FlashStore: High Throughput Persistent Key-Value Store

Biplob Debnath*
University of Minnesota
Twin Cities, USA
biplob@umn.edu

Sudipta Sengupta
Microsoft Research
Redmond, USA
sudipta@microsoft.com

Jin Li
Microsoft Research
Redmond, USA
jinl@microsoft.com

ABSTRACT

We present FlashStore, a high throughput persistent key-value store, that uses flash memory as a non-volatile *cache* between RAM and hard disk. FlashStore is designed to store the working set of key-value pairs on flash and use one flash read per key lookup. As the working set changes over time, space is made for the current working set by destaging recently unused key-value pairs to hard disk and recycling pages in the flash store. FlashStore organizes key-value pairs in a log-structure on flash to exploit faster sequential write performance. It uses an in-memory hash table to index them, with hash collisions resolved by a variant of cuckoo hashing. The in-memory hash table stores compact key signatures instead of full keys so as to strike tradeoffs between RAM usage and false flash read operations.

FlashStore can be used as a high throughput persistent key-value storage layer for a broad range of server class applications. We compare FlashStore with BerkeleyDB, an embedded key-value store application, running on hard disk and flash separately, so as to bring out the performance gain of FlashStore in not only using flash as a cache above hard disk but also in its use of flash aware algorithms. We use real-world data traces from two data center applications, namely, Xbox LIVE Primetime online multi-player game and inline storage deduplication, to drive and evaluate the design of FlashStore on traditional and low power server platforms. FlashStore outperforms BerkeleyDB by up to 60x on throughput (ops/sec), up to 50x on energy efficiency (ops/Joule), and up to 85x on cost efficiency (ops/sec/dollar) on the evaluated datasets.

1. INTRODUCTION

A broad range of server-side applications need an underlying, often persistent, key-value store to function. Examples include state maintenance in Internet applications like online multi-player gaming and inline storage deduplication. A high throughput persistent key-value store can help to improve the performance of such applications. Flash memory is a natural choice for such a store, providing persistency and 100-1000 times lower access times than hard disk. Compared to DRAM, flash access times are about 100 times higher. Flash stands in the middle between DRAM and disk also in terms of cost – it is 10x cheaper than DRAM, while 20x more expensive than disk – thus, making it an ideal gap filler between DRAM and disk.

There are two types of popular flash devices, NOR and NAND flash. NAND flash architecture allows a denser layout and greater storage capacity per chip. As a result, NAND flash memory has been significantly cheaper than DRAM, with cost decreasing at faster speeds. NAND flash characteristics have lead to an explosion in its usage in consumer electronic devices, such as MP3 players, phones, caches and Solid State Disks (SSDs). In the rest of the paper, we use NAND flash based SSDs as the architectural choice and simply refer to it as flash memory. We describe SSDs in detail in Section 2. To get the maximum performance per dollar out of SSDs, it is necessary to use flash aware data structures and algorithms to avoid small random writes that not only have a higher latency but also reduce flash device lifetimes through increased page wearing.

In this paper, we present the design and evaluation of FlashStore, a high performance key-value storage system using flash as a cache between RAM and hard disk. When a key-value blob is written, it is sequentially logged in flash. A specialized RAM-space efficient hash table index using a variant of cuckoo hashing [31] and compact key signatures is used to index the key-value blobs stored in flash memory. FlashStore attempts to capture the working set of key-value pairs and make them accessible using one flash read. It monitors the occupancy levels of RAM and flash, and when either reach configurable thresholds, recently unused key-value blobs are destaged to hard disk and their associated index entries are removed from RAM as well. Because FlashStore is designed to capture the working set of key-value pairs on flash and their associated index in RAM, it achieves an ideal balance between the performance, cost, and energy consumption of DRAM/flash/disk, and is able to provide a huge persistent key-value store at reduced cost.

The contributions of this paper are summarized as follows:

- **Key-value store using flash as persistent cache over hard disk:** FlashStore is designed to store the working set of key-value pairs on flash and using one flash read per key lookup. As the working set changes over time, space is made for the current working set

*Work is done during Summer Internship at Microsoft

VLDB '10, September 13-17, 2010, Singapore

by destaging recently unused key-value pairs to hard disk and recycling pages in the flash store. FlashStore organizes key-value pairs in a log-structure on flash to exploit fast sequential writes.

- **Specialized space efficient RAM hash table index:** FlashStore uses an in-memory hash table to index key-value pairs on flash, with hash collisions resolved by a variant of cuckoo hashing. The in-memory hash table stores compact key signatures instead of full keys so as to strike tradeoffs between RAM usage and false flash reads.

- **Evaluation on real-world data center applications:** We compare FlashStore with BerkeleyDB, an embedded key-value store application, running on hard disk and flash separately, so as to bring out the performance gain of FlashStore in not only using flash as a cache above hard disk but also in its use of flash aware algorithms. We use real-world data traces from two data center applications, namely, Xbox LIVE Primetime online multi-player game and inline storage deduplication, to drive and evaluate the design of FlashStore on traditional and low power server platforms. Our evaluations show that FlashStore outperforms BerkeleyDB by up to 60x on throughput (ops/sec), up to 50x on energy efficiency (ops/Joule), and up to 85x on cost efficiency (ops/sec/dollar).

The rest of the paper is organized as follows. We provide an overview of flash memory in Section 2. In Section 3, we describe two motivating real-world data center applications that can benefit from a high throughput key-value store and are used to evaluate FlashStore. We develop the design of FlashStore in Section 4. We evaluate FlashStore and compare it with BerkeleyDB in Section 5. We review related work in Section 6. Finally, we conclude in Section 7.

2. FLASH MEMORY OVERVIEW

Figure 1 gives a block-diagram of an NAND flash based SSD. In flash memory, data is stored in an array of flash blocks. Each block spans 32-64 pages, where a page is the smallest unit of read and write operations. A distinguishing feature of flash memory is that read operations are very fast compared to magnetic disk drive. Moreover, unlike disks, random read operations are as fast as sequential read operations as there is no mechanical head movement. A major drawback of the flash memory is that it does not allow in-place updates (i.e., overwrite). Page write operations in a flash memory must be preceded by an erase operation and within a block, pages need be to written sequentially. The *in-place update* problem becomes complicated as write operations are performed in the page granularity, while erase operations are performed in the block granularity. The typical access latencies for read, write, and erase operations are 25 microseconds, 200 microseconds, and 1500 microseconds, respectively [10].

The Flash Translation layer (FTL) is an intermediate software layer inside SSD, which makes linear flash memory device act like a virtual disk. The FTL receives logical read and write commands from the applications and converts them to the internal flash memory commands. To emulate disk like in-place update operation for a logical page (L_p),

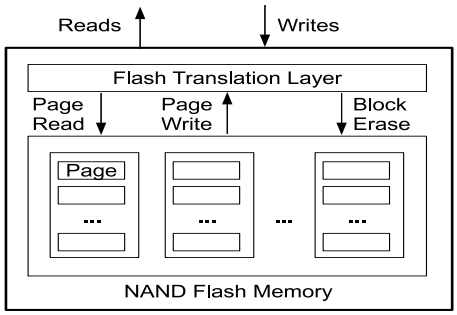

Figure 1: Solid State Disk (SSD)

the FTL writes data into a new physical page (P_p), maintains a mapping between logical pages and physical pages, and marks the previous physical location of L_p as invalid for future garbage collection. Although FTL allows current disk based application to use SSD without any modifications, it needs to internally deal with flash physical constraint of erasing a block before overwriting a page in that block. Besides the *in-place update* problem, flash memory exhibits another limitation – a flash block can only be erased for limited number of times (e.g., 10K-100K) [10]. FTL uses various wear leveling techniques to even out the erase counts of different blocks in the flash memory to increase its overall longevity [19]. Recent studies show that current FTL schemes are very effective for the workloads with sequential access write patterns. However, for the workloads with random access patterns, these schemes show very poor performance [15, 21, 24, 26, 29, 27]. One of the design goals of FlashStore is to use flash memory in an FTL friendly manner.

