. Each block has 2 sub-blocks represented by 2 sub-signatures each having 4 possible values Vs=4.

I/O sequence	Content	Signature	Heatmap[0] a b c d	Heatmap[1] a b c d
		Initialized	0 0 0 0	0 0 0 0
LBA1	A B	a b	1 0 0 0	0 1 0 0
LBA2	C D	c d	1 0 1 0	0 1 0 1
LBA3	A D	a d	2 0 1 0	0 1 0 2
LBA4	B D	b d	2 1 1 0	0 1 0 3

In our current design, the size of a cache block is fixed at 4 KB. Each 4KB block is divided into 8 512-bytes sub-blocks resulting in 8 sub-signatures to represent the content of a block. Unlike many existing content addressable storage systems, each sub-signature is 1 byte representing the sum of 4 bytes in a sub-block at offsets 0, 16, 32, and 64, respectively. In this way, the computation overhead is substantially reduced compared with hash value computation of the whole sub-block. What is more important is that our objective is to find the similarity rather than identical blocks. Hashing is efficient to detect identical blocks, but it also lowers the chance of finding similarity because a single byte change results in a totally different hash value. Therefore, additional computation of hashing does not help in finding more similarities.

With 4KB blocks, 512B sub-blocks, and 8 bits sub-signature for each sub-block, we have Heatmap with 8 rows corresponding to 8 sub-blocks and 256 columns to hold all possible signatures that a sub-block can have. Each time a block is read or written, its 8 1-byte sub-signatures are retrieved and the 8 popularity values of corresponding entries in the Heatmap are increased by one. This frequency spectrum of contents is the key to identify reference blocks. It is able to capture both the temporal locality and the content locality. If a block is accessed twice, the increase of corresponding popularity value in the Heatmap reflects the temporal locality. On the other hand, if two similar blocks with different addresses are accessed once each, the Heatmap can catch

the content locality since the popularity values are incremented at entries that have matched signatures.

Similarity detection to identify reference blocks is done in two separate cases in the prototype implementation. The first case is when a block is first loaded, I-CASH searches for the same virtual address among the existing blocks in the cache. The second case is periodical scanning after every 2,000 I/Os. At each scanning phase, I-CASH checks the 4,000 blocks from the beginning of an LRU queue to find the blocks with the most frequently accessed signatures as references. The other blocks of these 4,000 blocks are compared with references. The association between newly found reference blocks and their respective delta blocks is reorganized at the end of each scanning phase.

Table 2. Selection of a reference block. The popularities of all blocks are calculated according to the Heatmap of Table 1.

LBAs	Block	Popularity	LRU	Reference			
				A B	C D	A D	B D
LBA1	A B	2+1 = 3	A B	A B	A B	_ B	A B
LBA2	C D	1+3 = 4	C D	C D	C D	C _	C _
LBA3	A D	2+3 = 5	A D	_ D	A _	A D	A _
LBA4	B D	1+3 = 4	B D	B D	B _	B _	B D
		Cache space	4	3.5	3	2.5	3

Table 2 shows the calculation of popularity values and the cache space consumption using different choices of reference block for the example of Table 1. The popularity value of a data block is the sum of all its sub-block popularity values in the Heatmap. As shown in the table, the most popular block here is the data block at address *LBA3* with content (*A, D*) and its popularity value is 5. Therefore, block (*A, D*) should be chosen as the reference block. Once the reference block is selected, delta-coding is used to eliminate data redundancy. The result shows that using the most popular block (*A, D*) as the reference, cache space usage is minimum, about 2.5 cache blocks assuming perfect delta encoding. Without considering content locality, a simple LRU would need 4 cache blocks to keep the same hit ratio. The saved space can be used to cache more data. Figure 5 shows the data layout after selecting block (*A, D*) as the reference block.

4.3 Data Management

I-CASH uses an LRU list of virtual blocks to manage data. Each virtual block contains the LBA address, the signature, the pointer to the reference block, the pointer to data block, and the pointer to delta blocks. A virtual block can be one of three different types: *reference block, associate block, or independent block*. An associate block is a virtual block that is associated with a reference block together with a delta that is the difference between the content of the associate block and the reference block. An independent block is a virtual block that has no associated reference block in the cache. Delta blocks are managed using a linked list of 64-bytes segments. A virtual block can have one or more delta blocks due to (i) this virtual block refers to a reference block; (ii) this virtual block is a reference block and has been written since it was selected as a reference.

When a disk block is accessed the first time and brought into the cache, a virtual block and a data block are allocated to cache it. Before this virtual block is selected as a reference block or associate block, it is an independent block so that data is read from or written to its data block. Its signature is updated upon every write request. Once it is selected as a reference block or associate block, one or more delta blocks are allocated for this virtual block. A write request to a virtual block that is an associate block needs to read its reference block first, calculate the difference using delta-coding, and write the difference to the delta block. Read request to an associate block combines its delta and the reference block to obtain its data. As a result, a reference block is always ahead of its associate blocks in the LRU queue because accesses to its associate blocks also need to access the reference block. Similarly, write requests to a reference block need update its delta blocks. But the signature of the block does not change since its data is being referred. Read requests to the changed reference block need combine with its delta block.