3. KEY-VALUE STORE APPLICATIONS

We describe two real-world applications that can use FlashStore as an underlying persistent key-value store. Data traces obtained from real-world instances of these applications are used to drive and evaluate FlashStore's design.

3.1 Online Multi-player Gaming

Online multi-player gaming technology allows people from geographically diverse regions around the globe to participate in the same game. The number of concurrent players in such a game could range from tens to hundreds of thousands and the number of concurrent game instances offered by a single online service could range from tens to hundreds. An important challenge in online multi-player gaming is the requirement to scale the number of users per game and the number of simultaneous game instances. At the core of this is the need to maintain server-side state so as to track player actions on each client machine and update global game states to make them visible to other players as quickly as possible. These functionalities map to `set` and `get` key operations performed by clients on server-side state. The real-time responsiveness of the game is, thus, critically dependent on the response time and throughput of these operations.

There is also the requirement to store server-side game state in a persistent manner for (at least) the following reasons: (i) resume game from interrupted state if and when

crashes occur, (ii) offline analysis of game popularity, progression, and dynamics with the objective of improving the game, and (iii) verification of player actions for fairness when outcomes are associated with monetary rewards. We designed FlashStore to meet the high throughput and low latency requirement of such `get-set` key operations in online multi-player gaming.

3.2 Storage Deduplication

Deduplication is a recent trend in storage backup systems that eliminates redundancy of data across full and incremental backup data sets [37]. It works by splitting files into multiple chunks using a content-aware chunking algorithm like Rabin fingerprinting and using SHA-1 hash signatures for each chunk to determine whether two chunks contain identical data [37]. In *inline* storage deduplication systems, the chunks (or their hashes) arrive one-at-a-time at the deduplication server from client systems. The server needs to lookup each chunk hash in an index it maintains for all chunk hashes seen so far for that backup location instance – if there is a match, the incoming chunk contains redundant data and can be deduplicated; if not, the (new) chunk hash needs to be inserted into the index.

Because storage systems currently need to scale to tens of terabytes to petabytes of data volume, the chunk hash index is too big to fit in RAM, hence it is stored on hard disk. Index operations are thus throughput limited by expensive disk seek operations. Since backups need to be completed over windows of half-a-day or so (e.g., nights and weekends), it is desirable to obtain high throughput in inline storage deduplication systems. RAM prefetching and bloom-filter based techniques used by Zhu et al. [37] can avoid disk I/Os on close to 99% of the index lookups. Even at this reduced rate, an index lookup going to disk contributes about 0.1msec to the *average* lookup time – this is about 10^3 times slower than a lookup hitting in RAM. FlashStore can be used as the chunk hash index for inline deduplication systems. By reducing the penalty of index lookup misses in RAM by orders of magnitude by serving such lookups from flash memory, FlashStore can help to increase deduplication throughput.

4. FLASHSTORE DESIGN

We present the system architecture of FlashStore and the rationale behind some design choices in this section. FlashStore's design is driven by the need to work around two types of operations that are not efficient on flash media, namely:

1. **Random Writes:** Small random writes effectively need to update data portions within pages. Since a (physical) flash page cannot be updated in place, a new (physical) page will need to be allocated and the unmodified portion of the data on the page needs to be relocated to the new page.

2. **Writes less than flash page size:** Since a page is the smallest unit of write on flash, writing an amount less than a page renders the rest of the (physical) page wasted – any subsequent append to that partially written (logical) page will need copying of existing data and writing to a new (physical) page.

To validate the performance gap between sequential and random writes on flash, we used Iometer [5], a widely used

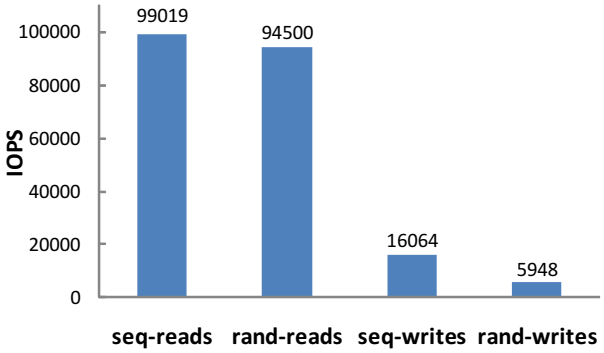

Figure 2: IOPS for sequential/random reads and writes using 4KB I/O request size on a 160GB fusionIO drive.

performance evaluation tool in the storage community, on a 160GB fusionIO SSD [4] attached over PCIe bus to an Intel Core 2 Duo E6850 3GHz CPU. The number of worker threads was fixed at 8 and the number of outstanding I/Os for the drive at 64. The results for IOPS (I/O operations per sec) on 4KB I/O request sizes are summarized in Figure 2. Each test was run for 1 hour. The IOPS performance of sequential writes is about 3x that of random writes and worsens when the tests are run for longer durations (due to accumulating device garbage collection overheads). We also observe that the IOPS performance of (random/sequential) reads is about 6x that sequential writes. (The slight gap between IOPS performance of sequential and random reads is possibly due to prefetching inside the device.)

Given the above, the most efficient way to write flash is to simply use it as an append log, where an append operation involves a flash page worth of data, typically 2KB or 4KB. This is the main constraint that drives the rest of our key-value store design. Flash has been used in a log-structured manner and its benefits reported in earlier work ([22, 36, 30, 16].

4.1 Design Goals

The design of FlashStore is driven by the following guiding principles:

- **Support Low-latency, high throughput operations.** This requirement is extracted from the needs of many server class applications that need an underlying key-value store to function. Two motivating applications that are used for evaluating FlashStore are described in Section 3.

- **Use flash aware data structures and algorithms.** This principle accommodates the constraints of the flash device so as to extract maximum performance out of it. Random writes and in-place updates are expensive on flash memory, hence must be reduced or avoided. Sequential writes should be used to the extent possible and the fast nature of random/sequential reads should be exploited.

- **Low RAM footprint per key independent of key-value size.** The goal here is to index all key-value pairs on flash in a RAM space efficient manner and make them accessible using one flash read. By being RAM space frugal, one can accommodate larger

flash drive capacities and correspondingly larger number of key-value pairs stored in it. Key-value pairs can be arbitrarily large but the RAM footprint per key should be independent of it and small (say, about 10 bytes).

4.2 Architectural Components

Our key-value store system has the following main components, as shown in Figure 3:

RAM Write Buffer: This is a fixed-size data structure maintained in RAM that buffers key-value writes so that a write to flash happens only after there is enough data to fill a flash page (which is typically 2KB or 4KB in size). To provide strict durability guarantees, writes can also happen to flash when a configurable timeout interval (e.g., 1 msec) has expired (during which period multiple key-value pairs are collected in the buffer). The client call returns only after the write buffer is flushed to flash. The RAM write buffer is sized to 2-3 times the flash page size so that key-value writes can still go through when part of the buffer is being written to flash.

RAM Hash Table (HT) Index: The index structure, for key-value pairs stored on flash, is maintained in RAM and is organized as a hash table with the design goal of one flash read per lookup. The index maintains pointers to the full key-value pairs stored on flash. Key features include resolving collisions using a variant of cuckoo hashing and storing compact key signatures in memory to tradeoff between RAM usage and false flash reads. We explain these aspects in Section 4.4. By destaging recently unused key-value pairs from flash to hard disk when either RAM or flash bottlenecks are reached, we eliminate the need for rehashing.

RAM Read Cache: This is a fixed-size read cache of recently read items that is maintained in RAM. We use a Least Recently Used (LRU) policy [32] to evict key-value pairs when inserting items into a full cache.

Recency Bit Vector: A recency bit vector in RAM is used to record the entries in the HT index that have been recently accessed. This is used by the flash recycling thread to determine whether a valid key-value pair on flash should be destaged to hard disk or not (as described in Section 4.5). The i-th entry of the bit vector provides recency information about the entry stored in the i-th slot of the HT index.

Disk-presence Bloom Filter: A *disk-presence bloom filter* in RAM is used to record keys destaged to hard disk so that hard disk access latencies can be avoided when lookups are done on non-existing keys. (Bloom filters are surveyed in [13].)

Flash Store: The flash store provides persistent storage for the key-value pairs and is organized as a *recycled* append log. Key-value pairs are written to flash in units of a page size. The pages on flash are used in a circular linked list order – entries in recycled pages are *destaged* to hard disk based on their validity and recently accessed status. The eviction algorithm from flash store to hard disk uses the recency bit vector described above.

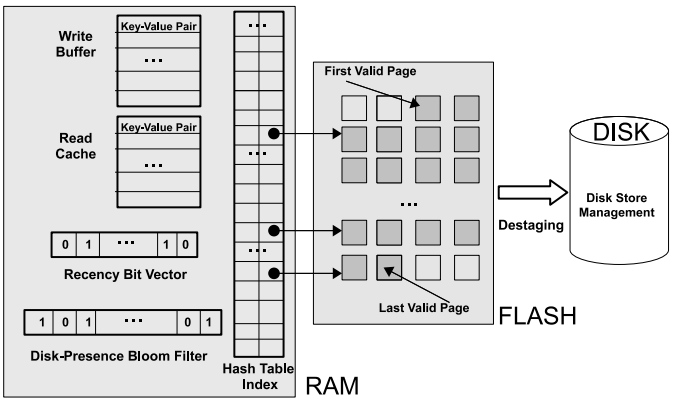

Figure 3: FlashStore architecture schematic showing the different hierarchical storage areas.

Hard Disk Store: The hard disk store serves to store all key-value pairs that have been evicted from flash because of page recycling. Because key lookups can miss in RAM and flash, we need to index the hard disk store also so as to provide fast access to keys stored here. In the current design, we use Berkeley DB [1], an embedded key-value database, as the hard disk store.

4.3 Key Lookup and Insert Operations

To understand the hierarchical relationship of the different storage areas in our design, it is helpful to understand the sequence of accesses in key insert and lookup operations. We describe this next.

A *key lookup* operation (`get`) first looks up the RAM read cache. Upon a miss there, it looks up the RAM write buffer. Upon a miss there, it searches the RAM HT Index in order to locate the key on flash store. Upon a miss there, it looks up the disk-presence bloom filter – if the key is not present, it returns `null`, otherwise it searches the hard disk store for the key. If the key is found at any place other than the RAM read cache, it is inserted into the RAM read cache and a key evicted from there if it was full. The corresponding bit in the recency bit vector is also set to indicate that this key has been recently accessed.

A *key insert* (or, *update*) operation (`set`) writes the key-value pair (together with its timestamp) into the RAM write buffer. If (an earlier value of) the key exists in RAM read cache, it will be invalidated. When there are enough key-value pairs in RAM write buffer to fill a flash page (or, a configurable timeout interval since the client call has expired, say 1 msec), these entries are written to flash and inserted to the RAM HT index. When flash usage exceeds a certain threshold, previously used pages are recycled. During this recycle operation, valid keys on recycled pages, depending on their access pattern, are either reinserted into the flash store or destaged to hard disk.

A *delete* operation on a key is supported through insertion of a `null` value for that key. As an optimization, the entry in RAM HT index can be deleted after the insertion goes to flash. When this is done, the entry for that key in the hard disk store needs to be deleted also. Eventually the *null* entry on flash will be garbage collected.

The functionalities of (i) client key lookup/insert operations, (ii) writing key-value pairs to flash store and updating RAM HT index, and (iii) recycling flash pages and destaging key-value pairs to hard disk are handled by separate threads in a multi-threaded architecture. Concurrency issues with shared data structures arise in our multi-threaded design, which we do not describe here due to lack of space.

4.4 Hash Table Design

We outline the salient aspects of the hash table design in FlashStore.

Resolving hash collisions using cuckoo hashing: Hash function collisions on keys result in multiple keys mapping to the same hash table index slot – these need to be handled in any hashing scheme. Two common techniques for handling such collisions include *linear probing* and *chaining* [25]. Linear probing can increase lookup time arbitrarily due to long sequences of colliding slots. Chaining hash table entries in RAM, on the other hand, leads to increased memory usage, while chaining buckets of key-value pairs is not efficient for use with flash, since partially filled buckets will map to partially filled flash pages that need to be appended over time, which is not an efficient flash operation. Moreover, the latter will result in multiple flash page reads during key lookups and writes, which will degrade performance.

FlashStore structures the HT index as an array of slots and uses a variant of *cuckoo hashing* [31] to resolve collisions. Cuckoo hashing provides flexibility for each key to be in one of $n \geq 2$ candidate positions and for later inserted keys to relocate earlier inserted keys to any of their other candidate positions– this keeps the linear probing chain sequence upper bounded at n. In fact, the value proposition of cuckoo hashing is in increasing hash table load factors while keeping lookup times bounded to a constant. A study [38] has shown that cuckoo hashing is much faster than chained hashing as hash table load factors increase. The name "cuckoo" is derived from the behavior of some species of the cuckoo bird – the cuckoo chick pushes other eggs or young out of the nest when it hatches, much like the hashing scheme kicks previously inserted items out of their location as needed.

In the variant of cuckoo hashing we use, we work with n random hash functions $h_1, h_2, \ldots, h_n$ that are used to obtain n *candidate positions* for a given key x. These candidate position indices for key x are obtained from the lower-order bit values of $h_1(x), h_2(x), \ldots, h_n(x)$ corresponding to a modulo operation. During insertion, the key is inserted in the first available candidate slot. When all slots for a given key x are occupied during insertion (say, by keys $y_1, y_2, \ldots, y_n$), room can be made for key x by relocating keys y_i in these occupied slots, since each key y_i has a choice of $(n-1)$ other locations to go to. In the original cuckoo hashing scheme [31], a recursive strategy is used to relocate one of the keys y_i – in the worst case, this strategy could take many key relocations or get into an infinite loop, the probability for which can be shown to be very small and decreasing exponentially in n [31]. In our design, the system attempts a small number of key relocations after which it makes room by picking a key to move to an auxiliary linked list. In practice, by dimensioning the HT index for a certain load factor and by choosing a suitable value of n, such events can be made extremely rare, as we investigate in Section 5.5. Hence, the size of this linked list is small.

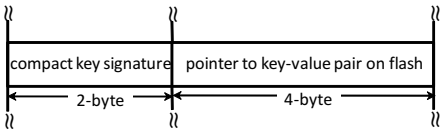

Figure 4: RAM HT Index entry and example sizes in FlashStore. (The all-zero pointer is reserved to indicate an empty HT index slot.)

Reducing RAM usage per slot by storing compact key signatures: Traditional hash table designs store the respective key in each entry of the hash table index [25]. Depending on the application, the key size could range from few tens of bytes (e.g., 20-byte SHA-1 hash) to hundreds of bytes or more. Given that RAM size is limited (in the order of few gigabytes) and is much costly than flash, if we store the full key in each entry of the RAM HT index, it may well become the bottleneck for the maximum number of entries in HT index before flash storage capacity bounds kick in. On the other hand, if we do not store the key at all in the HT index, the search operation on the HT index would have to follow HT index pointers to flash to determine whether the key stored in that slot matches the search key – this would lead to many *false flash reads*, which are expensive, since flash access speeds are 2-3 orders of magnitude slower than that of RAM.

To address the goals of maximizing HT index capacity (number of entries) and minimizing false flash reads, we store a *compact key signature* (order of few bytes) in each entry of the HT index. This signature is derived from *both the key and the candidate position number that it is stored at*. In FlashStore, when a key x is stored in its candidate position number i, the signature in the respective HT index slot is derived from the higher order bits of the hash value $h_i(x)$. During a search operation, when a key y is looked up in its candidate slot number j, the respective signature is computed from $h_j(y)$) and compared with the signature stored in that slot. Only if a match happens is the pointer to flash followed to check if the full key matches. We investigate the percentage of false reads as a function of the compact signature size in Section 5.5.