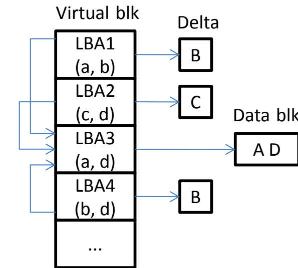

Figure 5. The data layout of I-CASH buffer cache.

To manage cached data blocks described above, we need to consider 3 kinds of replacements. The first is virtual block replacement when there is no available virtual block. I-CASH searches from the end of the LRU queue and replaces the first non-reference block. The second is data block replacement. I-CASH searches from the end of LRU queue and replaces the first data block. The data block of a reference block can also be evicted indicating that the reference block and its associate blocks have not been accessed for a long time. The third is delta replacement which leads to virtual block replacement. I-CASH searches from the end of the LRU queue, replaces the first virtual block that has delta and is not a reference. The data block, if exists, of the replaced virtual block is released because its content is invalid without delta blocks.

4.4 Experimental Setup and Workload Characteristics

The prototype I-CASH is installed on KVM running on a PC server that is a Dell PowerEdge T410 with

1.8GHz Xeon CPU, 8GB RAM, and 160G Seagate SATA drive. This PC server acts as the primary server. A Fusion-io ioDrive 80G SLC SSD is installed on the primary server. Another Dell Precision 690 with 1.6GHz Xeon CPU, 2GB RAM, and 400G Seagate SATA drive is used as the workload generator. The two servers are interconnected using a gigabit Ethernet switch. The operating system on both the primary server and the workload generator is Ubuntu 9.10 64bit. Multiple virtual machines, including Ubuntu 8.10, Ubuntu10.04, and Windows 2003, are built to execute a variety of benchmarks. The virtual machine RAM size used ranges from 128MB to 512MB depending on benchmarks as shown in the last column of Table 4.

In all experiments, the I-CASH software module runs on the host CPU with a partition of the system RAM to store delta blocks. The SSD in the I-CASH is the Fusion-io ioDrive 80G SLC and the HDD is the 160GB Seagate SATA drive. For performance comparison purpose, four baseline systems are setup on the same hardware environment:

1) Fusion-io: The first baseline system is using the Fusion-io ioDrive 80G SLC as the pure data storage with no HDD involved. All applications run on this SSD that stores the entire data set.
2) RAID0: The second baseline is RAID0 with data striping on 4 SATA disks. Linux MD is used as the RAID controller.
3) DeDup: The third baseline is data deduplication that saves only one copy of data in SSD for identical blocks.
4) LRU: The fourth case is using SSD as an LRU cache on top of the SATA disk drive.

Except for the first baseline, Fusion-io, that allocates enough SSD to store all application data, DeDup and LRU use exactly the same amount of SSD space as I-CASH which is typically about 10% of the size of data set for each benchmark.

Right workloads are important for performance evaluations. It should be noted that evaluating the performance of I-CASH is unique in the sense that I/O address traces are not sufficient because deltas are content dependent. That is, the workload should have data contents in addition to addresses. We have collected 6 standard I/O benchmarks available to the research community as shown in Table 3. Table 4 summarizes the characteristics of these benchmarks.

The first benchmark, SysBench, is a multi-threaded benchmark tool for evaluating the capability of a system to run a database under intensive load [27]. SysBench runs against MySQL database with a table of size 4,000,000, max requests of 100,000, and 16 threads.

Hadoop is currently one of the most important frameworks for large scale data processing [7]. Two Ubuntu 10.04 virtual machines are built to form a two-node Hadoop environment with default settings. The two virtual machines share one storage system. We measure the execution time of the MapReduce job, WordCount, to process the access log of our university website.

TPC-C is a benchmark modeling the operations of real-time transactions [12]. It simulates the execution of a set of distributed and on-line transactions (OLTP) on a number of warehouses. These transactions perform the basic database operations such as inserts, deletes, updates and so on. TPCC-UVA [33] is used on the Postgres database with 5 warehouses, 10 clients for each warehouse, and 30 minutes running time.

LoadSim2003 is a load simulator for Microsoft Exchange Server 2003 [56]. Multiple clients send messages to Exchange Server to simulate an email workload. In our test, the client computer simulates 100 clients of heavy user type and stress mode to access the Exchanger Server 2003 which is installed on Windows Server 2003. The duration of the simulation is 1 hour.

SPECsfs, is used to evaluate the performance of an NFS or CIFS file server. Typical file server workloads such as LOOKUP, READ, WRITE, CREATE, and REMOVE, etc are simulated. The benchmark results summarize the server's capability in terms of the number of operations that can be processed per second and the I/O response time. The client computer generates 100 LOADs on an Ubuntu8.10 NFS server.