Storing key-value pairs to flash: Key-value pairs are organized on flash in a log-structure in the order of the respective write operations coming into the system. The HT index contains pointers to key-value pairs stored on flash. We use a 4-byte pointer, which is a combination of a page pointer and a page offset. Consider a 160GB flash SSD with 4KB pages, which is representative of SSDs currently selling in the market. Then, a page number can be specified with $\log_2$ (160GB/4KB) = 26 bits. The remaining 6 bits can be used for in-page offset, which point to 128B boundaries in a 4KB page. We thus aligned the stored key-value pairs at 128B boundaries. FlashStore reserves the all-one pointer to indicate an empty HT index slot.

RAM HT Index Sizing. FlashStore is designed to use a small number of bytes in RAM per entry so as to maxi-

mize the RAM HT index capacity for a given RAM usage size. The RAM HT index capacity determines the number of key-value piars stored on flash that can be accessed with one flash read. The RAM size for the HT index can be determined with application requirements in mind. With a 2-byte compact key signature and 4-byte flash pointer per entry, which is a total of 6 bytes per entry as shown in Figure 4, a typical RAM usage of 4GB per machine for the HT index accommodates a maximum of about 715 million entries.

Whether RAM or flash capacity becomes the bottleneck for storing the working set of keys on flash depends on the key-value pair size. With 64-byte key-value pairs, 715 million entries in the HT index occupy 42GB on flash that is well within the capacity range of SSDs shipping in the market today (from 64GB to 640GB). When there are multiple such SSDs attached, additional RAM is needed to fully utilize them. On the other hand, with 1024-byte key-value pairs, these same number of entries in the HT index will need 672GB of flash for storage, hence more than one flash disk may be needed.

4.5 Flash Storage Management

Key-value pairs are organized on flash in a log-structure in the order of the respective write operations coming into the system. When there are enough key-value pairs in the RAM write buffer to fill a flash page (or, when a pre-specified coalesce time interval is reached), they are written to flash. Each flash page begins with a header portion, contains metadata information including the time when the page was written, number of key-value pairs in the page, and begin offset for each. Each key-value entry on flash also has an associated write operation timestamp.

The pages on flash are maintained *implicitly* as a circular linked list. Since the Flash Translation Layer (FTL) translates logical page numbers to physical ones, this circular linked list can be easily implemented as a contiguous block of logical page address with wraparound, realized by two page number variables, one for the first valid page (oldest written) and the other for the last valid page (most recently written).

When RAM HT index exceeds a target maximum load factor (say 90%) or flash usage exceeds a certain threshold (say 80%), a cleaning operation is performed to bring the usage below this threshold. The cleaning operation considers currently used flash pages in *oldest first* order and deallocates them in a way similar to garbage collection in log-structured file systems. One each page, the sequence of key-value pairs are scanned to determine whether they are valid or not. A key-value pair on a flash page is invalid (or, *orphaned*) if the record in the HT index for that key does not point to this entry on this flash page – this happens when a later write to the key happened. When a key-value entry is determined to be valid, its access pattern information (maintained in RAM) is used to determine whether it should be reinserted into flash or destaged to disk. (This flash eviction policy is described next.) In the former case, it is inserted into the RAM write buffer (for later re-entry into flash). In the latter case, it is written to the index on hard disk and the corresponding entry in the HT index marked as empty. After all the key-value pairs on the current flash page are processed, the page is recycled and the first page number variable is incremented. We choose destaging thresholds and strategies based on past work [20].

Flash Eviction Policy. FlashStore is designed to store a superset of the working set of key-value pairs on flash to the extent that the working set fits on flash. As the working set changes, space is made for key-value pairs in the current working set by destaging earlier key-value pairs to hard disk and recycling pages in the flash store. To implement this functionality, the flash recycling thread needs to determine whether a key has been recently accessed. This is achieved by storing a recency bit vector in RAM that records entries in the HT index that have been recently accessed. When a key in the i-th slot of the HT index is accessed, the i-th bit of the vector is set to 1. (Note that when a key is relocated due to insertion of another key in cuckoo hashing, its recency bit needs to be relocated in the bit vector also.)

During the flash recycling operation, the flash recycling thread checks the recency bit vector to determine whether a valid key-value pair on flash has been recently accessed. If so, it reinserts it into the RAM write buffer, otherwise it destages the key-value pair to hard disk (and also clears the corresponding bit in the bit vector in both cases). In effect, when a key is reinserted into the RAM write buffer, it gets a second chance and may end up being removed in subsequent flash recycling iterations. This use of the recency bit is similar to the class of clock or second-chance based FIFO page replacement algorithms that aim to approximate LRU [17].

4.6 Crash Recovery

FlashStore's persistency guarantee enables it to recover from system crashes due to power failure or other reasons. Because the system logs all key-value write operations to flash, it is straightforward to rebuild the HT index in RAM by scanning all valid flash pages on flash. Recovery using this method can take some time, however, depending on the total size of valid flash pages that need to be scanned and the read throughput of the flash memory.

If crash recovery needs to be executed faster so as to support "near" real-time recovery, then it is necessary to checkpoint the RAM HT index periodically into flash (in a separate area from the key-value pair logs). Then, recovery involves reading the last written HT index checkpoint from flash and scanning key-value pair logged *flash pages with timestamps after that* and inserting them into the restored HT index. During the operation of checkpointing the HT index, all insert operations into it will need to be suspended (but the read operations by other threads can continue). The flash writing thread can continue with flash writing operations during this time but cannot insert items into the HT index. We use a temporary, small in-RAM hash table to provide index for the interim items. After the checkpointing operation completes, key-value pairs from the flash pages written in the interim are inserted into the HT index. Key lookup operations, upon missing in the HT index, will need to check in these flash pages (via the small additional hash table) until the latter insertions into HT index are complete. The flash recycling thread is suspended during the HT index checkpointing operation, since the recycling thread cannot set HT index entries to null.

4.7 Extending FlashStore to Multiple Nodes

The current design of FlashStore focuses on maximizing throughput of a single node. The design should be extensible in achieving high overall throughput when the system is extended to multiple nodes, and existing ideas in the literature can be used for this. One approach is to use a one-hop Distributed Hash Table (DHT) based on consistent hashing to map the key space across multiple nodes [33, 18]. A second and simpler approach is to use hash function based partitioning of keys across nodes, with each node protected by buddy pair machines. (This design, of course, has the issue that new nodes cannot be added easily, since a hash function does not have the locality preserving redistribution properties of consistent hashing.) Extending FlashStore to a multiple node system is the focus of future work.

5. EVALUATION

We evaluate FlashStore on two different computing platforms on real-world traces obtained from the two applications described in Section 3.

5.1 C# Implementation

We have prototyped FlashStore in approximately 2500 lines of C# code. MurmurHash [6] is used to realize the hash functions used in our variant of cuckoo hashing to compute hash table indices and compact signatures for keys; different seeds are used to generate different hash functions in this family. The `ReaderWriterLockSlim` and `Monitor` classes in .NET 3.0 framework [3] are used to implement the concurrency control solution for multi-threading.

5.2 Comparison with BerkeleyDB

We compare FlashStore with BerkeleyDB [1], an embedded key-value database that is widely used as a comparison benchmark for its good performance. BerkeleyDB does not use flash aware algorithms but we used the parameter settings recommended in [1] to improve its performance with flash. We use BerkeleyDB in its non-transactional concurrent data store mode that supports a single writer and multiple readers [35]. This mode does *not* support a transactional data store with the ACID properties, hence provides a fair comparison with FlashStore (which supports some of the ACID properties, e.g., durability). BerkeleyDB provides a choice of B-tree and hash table data structures for building indexes – we use the hash table version which we found to run substantially faster. We use the C++ implementation of BerkeleyDB with C# API wrappers [2].

To make a fair performance comparison with the persistency guarantee provided by FlashStore, we configured the standalone BerkeleyDB solution (on either flash or hard disk) to flush to stable storage after key-value writes accumulate to 4KB worth of data (which is the page size on flash). Without this, we found that BerkeleyDB buffers writes in RAM so long as it has sufficient available RAM and flushes to stable storage only when the database is closed – clearly, this does not provide the persistency guarantee that comes with FlashStore. This modification slows down BereleyDB but makes it an "apples-to-apples" comparison with FlashStore.

To eliminate any possible performance improvements in FlashStore due to caching in RAM for comparison purposes, we turned off the RAM read cache in FlashStore for the experiments. (Any RAM based caching used in BerkeleyDB is left "as-is".) Also, for the traces used for the evaluations (see Table 2), all of the key-value pairs fit on the flash SSDs we used (see Table 1) – hence, no destaging activity to hard disk is incurred in FlashStore. To evaluate the impact of destaging activity on FlashStore performance (Section 5.9), we explicitly reduce the amount of flash used by FlashStore to lower levels. We also turned off the disk-presence bloom filter in FlashStore described in Section 4.2.

Trace	Total get-set ops	get:set ratio	Avg. size (bytes)	
			Key	Value
Xbox	5.5 million	7.5:1	92	1200
Dedup	40 million	2.2:1	20	44

Table 2: Properties of the two traces used in the performance evaluation of FlashStore.

5.3 Evaluation Platforms

We use two different platforms to evaluate the performance of FlashStore and compare it with BerkeleyDB. The first platform, **Blue**, is a traditional server configuration using high power CPU, fusionIO flash SSD [4], and hard disk. The second platform, **Green**, is a low power "green" platform using a low power CPU, Samsung flash SSD [7], and low-power hard disk. The details of these two platforms are provided in Table 1. The RAM power consumption figures were obtained using the estimate of 878mW per 1GB of DDR2 DRAM given in [23]. The chassis/motherboard cost, which includes CPU and RAM but not flash or HDD, is indicated in the last column of the table.

For each of these platforms, we compare the throughput (operations per sec) on the two traces described in Table 2 for the following systems: (i) FlashStore on flash (FlashStore), (ii) BerkeleyDB on flash (BerkeleyDB-SSD), and (iii) BerkeleyDB on hard disk (BerkeleyDB-HDD). It is obvious that Blue will provider better performance over Green in terms of throughput but the question we also explore is that at what cost this higher performance comes. To this end, we also compare the (i) number of operations per unit energy and (i) throughput per dollar for the above three systems on the two traces on the two platforms in Section 5.8.

5.4 Evaluation Datasets

We described two real-world applications in Section 3 that can use FlashStore as an underlying persistent key-value store. Data traces obtained from real-world instances of these applications are used to drive and evaluate FlashStore's design.

Xbox LIVE Primetime trace

We evaluate the performance of FlashStore on a large trace of `get-set` key operations obtained from real-world instances of the Microsoft Xbox LIVE Primetime online multiplayer game [9]. In this application, the key is a dot-separated sequence of strings with total length averaging 94 characters and the value averages around 1200 bytes. The ratio of `get` operations to set `set` operations is about 7.5.

Storage Deduplication trace

We have built a storage deduplication analysis tool that can crawl a root directory, generate the sequence of chunk hashes for a given average chunk hash size, and compute the number of deduplicated chunks and storage bytes. The enterprise data backup trace we use for evaluations in this

Platform	CPU		RAM		Flash (SSD)			Hard Disk (HDD)			Chassis
	Type	Power	Size	Power	Type	Cost	Power	Type	Cost	Power	Cost
Blue	Intel Core 2 Duo E8500 @3.16GHz	65W	4GB	3.5W	fusionIO 80GB	$2200	15W	Seagate Barracuda 250GB 7200rpm	$50	12W	$1150
Green	Intel Atom 330 @1.60GHz	8W	2GB	1.8W	Samsung 64GB	$500	0.4W	Seagate Momentus 160GB 5400rpm	$50	1.6W	$200

Table 1: Blue and Green: The two evaluation platforms, their components, power usage, and cost.

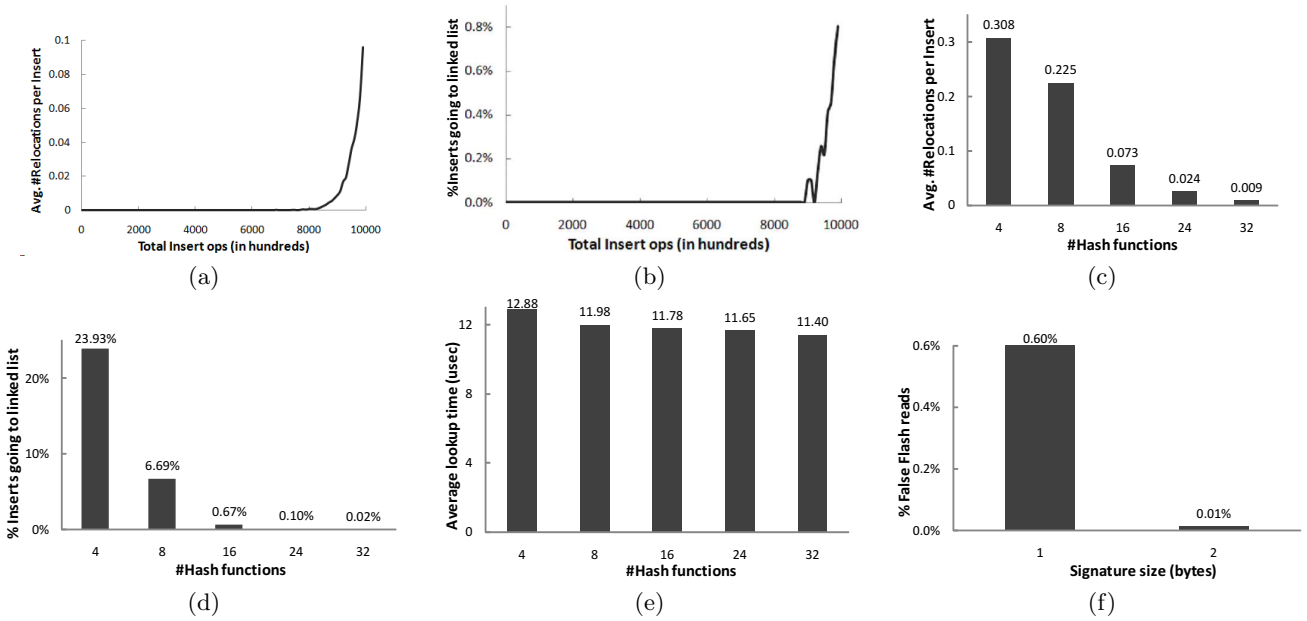

Figure 5: Tuning hash table parameters in FlashStore: (a) Average number of relocations per insert and (b) Percentage of inserts going to linked list, as keys are inserted into hash table; (c) Average number of relocations per insert, (d) Percentage of inserts going to linked list, and (e) Average lookup time (μsec) on Blue platform, vs. number of hash functions, averaged between 75%-90% load factors; (f) Percentage of false flash reads during lookup vs. signature size (bytes) stored in hash table.

paper was obtained by our storage deduplication analysis tool using 4KB chunk sizes. The trace contains 27,748,824 total chunks and 12,082,492 unique chunks. Using this, we obtain the ratio of `get` operations to `set` operations in the trace as 2.2:1. In this application, the key is a 20-byte SHA-1 hash of the corresponding chunk and the value is the meta-data for the chunk, with key-value pair total size upper bounded by 64 bytes.

The properties of the two traces are summarized in Table 2. Both traces include `get` operations on keys that have not been `set` earlier in the trace. (Such `get` operations will return `null`.) This is an inherent property of the nature of the application, hence we play the traces "as-is" to evaluate throughput in operations per second. For tuning hash table parameters in Section 5.5, we need to measure the performance of `get` and `set` operations separately. We do this using the storage deduplication trace but with some modifications as explained there.