RUBiS is a prototype that simulates an e-commerce server performing auction operations such as selling, browsing, and bidding similar to eBay [10]. To run this benchmark, each virtual machine on the server has installed Apache, MySQL, PHP, and RUBiS client. The database is initialized using the sample database provided by RUBiS. RUBiS runs with 300 clients and 15 minutes running time.

Table 3. Benchmarks used in performance evaluation.

Name	Description
SysBench	OLTP benchmark
MapReduce Word Counter	Hadoop example job
TPC-C	Database server workload
LoadSim2003	Exchange mail server benchmark
SPEC sfs	NFS file server
RUBiS	e-Commerce web server workload

Table 4. Characteristics of benchmarks.

	# of Read	# of Write	Avg. Read Len	Avg. Write Len	Data Size	VM RAM
SysBench	619K	236K	6656B	7680B	960MB	256MB
Hadoop	241K	62K	20992B	101376B	4.4GB	512MB
TPC-C	339K	156K	13312B	10752B	1.2GB	256MB
LoadSim	4329K	704K	12288B	11776B	17.5GB	512MB
SPEC-sfs	64K	715K	6144B	17408B	10GB	512MB
RUBiS	799K	7K	4608B	20480B	1.8GB	256MB
TPC-C 5VMs	256K	153K	23552B	23040B	5.2GB	256MB
RUBiS 5VMs	3396K	52K	5632B	25088B	10GB	256MB

5. Numerical Results and Evaluations

5.1 Performance

Our first experiment is on **SysBench**. Figure 6(a) shows the transaction rate results of SysBench running on the 5 different storage architectures. We allocated 128MB SSD space for I-CASH, LRU, and Dedup and the rest is stored in the HDD. It is interesting to observe from Figure 6(a) that I-CASH is able to finish more transactions per second than Fusion-io even though Fusion-io stores the entire data set in the SSD. Both LRU and DeDup provide better performance than RAID0 because of data locality that exists in this benchmark. Among all these storage architectures, I-CASH performs the best showing 2.24x faster than RAID0, 9% better than LRU, and 18% faster than DeDup.

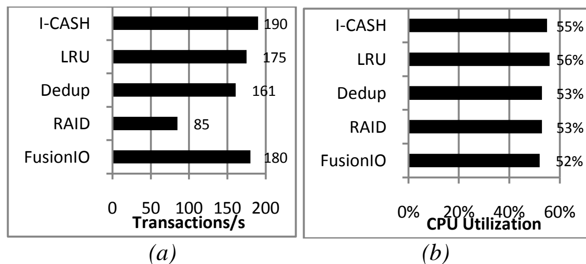

Figure 6. SysBench transaction rate and CPU utilization.

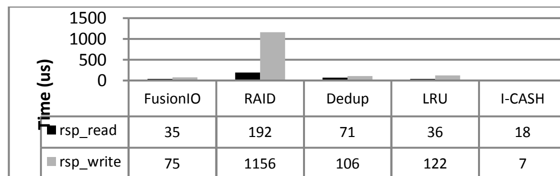

Figure 7. Response time of SysBench.

To better understand why I-CASH performs better than Fusion-io that stores the entire dataset in the SSD, we took a close look at how the two systems work by collecting more I/O statistics. Experiments showed that the percentages of reference blocks, delta blocks, and independent blocks are 1%, 85%, and 14%, respectively. That is, I-CASH is able to find 85% of data blocks that are similar in contents to 1% of reference blocks stored in the SSD and is able to cache all delta blocks within 32MB RAM. In addition, write operations to SSD are inevitable in Fusion-io whereas I-CASH seldom performs online writes to SSD because reference blocks stored in the SSD are relatively stable.

In order to further investigate the I/O behaviors of the systems, we measured the average response times for read I/Os and write I/Os while running the SysBench on the 5 different architectures as shown in Figure 7. Intuitively, the I/O response times of I-CASH should be longer than the first baseline system because of the additional computation time for compression/ decompression. However, Figure 7 shows that the average read time of I-CASH is half of that of Fusion-io and the average write time of I-CASH is more than 10 times faster than that of Fusion-io. Although such speedups are counter intuitive, careful analysis of the two systems makes it easily understandable. First of all, recall that only 1% of data blocks are reference blocks that are stored in SSD in I-CASH system. This means I-CASH accesses only 10MB SSD very frequently with mostly read I/Os while Fusion-io needs to access 1GB SSD with both read and write I/Os. Secondly, the most time consuming part of processing a write request is compression which can be done in parallel with I/O processing. It is observed that the time difference of accessing 4KB block between randomly accessing a 10MB file and randomly accessing 1GB file on Fusion-io is about 15µs. By keeping Fusion-io working at its peak speed, I-CASH is able to get average read response time of 18µs including 10µs decompression time. We further measured the average I/O times of Fusion-io with asynchronous writes and the results are similar to Figure 7 although write time is smaller but read time is larger.

Most performance gains of the I-CACH come from substituting disk I/Os by high speed computations. One obvious question is how much computation overhead that I-CASH incurs during I/O operations. Such computation overheads compete with normal applications that run on the same host CPU. We measured the CPU utilizations of the 5 storage systems while running the benchmarks. These CPU utilizations are shown in Figure 6(b). It was observed that the additional CPU busy time due to I-CASH algorithm is manageable. The CPU utilizations of all 5 systems are about the same with the difference less than 4%.

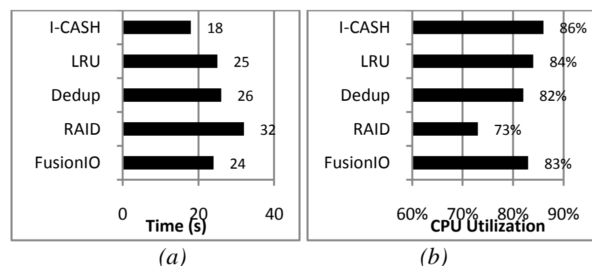

Figure 8. Hadoop performance and CPU utilization.

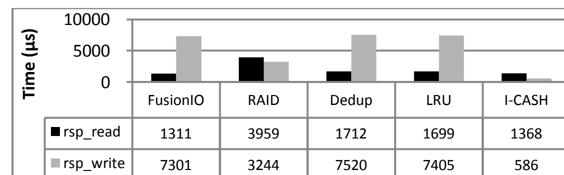

Figure 9. Response time of Hadoop.

The measured Hadoop execution times are shown in Figure 8(a) for the five different storage systems. It is clear from this figure that I-CASH out-performed all other baseline systems with speedups ranging from 1.3 to 1.8. For an I/O bound application, I-CASH clearly

showed superb performance advantages over the baseline systems. In this experiment, I-CASH uses 512MB SSD, which is about one-tenth of the total data set. The memory footprint to cache deltas is 256MB which is less than the memory requirement of Fusion-io driver. This required RAM cache size is expected to become smaller with further optimization of the delta compression algorithm. Figure 9 shows the detailed read and write response times measured at block I/O level. I-CASH has much shorter response time than other systems. The baseline system writes many modified data blocks to SSD, giving rise to large write response time due to slow SSD writes and possible erasure operations. Similar to previous experiments, the additional computation overhead of I-CASH is within a few percents as shown in Figure 8(b) except for RAID system that uses much less CPU resources than other systems.

Measured **TPC-C** results of the 5 different storage systems are shown in Figures 10(a) and 11 in terms of transaction rate and average application level response time, respectively. In this benchmark, clients commit small transactions frequently generating a large amount of write requests. As a result, I-CASH is able to improve the application level response time by 64% and 81% over Fusion-io and RAID0, respectively as shown in Figure 11. The fast write performance of I-CASH helps to reduce the total response time seen from application level. However, the actual speedup at application level depends on the fraction of the I/O time in the total execution time. Figure 10(a) shows that I-CASH can process 14% and 45% more transactions per minute than Fusion-io and RAID0, respectively. We noticed that RAID0 performs poorly because of a large amount of random and small transactions.

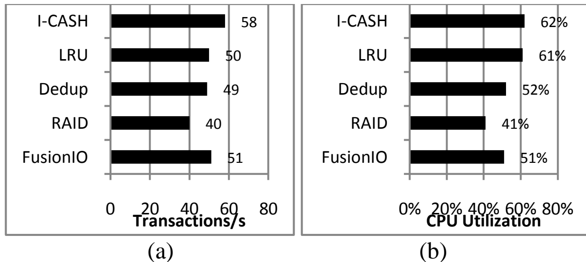

Figure 10. TPC-C performance and CPU utilization.

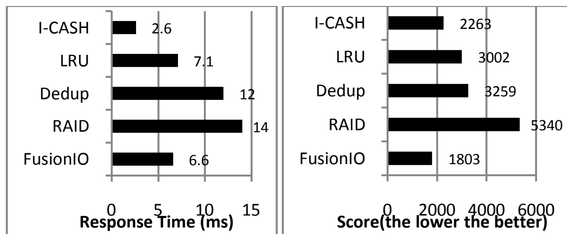

Figure 11. TPC-C RSP time. *Figure 12. LoadSim score.*

Figure 12 shows the measured LoadSim results in terms of score which is calculated based on response time. The Exchange Server database in our experiment is set to 16GB and I-CASH is configured to use 1GB SSD and the delta buffer of 256MB. From Figure 12 we can see that I-CASH is 2.4x faster than RAID0 but about 20% slower than Fusion-io. We noticed that I-CASH uses 1GB SSD implying that I-CASH can greatly improve the performance of the Exchange Server running on RAID with a small SSD device. Fusion-io is faster in this case because the workload generated by LoadSim is almost 100% random with little data locality. However, I-CASH is able to catch content locality and therefore performs much better than LRU and dedup caches as shown in Figure 12.

Figure 13 plots the measured response time while running SPEC sfs benchmark. I-CASH is configured to use 1GB SSD with 128MB RAM delta buffer. From this figure we can see that I-CASH performs as well as Fusion-io while using only one-tenth of the SSD space. As shown in Table 4, SPEC sfs is a write intensive benchmark. For Dedup cache, changing a block that is shared by several other identical blocks results in a new copy of data so that write performance is slowed down. The reduction of the response time of I-CASH over Dedup is 28% because I-CASH is able to exploit the content similarity between the new data and the old data to store only the changed data in small deltas.