5.5 Tuning Hash Table Parameters

Before we make performance comparisons of FlashStore with BerkeleyDB, we need to tune two parameters in our hash table design from Section 4.4, namely, (a) number of hash functions used in our variant of cuckoo hashing, and (b) size of compact key signature. These affect the throughput of read key and write key operations in different ways that we discuss shortly. For this set of experiments, we use the storage deduplication trace in a modified way so as to study the performance of hash table insert and lookup operations separately. We pre-process the trace to extract a set of 1 million unique keys, then insert all the key-value pairs, and then read all these key-value pairs in random order.

As has been explained earlier, the value proposition of cuckoo hashing is in accommodating higher hash table load factors without increasing lookup time. It has been shown through theoretical analysis that with 3 or more hash functions and with load factors up to 91%, insertion operations succeed in expected constant time [31]. With this prior evidence in hand, we target a maximum load factor of 90% for our cuckoo hashing implementation. Hence, for the dedup trace used in this section with 1 million keys, we fix the number of hash table slots to 1.1 million.

Number of Hash Functions. When the number of hash functions n is small, the performance of insert operations can be expected to degrade in two ways as the hash table loads up. First, an insert operation will find all its n slots

occupied by other keys and the number of cascaded key relocations required to complete this insertion will be high. Since each key relocation involves a flash read (to read the full key from flash and compute its candidate positions), the insert operation will take more time to complete. Second, with an upper bound on the number of allowed key relocations (which we set to 100 for this set of experiments), the insert operation could lead to a key being moved to the auxiliary linked list – this increases the RAM space usage of the linked list as well as its average search time. On the other hand, as the number of hash functions increases, lookup time will increase because of increasing number of hash function computations per lookup. However, the latter undesirable effect is not expected to be as degrading as those for insertions, since a single hash function computation (in CPU) takes orders of magnitude less time than a flash read. We study these effects to determine a suitable number of hash functions for our design.

The performance of insert operations degrades as the hash table loads up, as seen by the impact of the above two effects in Figures 5(a) and 5(b) – here, for $n = 24$ hash functions, we plot the average number of key relocations (hence, flash reads) per insert operation and the fraction of insert operations going to the auxiliary linked list respectively, as keys are inserted into hash table, both averaged over every 10,000 insert operations. Hence, for the following plots in this section, we present average numbers between 75% and 90% load factors as the insert operations are performed.

In Figure 5(c), we plot the average number of key relocations per insert operation as the number of hash functions n is varied, $n = 4, 8, 16, 24, 32$. Beyond $n = 16$ hash functions, the hash table incurs less than 0.1 key relocations (hence, flash reads) per insert operation. In Figure 5(d), we see that the fraction of insert operations that go to the linked list drops below 0.001 beyond $n = 24$ hash functions. On the other hand, Figure 5(e) shows that there is no appreciable increase in average lookup time as the number of hash functions increase. (The slight decrease in average lookup time with increasing number of hash functions can be attributed to faster search times in the auxiliary linked list, whose size decreases as number of hash functions increases.)

Based on these effects, we choose $n = 24$ hash functions in the RAM HT Index for our FlashStore implementation. Note that during a key lookup, all n hash values on the key need not be computed, since the lookup stops at the candidate position number the key is found in. We want to add that using fewer hash functions may be acceptable depending on overall performance requirements, but we do not recommend a number below $n = 8$. Also, with $n = 24$ hash functions, we observed that it is sufficient to set the maximum of allowed retries to 5-10 to keep the number of inserts that go to the linked list to the low value in Figure 5(d).

Compact Key Signature Size. As explained in Section 4.4, we store compact key signatures in the HT index (instead of full keys) to reduce RAM usage. However, shorter signatures lead to more *false flash reads* during lookups – when a pointer into flash is followed because the signature matches, but the full key on flash does not match with the key being looked up. We study this effect in Figure 5(f) where we fix the number of hash functions to $n = 24$. (We did not find much variation in the fraction of false flash reads when number of hash functions is increased so long as $n \geq 8$.) We observe that the fraction of false reads drops sharply when the number of signature bytes increases from 1 to 2.

Since flash reads are expensive compared to RAM reads, the design should strike a balance between reducing false flash reads and RAM usage. Based on the above numbers, we fix the signature size to 2 bytes in our implementation. Even a 1-byte signature size may be acceptable given that only 0.6% of the flash reads are false in that case.

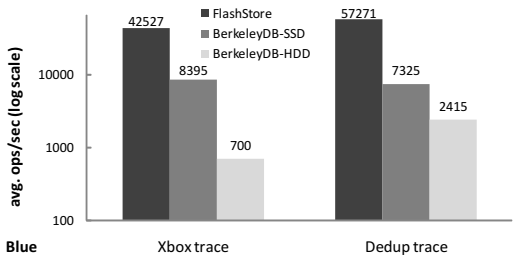

Figure 6: Blue platform: Comparative throughput (ops/sec) evaluation of FlashStore and BerkeleyDB on the two traces.

5.6 Evaluation on Blue Platform

We ran the Xbox LIVE Primetime online multi-player game and storage deduplication traces (from Table 2) on FlashStore and BerkeleyDB systems. We log the average number of ops/sec (`get-set` operations per sec) at a period of every 10,000 operations during each run and then take the overall average over a run to obtain throughput numbers shown in Figure 6 for the Blue platform.

FlashStore achieves an average throughput of 42,527 ops/sec on the Xbox trace. This is about 5x higher than that of BerkeleyDB on flash SSD and about 60x higher than that of BerkeleyDB on HDD. On the dedup trace, FlashStore achieves a higher average throughput of 57,271 ops/sec. The throughput of BerkeleyDB also increases on the dedup trace. This can be explained as follows. The write mix per unit operation in the dedup trace is about 2.65 times that of the Xbox trace. However, since the key-value pair size is about 20 times smaller for the dedup trace, the number of syncs to stable storage per *write operation* is about 20 times less. Overall, the number of syncs to stable storage per *unit operation* is about 7.6 times less for the dedup trace. Hence, both FlashStore and BerkeleyDB obtain higher throughputs on the dedup trace, though the gain is more for BerkeleyDB since it benefits more from fewer syncs which translate to random writes in BerkeleyDB but sequential writes in FlashStore. Moreover, because of this, the gains of FlashStore over BerkeleyDB on the dedup trace are less than the corresponding ones for the Xbox trace – on the dedup trace, the average throughput of FlashStore is about 8x higher than that of BerkeleyDB on flash SSD and about 24x higher than that of BerkeleyDB on HDD. A second reason for this is that the Xbox trace has many updates to (earlier written) keys (while the dedup trace has none because of its application nature) and these translate to in-place updates to stable storage in BerkeleyDB.

Platform	Xbox trace					
	ops/Joule			ops/sec/dollar		
	FlashStore	BerkeleyDB-SSD	BerkeleyDB-HDD	FlashStore	BerkeleyDB-SSD	BerkeleyDB-HDD
Blue	509	100	9	13	2	0.6
Green	830	489	42	12	7	2

Table 3: Blue vs. Green: Comparison of power & cost aware metrics for FlashStore and BerkeleyDB on Xbox trace.

Platform	Dedup trace					
	ops/Joule			ops/sec/dollar		
	FlashStore	BerkeleyDB-SSD	BerkeleyDB-HDD	FlashStore	BerkeleyDB-SSD	BerkeleyDB-HDD
Blue	686	88	30	17	0.2	2
Green	1067	197	89	16	3	4

Table 4: Blue vs. Green: Comparison of power & cost aware metrics for FlashStore and BerkeleyDB on dedup trace.

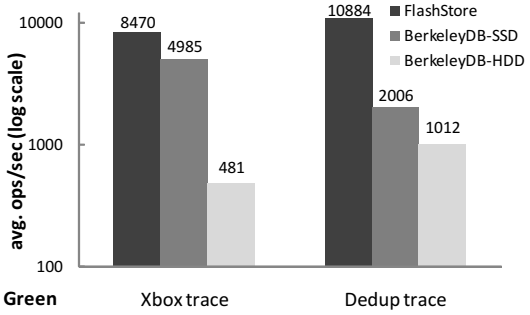

Figure 7: Green platform: Comparative throughput (ops/sec) evaluation of FlashStore and BerkeleyDB on the two traces.