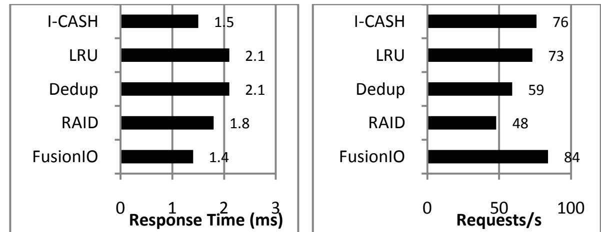

Figure 13. SPEC-sfs RSP time. Figure 14. RUBiS request rate.

RUBiS benchmark results are shown in Figure 14 in terms of number of requests finished per second. In this experiment, I-CASH uses 128MB SSD and 32MB delta buffer. As shown in Figure 14, I-CASH is 1.5x faster than RAID0 while 10% slower than baseline Fusion-io. Recall that over 90% of the requests are read requests (Table 4) in this benchmark, which limits the write performance advantage of I-CASH over Fusion-io. The major speedups of I-CASH over LRU and Dedup, which are 1.04 and 1.29, respectively, come from the capability of storing more data in the SSD to reduce disk accesses. The results show that the online similarity detection of I-CASH is effective under read intensive workloads. It is also observed that LRU cache is faster than Dedup though Dedup can store more data in the SSD. This is because the cost of dedup has overweighed the gain of extra SSD capacity in this case.

As discussed in Section 2, widespread use of virtual machines create additional burden to disk I/O systems. It is common to setup several similar virtual machines on

the same physical machine to run multiple services. Fast I/O performance becomes more important when multiple virtual machines compete for I/O resources. To evaluate how I-CASH performs in such virtual machine environment, we carried out experiments on multiple virtual machines. On each virtual machine, a distinct data set and benchmark parameters are used. The five TPC-C virtual machines use 1 to 5 warehouses, respectively. The five RUBiS machines use 20 to 24 items per page, respectively. I-CASH uses 512MB SSD and 512MB RAM for delta blocks for both benchmarks.

Figures 15 and 16 show the normalized performances of running TPC-C and RUBiS on multiple virtual machines, respectively. The performance advantage of I-CASH is clearly shown in these figures compared with the other four baseline systems. When the five virtual machines are running TPC-C benchmarks concurrently with different data sets, I-CASH provides 2.8x speedup over the baseline Fusion-io and over 5x to 6x speedup over the other three baseline systems. For RUBiS benchmark, the performance improvements are 20%, 6x, 4x, and 4x over baseline Fusion-io, RAID, Dedup, and LRU cache, respectively. Fusion-io performs fairly well for RUBiS benchmark as shown in Figure 16 because RUBiS is read intensive workload.

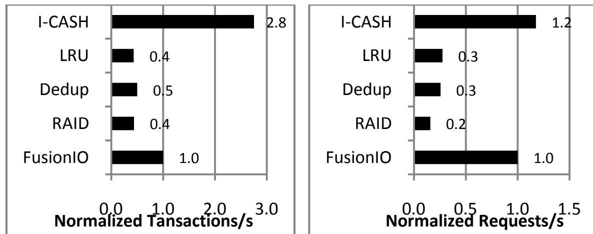

Figure 15. Five TPC-C VMs. Figure 16. Five RUBiS VMs.

5.2 Power Efficiency

Table 5 lists the energy consumption measured using a power meter (Electricity Usage Monitor) which can record power usage accumulatively while a benchmark is running. The power meter is connected to the power supply of the PC server and therefore the measured power consumption includes energy consumed by CPU, memory, and I/O operations. Since measuring energy consumption involves open the server box and reconnecting power supply and the power meter, we measured only two benchmarks as listed in Table 5. The numbers in Table 5 were calculated as follows. We first subtract the power consumption level when system is idle from the power consumption level while benchmarks are running. The difference is then multiplied by the benchmark running time resulting in Walt-Hours. RAID0 has 4 disks, 15 Walts each, and consumed 240% more energy than I-CASH for Hadoop benchmark and 150% more for TPC-C benchmark. Fusion-io (including the system disk), LRU, and Dedup use the same hardware as I-CASH but took longer time to finish the same Hadoop job resulting in more energy consumption. The energy saving of I-CASH also comes from less write requests to the SSD given that each 4KB read and write operation consumes 9.5μJ and 76.1μJ [47]. I-CASH saved about 12% energy compared to baseline Fusion-io running Hadoop. For TPC-C benchmark, the power consumptions of the 4 systems are comparable as shown in the table. The power savings compared to RAID are mainly attributed to the use of SSD as opposed to multiple HDDs.

Table 5. Power consumption in terms of Walt-hours.