5.7 Evaluation on Green Platform

We next compare the throughput (`get-set` operations per sec) performance of FlashStore and BerkeleyDB on the Green platform for the Xbox LIVE Primetime and storage deduplication traces (from Table 2). As for the Blue platform, we log the average number of ops/sec at a period of every 10,000 operations during each run and then take the overall average over a run to obtain throughput numbers shown in Figure 7 for the Green platform.

FlashStore achieves an average throughput of 8,470 ops/sec on the Xbox trace and 10,884 ops/sec on the dedup trace. These numbers indicate a reduction of about 5x in throughput compared to the Blue platform. This is expected because of the low power nature of the Green platform. A more appropriate comparison between the Blue and Green platforms should involve performance per unit cost and/or per unit power – we present this in Section 5.8. Most of the relative performance comparisons between FlashStore and BerkeleyDB on the two traces and their explanations for the Blue platform also hold for the Green platform.

5.8 Blue vs. Green Platforms: Power and Cost Aware Metrics

We undertake a comparison between FlashStore and BerkeleyDB on the Blue and Green platforms involving operations per unit energy (Joule) and throughput (ops/sec) per unit cost (dollar). The first metric obtain (ops/Joule) is obtained by dividing the throughput (ops/sec) of the application by the power usage (Joules/sec) of the platform. for computing this metric, we obtained power usage numbers for CPU, RAM, flash SSD, and HDD for each platform as provided in Table 1. We did not have access to total motherboard power usage information for either platform, hence the ops/Joule metrics numbers we present should be interpreted in a relative sense and not used for absolute comparison with benchmarks of other systems.

The second metric (ops/sec/dollar) is obtained by dividing the throughput (ops/sec) of the system by the upfront dollar investment in the platform and attempts to approximate a TPC-style benchmark [8] for key-value stores. The latter is obtained by adding the chassis/motherboard cost (which is indicated in the last column of Table 1) to the cost of the respective flash SSD or HDD used by the application. These metrics are provided in Tables 3 and 4 for the Xbox and dedup traces respectively.

For the first metric of ops/Joule, FlashStore achieves about 500 ops/Joule on the Xbox trace and 700 ops/Joule on the dedup trace on the Blue platform. The corresponding numbers for the Green platform are about 60% higher. Thus, the Green platform is more energy efficient than the Blue platform, but that comes at the cost of lower absolute throughput, a tradeoff that is often present in low power computing systems. The ops/Joule provided by BerkeleyDB (on either SSD or HDD) is about an order of magnitude lower than that of FlashStore on both platforms.

For the second metric of ops/sec/dollar, FlashStore achieves comparable performance on both platforms and both traces. FlashStore's performance on this metric is higher than that of BerkeleyDB by about 1-2 orders of magnitude on the Blue platform and about 2x-6x on the Green platform. Thus, FlashStore wins on either platform but the choice of platform needs a tie-breaker based on other metrics.

5.9 Storage Deduplication Performance

The throughput in ops/sec obtained by FlashStore on the dedup trace can be converted to an average *throughput of the data stream* being backed up as follows. For the given ratio of unique chunks to total chunks in the dedup trace, we obtain an estimate of 1.44 operations per processed incoming chunk in the trace (1 `get` and 0.44 `set`s on average). Thus, the throughput of 57,271 ops/sec obtained by FlashStore on the Blue platform translates to 39,772 chunks processed per sec. With a chunk size of 4KB used to obtain the trace, this gives an incoming data throughput of 155 MB/sec. (For

a 1TB dataset, note that with 3GB of RAM and 32GB of flash, FlashStore can index all chunk hashes on flash without destaging to hard disk, hence we do not need to consider throughput reduction due to destaging here.)

This is comparable in value to the data transfer rates obtained by RAM prefetching mechanisms proposed in [37] for hard disk index based inline deduplication systems. The point we want to make here is that *a storage deduplication system using FlashStore as its index can achieve high data backup throughputs by mitigating hard disk seek bottlenecks on index lookups.*

Our design of FlashStore accommodates the RAM prefetching mechanisms for hard disk index based inline deduplication systems proposed in [37, 28]. We are currently building a storage deduplication system using FlashStore as the index for chunk hash lookups that also incorporates such RAM prefetching mechanisms. Hence, the performance numbers we report for FlashStore on the storage deduplication traces would be further enhanced when these optimizations customized for inline storage deduplication are incorporated.

5.10 Impact of Flash Recycling Activity

Finally, we evaluate the impact of flash recycling activity on FlashStore performance. We configured FlashStore to use amounts of flash equal to varying percentages of the dataset size, starting from 90% down to 25%. Because of flash recycling activity, we expect reductions in FlashStore performance as the usable flash capacity is reduced. We plot this effect in Figure 8 where we also overlay the throughput obtained by BerkeleyDB on hard disk for comparison purposes.

FlashStore shows a graceful degradation in throughput as flash capacity is reduced. This is not only because of flash recycling activity but also because lookups on destaged keys need to be served from hard disk (when the working set does not fit on flash). The performance the on Xbox trace drops less sharply because it has many key updates, hence recycled pages have many orphaned entries that are discarded. In contrast, the dedup trace has no key updates (after the first unique instance of a chunk appears, its associated key is set and never changed), hence recycled pages contain many valid keys that need to be destaged to hard disk, thus hitting disk bottlenecks. Note that even with flash reduced to 25% of dataset size, the throughput of FlashStore is several times that of BerkeleyDB on hard disk.

Storage deduplication index is an example of an application for FlashStore that can benefit from intelligent caching and prefetching strategies like those proposed in [37, 28]. Evaluating the impact of these strategies on boosting FlashStore performance for storage deduplication is the focus of future work.

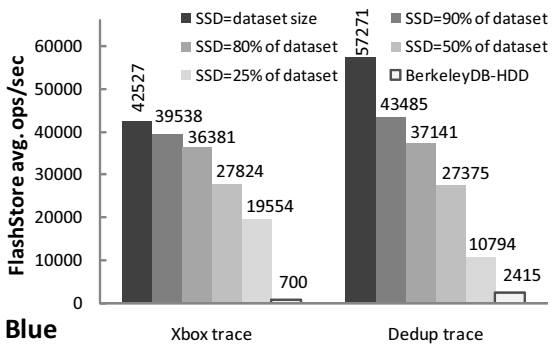

Figure 8: Impact of flash recycling activity on FlashStore performance and comparison with BerkeleyDB on hard disk on the two traces. The flash capacity in FlashStore is reduced as a percentage of the dataset size for each trace.

6. RELATED WORK

Flash memory has received lots of recent interest as a stable storage media that can overcome the access bottlenecks of hard disks. Researchers have considered modifying existing applications to improve performance on flash as well as providing operating system support for inserting flash as another layer in the storage hierarchy. In this section, we briefly review work that is related to FlashStore and point out its differentiating aspects.

MicroHash [36] designs a memory-constrained index structure for flash-based sensor devices with the goal of optimizing energy usage and minimizing memory footprint. This work does not target low latency operations as a design goal – in fact, a lookup operation may need to follow chains of index pages on flash to locate a key, hence involving multiple flash reads.

FlashDB [30] is a self-tuning B^+-tree based index that dynamically adapts to the mix of reads and writes in the workload. Like MicroHash, this design also targets memory and energy constrained sensor network devices. Because a B^+-tree needs to maintain partially filled leaf-level buckets on flash, appending of new keys to these buckets involves random writes, which is not an efficient flash operation. Hence, an adaptive mechanism is also provided to switch between disk and log-based modes. The system leverages the fact that key values in sensor applications have a small range and that at any given time, a small number of these leaf-level buckets are active. Minimizing latency is not an explicit design goal.

The benefits of using flash in a log-like manner have been exploited in FlashLogging [16] for synchronous logging. This system uses multiple inexpensive USB drives and achieves performance comparable to flash SSDs but with much lower price. Flashlogging assumes sequential workloads. In contrast, FlashStore works with arbitrary key access workloads.