	Hadoop	TPC-C
Fusion-io	8	11
RAID	24	28
Dedup	10	11
LRU	10	12
I-CASH	7	11

5.3 Prolonged SSD Life Time

Finally, we measured the number of write I/Os performed on SSD of the 5 different storage systems. Only four benchmarks with a large percentage of write I/Os are measured as shown in Table 6. Recall that our preliminary prototype does not strictly disallow random writes to SSD. For blocks that have deltas larger than the threshold value (2048 byte in the current implementation), the new data are written directly to the SSD to release delta buffer. Nevertheless, random writes to SSD are still substantially smaller than LRU and DeDup baseline systems. For SysBench, Hadoop, and TPC-C, I-CASH performs much less write I/Os to SSD than the baseline Fusion-io does. For benchmark SPEC sfs, the write operations in SSD of the two systems are comparable. The write I/O reductions of I-CASH imply prolonged life time of the SSD as discussed previously.

Table6. Number of write requests on SSD.

	SysBench	Hadoop	TPC-C	SPEC sfs
Fusion-io	893,700	2,540,124	1,173,741	5,752,436
Dedup	1,419,023	3,082,196	1,963,988	5,559,698
LRU	1,494,220	3,469,785	2,051,511	5,514,935
I-CASH	232,452	1,521,399	359,919	5,096,890

6. Conclusions

In this paper, a novel disk I/O architecture has been presented to exploit the high random access speed of flash memory SSDs. The idea of the new disk I/O architecture is intelligently coupling an array of SSD and HDD, referred to as I-CASH, in such a way that read I/Os are done mostly in SSD and write I/Os are done in HDD in batches by packing deltas with respect to the reference blocks stored in the SSD. By making use of the computing power of CPU and exploiting regularity and content locality of I/O data blocks, I-CASH achieved high I/O performance. Many I/O operations that would have been mechanical operations in HDDs are now replaced by high speed computations and SSD reads. A preliminary prototype of I-CASH has been built on Linux OS to provide a proof-of-concept of I-CASH.

Performance evaluation experiments using standard I/O benchmarks have shown great performance improvements over RAID0 and traditional systems using SSD as a storage cache. In some cases, I-CASH even performs better than SSD only storage that stores the entire data set with no HDD. As a future research, we are building a hardware prototype using an embedded processor in order to fully realize the performance potential of I-CASH.

Acknowledgements

This research is supported in part by National Science Foundation under Grants CCF-0811333, CPS-0931820, CCF-1017177 and Natural Science Foundation of China under grant NSFC-60736013. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation. The authors are grateful to anonymous referees for their comments that improve the quality of the paper.

References

[1] N. Agrawal, V. Prabhakaran, T. Wobber, J. Davis, M. Manasse, and R. Panigrahy, "Design Tradeoffs for SSD Performance," in *Proc. of USENIX Annual Technical Conference*, Boston, MA, 2008, pp. 57-70.

[2] M. Ajtai, R. Burns, R. Fagin, D. Long, and L. Stockmeyer, "Compactly Encoding Unstructured Inputs with Differential Compression," *Journal of the ACM (JACM),* vol. 49, pp. 318-367, 2002.

[3] F. Bellard, "QEMU, a Fast and Portable Dynamic Translator," in *Proc. of USENIX Annual Technical Conference*, 2005.

[4] M. Bhadkamkar, J. Guerra, L. Useche, S. Burnett, J. Liptak, R. Rangaswami, and V. Hristidis, "BORG: Block-reORGanization for Self-optimizing Storage Systems," in *Proc. of USENIX Conference on File and Storage Technologies*, 2009, pp. 183-196.

[5] A. Birrell, M. Isard, C. Thacker, and T. Wobber, "A Design for High-Performance Flash Disks," *ACM SIGOPS Operating Systems Review,* vol. 41, pp. 88-93, 2007.

[6] S. Boboila and P. Desnoyers, "Write Endurance in Flash Drives: Measurements and Analysis," in *Proc. of USENIX Conference on File and Storage Technologies*, San Jose, California, 2010.

[7] D. Borthakur, "The Hadoop Distributed File System: Architecture and Design," *http://hadoop.apache.org/ core/docs/current/hdfs_design.pdf,* 2007.

[8] A. Broder, "Identifying and Filtering Near-Duplicate Documents," in *Proc. of 11th Annual Symposium on Combinatorial Pattern Matching*, 2000, pp. 1-10.

[9] B. Calder, G. Reinman, and D. Tullsen, "Selective Value Prediction," in *Proc. of 26th International Symposium on Computer Architecture (ISCA'99)*, 1999, p. 74.

[10] E. Cecchet, J. Marguerite, and W. Zwaenepoel, "Performance and Scalability of EJB Applications," in *Proc. of 17th ACMConference on Object-oriented programming, systems, languages, and applications*, 2002, pp. 246-261.

[11] A. Clements, I. Ahmad, M. Vilayannur, and J. Li, "Decentralized Deduplication in SAN Cluster File Systems," in *Proc. of USENIX Annual Technical Conference*, 2009, pp. 102-114.

[12] Transaction Processing Performance Council, "TPC Benchmark™C Standard Specification," *http://tpc.org/ tpcc,* 2005.

[13] B. Debnath, S. Sengupta, and J. Li, "ChunkStash: Speeding up Inline Storage Deduplication using Flash Memory," in *Proc. of USENIX Annual Technical Conference*, 2010.

[14] C. Dirik and B. Jacob, "The Performance of PC Solid-State Disks (SSDs) as a Function of Bandwidth, Concurrency, Device Architecture, and System Organization," in *Proc. of 36th International Symposium on Computer Architecture (ISCA 2009)*, 2009, pp. 279-289.

[15] F. Douglis, "The Compression Cache: Using On-line Compression to Extend Physical Memory," in *Proc. of 1993 Winter USENIX Conference.*, 1993, pp. 519-529.

[16] Fusion-io, "Fusion-io ioDrive specification sheet," *http://www.fusionio.com/*.

[17] F. Gabbay and A. Mendelson, "Can Program Profiling Support Value Prediction?," in *Proc. of 30th Annual ACM/IEEE Int. Symposium on Microarchitecture*, 1997, pp. 270-280.

[18] L. Grupp, A. Caulfield, J. Coburn, S. Swanson, E. Yaakobi, P. Siegel, and J. Wolf, "Characterizing Flash memory: Anomalies, Observations, and Applications," in *Proc. of 42nd Annual IEEE/ACM International Symposium on Microarchitecture* 2009, pp. 24-33.

[19] D. Gupta, S. Lee, M. Vrable, S. Savage, A. Snoeren, G. Varghese, G. Voelker, and A. Vahdat, "Difference Engine: Harnessing Memory Redundancy in Virtual Machines," in *Proc. of 8th USENIX Symposium on Operating Systems Design and Implementation*, 2008.

[20] Y. Hu and Q. Yang, "DCD---Disk Caching Disk: A New Approach for Boosting I/O Performance," in *Proc. of 23rd Annual International Symposium on Computer Architecture (ISCA'96)*, Philadelphia, PA, 1996, pp. 169-178.

[21] J. Huang and D. Lilja, "Exploiting Basic Block Value Locality With Block Reuse," in *Proc. of 5th IEEE International Symposium on High-Performance Computer Architecture*, 1999, pp. 106-114.

[22] S. Jiang, K. Davis, and X. Zhang, "Coordinated Multilevel Buffer Cache Management with Consistent Access Locality Quantification," *IEEE Transactions on Computers,* vol. 15, pp. 95-108, 2007.

[23] W. Josephson, L. Bongo, D. Flynn, and K. Li, "DFS: A File System for Virtualized Flash Storage," in *Proc. of USENIX Conference on File and Storage Technologies*, 2010, pp. 85–100.

[24] T. Kgil, D. Roberts, and T. Mudge, "Improving NAND Flash Based Disk Caches," in *Proc. of 35th International Symposium on Computer Architecture (ISCA 2008)*, Beijing, China, 2008, pp. 327-338.

[25] D. Klein, "The Future of Memory and Storage: Closing the Gaps," *Micron Technology, Inc. ,* 2007.

[26] R. Koller and R. Rangaswami, "I/O Deduplication: Utilizing Content Similarity to Improve I/O Performance," in *Proc. of USENIX Conference on File and Storage Technologies*, 2010.
[27] A. Kopytov, "SysBench, a System Performance Benchmark," http://sysbench.sourceforge.net/, 2004.
[28] E. Kruus, C. Ungureanu, and C. Dubnicki, "Bimodal Content Defined Chunking for Backup Streams," in *Proc. of USENIX Conference on File and Storage Technologies*, 2010.
[29] P. Kulkarni, F. Douglis, J. LaVoie, and J. Tracey, "Redundancy Elimination within Large Collections of Files," in *Proc. of USENIX Annual Technical Conference*, 2004, pp. 59-72.
[30] S. Lee, B. Moon, C. Park, J. Kim, and S. Kim, "A Case for Flash Memory SSD in Enterprise Database Applications," in *Proc. of ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2008, pp. 1075-1086.
[31] A. Liguori and E. Hensbergen, "Experiences with Content Addressable Storage and Virtual Disks," in *Proc. of Workshop on I/O Virtualization USENIX Association*, 2008.
[32] M. Lipasti, "Value Locality and Speculative Execution," PhD thesis, Carnegie Mellon University, 1997.
[33] D. Llanos, "TPCC-UVa: an Open-Source TPC-C Implementation for Global Performance Measurement of Computer Systems," *ACM SIGMOD Record,* vol. 35, p. 15, 2006.
[34] V. Mohan, T. Siddiqua, S. Gurumurthi, and M. Stan, "How I Learned to Stop Worrying and Love Flash Endurance," in *Proc. of the 2nd USENIX Conference on Hot Topics in Storage and File Systems* 2010.
[35] C. Morrey III and D. Grunwald, "Peabody: The Time Travelling Disk," in *Proc. of IEEE Mass Storage Conference*, San Diego, CA, 2003.
[36] D. Narayanan, E. Thereska, A. Donnelly, S. Elnikety, and A. Rowstron, "Migrating Server Storage to SSDs: Analysis of Tradeoffs," in *Proc. of fourth ACM European conference on Computer systems*, 2009, pp. 145-158.
[37] P. Nath, M. Kozuch, D. O'hallaron, J. Harkes, M. Satyanarayanan, N. Tolia, and M. Toups, "Design Tradeoffs in Applying Content Addressable Storage to Enterprise-Scale Systems Based on Virtual Machines," in *Proc. of USENIX Annual Technical Conference*, 2006.
[38] F. Oliveira, G. Guardiola, J. Patel, and E. Hensbergen, "Blutopia: Stackable Storage for Cluster Management," in *Proc. of IEEE Cluster*, 2007, pp. 293-302.
[39] D. Patterson, G. Gibson, and R. Katz, "A Case for Redundant Arrays of Inexpensive Disks (RAID)," in *Proc. of ACM SIGMOD International Conference on Management of Data*, 1988, pp. 109-116.
[40] T. Pritchett and M. Thottethodi, "SieveStore: A Highly-selective, Ensemble-level Disk Cache for Cost-Performance," in *Proc. of 37th International Symposium on Computer Architecture (ISCA 2010)*, 2010, pp. 163-174.
[41] S. Quinlan and S. Dorward, "Venti: a new approach to archival storage," in *Proc. of First USENIX Conference on File and Storage Technologies*, Monterey, CA, 2002.
[42] D. Reimer, A. Thomas, G. Ammons, T. Mummert, B. Alpern, and V. Bala, "Opening Black Boxes: Using Semantic Information to Combat Virtual Machine Image Sprawl," in *Proc. of ACM/Usenix International Conference On Virtual Execution Environments*, 2008, pp. 111-120.
[43] S. Rhea, R. Cox, and A. Pesterev, "Fast, Inexpensive Content-Addressed Storage in Foundation," in *Proc. of USENIX Annual Technical Conference*, Boston, Massachusetts, 2008, pp. 143-156.
[44] Y. Sazeides, "An Analysis of Value Predictibality and its Application to a Superscalar Processor," PhD thesis, University of Wisconsin, Madison, 1999.
[45] A. Sodani and G. Sohi, "Dynamic Instruction Reuse," in *Proc. of 24th International Symposium on Computer Architecture*, 1997, pp. 194-205.
[46] G. Soundararajan, V. Prabhakaran, M. Balakrishnan, and T. Wobber, "Extending SSD Lifetimes with Disk-Based Write Caches," in *Proc. of 8th USENIX Conference on File and Storage Technologies*, 2010.
[47] G. Sun, Y. Joo, Y. Chen, D. Niu, Y. Xie, and H. Li, "A Hybrid Solid-State Storage Architecture for the Performance, Energy Consumption, and Lifetime Improvement," in *Proc. of 16th IEEE International Symposium on High-Performance Computer Architecture*, 2010, pp. 141–153.
[48] I. Tuduce and T. Gross, "Adaptive Main Memory Compression," in *Proc. of USENIX Annual Technical Conference*, Anaheim, CA, 2005.
[49] C. Ungureanu, B. Atkin, A. Aranya, S. Gokhale, S. Rago, G. Całkowski, C. Dubnicki, and A. Bohra, "HydraFS: a High-Throughput File System for the HYDRAstor Content-Addressable Storage System," *8th USENIX Conference on File and Storage Technologies,* 2010.
[50] P. Wilson, S. Kaplan, and Y. Smaragdakis, "The Case for Compressed Caching in Virtual Memory Systems," in *Proc. of USENIX Annual Technical Conference*, 1999.
[51] M. Wu and W. Zwaenepoel, "eNVy: A Non-Volatile, Main Memory Storage System," in *Proc. of 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1994, pp. 86-97.
[52] G. Yadgar, M. Factor, and A. Schuster, "Karma: Know-it-All Replacement for a Multilevel Cache," in *Proc. of USENIX Conference on File and Storage Technologies*, 2007, pp. 25-25.
[53] Q. Yang, W. Xiao, and J. Ren, "TRAP-Array: A disk Array Architecture Providing Timely Recovery to Any Point-in-Time," in *Proc. of 33rd Annual International Symposium on Computer Architecture*, 2006.
[54] Y. Zhou, Z. Chen, and K. Li, "Second-Level Buffer Cache Management," *IEEE Transactions on Parallel and Distributed Systems,* vol. 15, pp. 505-519, 2004.
[55] B. Zhu, K. Li, and H. Patterson, "Avoiding the Disk Bottleneck in the Data Domain Deduplication File System," in *Proc. of 6th USENIX Conference on File and Storage Technologies*, San Jose,CA, 2008.
[56] A. Zinman, "Simulating Stress for your Exchange 2003 Hardware using LoadSim 2003," *http://www.msexchange.org/tutorials/Simulating-Stress-Exchange-2003-LoadSim.html,* 2004.