Gordon [14] uses low power processors and flash memory to build fast power-efficient clusters for data-intensive applications. It uses a flash translation layer design tailored to data-intensive workloads. In contrast, FlashStore builds a persistent key-value store using existing flash devices (and their FTLs) with throughput maximization as the main design goal.

FAWN [12] uses an array of embedded processors equipped with small amounts of flash storage (but no hard disk) to build a power-efficient cluster architecture for data-intensive computing. Its design focuses mainly on read-mostly workloads. The differentiating aspects of FlashStore include the design goal of achieving high throughput on read-write mixed workloads in server-class applications, use of flash as a cache above hard disk to hold the working set of key-value pairs, and a specialized in-memory hash table structure with cuckoo hashing to minimize RAM usage per entry and achieve high load factors with bounded lookup

time. FAWN uses SSD as a replacement for hard disk. In contrast, FlashStore uses SSD as cache above hard disk and incorporates mechanisms for destaging as well as popularity measures for deciding which key-value pairs to destage to hard disk. Unlike FAWN, FlashStore eliminates the need for rehashing, a prohibitively expensive operation, by destaging key-value pairs to disk when hash table target load factors are reached.

BufferHash [11] builds a content addressable memory (CAM) system using flash storage for networking applications like WAN optimizers. It buffers key-value pairs in RAM, organized as a standard hash table, and flushes the hash table to flash when the buffer is full. Past copies of hash tables on flash are searched using a time series of Bloom filters maintained in RAM and searching keys on a given copy involve multiple flash reads. In contrast, FlashStore is designed to access any key in the current working set using one flash read, leveraging cuckoo hashing and compact key signatures to minimize RAM usage of a customized in-memory hash table index. Moreover, it uses flash as a cache on top of hard disk.

The work in [23] provides architectural support for using flash as an operating system managed disk cache and improves performance and reliability by splitting flash disk caches into separate read and write regions. This work uses *raw flash memory* and develops a programmable flash controller. FlashStore can be viewed as optimizing flash usage for a specific application, namely persistent key-value store, by using flash-aware data structures and algorithms on top of device FTL. By focusing on a single application and optimizing for it, we strive to squeeze out better performance than generic approaches at the operating system level.

eNVy [34] uses raw flash memory as the replacement for hard disk. It uses page level mapping to manage flash. It also does garbage collection and wear-leveling. FlashStore uses high-performance SSD as a cache above hard disk to build a high throughput key-value store. In ChunkStash, the flash management and wear-leveling is done by the flash device itself.

7. CONCLUSION

We designed FlashStore to be used as a high throughput persistent key-value storage layer for a broad range of server class applications. To this end, we used real-world data traces from two data center applications, namely, Xbox LIVE Primetime online multi-player game and inline storage deduplication, to drive and evaluate the design of FlashStore on traditional and low power server platforms. FlashStore outperforms BerkeleyDB (running on hard disk and flash separately) by 1-2 orders of magnitude on the metrics of throughput (ops/sec), energy efficiency (ops/Joule), and cost efficiency (ops/sec/dollar) for both applications running on both server platforms.

8. REFERENCES

[1] BerkeleyDB. `http://www.oracle.com/technology/products/berkeley-db/index.html`.

[2] BerkeleyDB for .NET. `http://sourceforge.net/projects/libdb-dotnet/`.

[3] C# `System.Threading`. `http://msdn.microsoft.com/en-us/library/system.threading.aspx`.

[4] Fusion-IO Drive Datasheet . `http://www.fusionio.com/PDFs/Data_Sheet_ioDrive_2.pdf`.

[5] Iometer. `http://www.iometer.org/`.

[6] MurmurHash Fuction. `http://en.wikipedia.org/wiki/MurmurHash`.

[7] Samsung SSD. `http://www.samsung.com/global/business/semiconductor/productInfo.do?fmly_id=161&partnum=MCCOE64G5MPP`.

[8] TPC: Transaction Processing Benchmark. `http://www.tpc.org/`.

[9] Xbox LIVE Primetime game. `http://www.xboxprimetime.com/`.

[10] N. Agrawal, V. Prabhakaran, T. Wobber, J. Davis, M. Manasse, and R. Panigrahy. Design Tradeoffs for SSD Performance. In *USENIX*, 2008.

[11] A. Anand, S. Kappes, A. Akella, and S. Nath. Building Cheap and Large CAMs Using BufferHash. *University of Wisconsin Madison Technical Report TR1651*, Feb 2009.

[12] D. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan. FAWN: A Fast Array of Wimpy Nodes. In *SOSP*, Oct. 2009.

[13] A. Broder and M. Mitzenmacher. Network Applications of Bloom Filters: A Survey. In *Internet Mathematics*, 2002.

[14] A. M. Caulfield, L. M. Grupp, and S. Swanson. Gordon: Using Flash Memory to Build Fast, Power-Efficient Clusters for Data-Intensive Applications. In *ASPLOS*, 2009.

[15] F. Chen, D. Koufaty, and X. Zhang. Understanding Intrinsic Characteristics and System Implications of Flash Memory Based Solid State Drives. In *ACM SIGMETRICS*, 2009.

[16] S. Chen. Flashlogging: Exploiting flash devices for synchronous logging performance. In *SIGMOD*, 2009.

[17] F. Corbato. A Paging Experiment with the Multics System. In *MIT Project MAC Report MAC-M-384*, 1968.

[18] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's Highly Available Key-value Store. In *SOSP*, 2007.

[19] E. Gal and S. Toledo. Algorithms and Data Structures for Flash Memories. In *ACM Computing Surveys*, volume 37, 2005.

[20] B. Gill, M. Ko, B. Debnath, and W. Belluomini. STOW: A Spatially and Temporally Optimized Write Caching Algorithm. In *USENIX*, 2009.

[21] A. Gupta, Y. Kim, and B. Urgaonkar. DFTL: a Flash Translation Layer Employing Demand-Based Selective Caching of Page-Level Address Mappings. In *ASPLOS*, 2009.

[22] A. Kawaguchi, S. Nishioka, and H. Motoda. A Flash-Memory Based File System. In *USENIX*, 1995.

[23] T. Kgil, D. Roberts, and T. Mudge. Improving NAND Flash Based Disk Caches. In *ISCA*, 2008.

[24] H. Kim and S. Ahn. BPLRU: A Buffer Management Scheme for Improving Random Writes in Flash Storage. In *FAST*, 2008.

[25] D. E. Knuth. *The Art of Computer Programming: Sorting and Searching (Volume 3)*. Addison-Wesley, Reading, MA, 1998.

[26] I. Koltsidas and S. Viglas. Flashing Up the Storage Layer. In *VLDB*, 2008.

[27] S. Lee, D. Park, T. Chung, D. Lee, S. Park, and H. Song. A Log Buffer-Based Flash Translation Layer Using Fully-Associate Sector Translation. In *ACM TECS*, volume 6, 2007.

[28] M. Lillibridge, K. Eshghi, D. Bhagwat, V. Deolalikar, G. Trezise, and P. Camble. Sparse Indexing: Large Scale, Inline Deduplication Using Sampling and Locality. In *FAST*, 2009.

[29] S. Nath and P. Gibbons. Online Maintenance of Very Large Random Samples on Flash Storage. In *VLDB*, 2008.

[30] S. Nath and A. Kansal. FlashDB: Dynamic Self-tuning Database for NAND Flash. In *IPSN*, 2007.

[31] R. Pagh and F. F. Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, May 2004.

[32] A. Silberschatz, P. B. Galvin, and G. Gagne. *Operating System Concepts*. John Wiley & Sons, 2008.

[33] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. In *SIGCOMM*, 2001.

[34] M. Wu and W. Zwaenepoel. envy: A non-volatile, main memory storage system. In *ASPLOS*, 1994.

[35] H. Yadava. *The Berkeley DB Book*. Apress, 2008.

[36] D. Zeinalipour-yazti, S. Lin, V. Kalogeraki, D. Gunopulos, and W. A. Najjar. Microhash: An Efficient Index Structure for Flash-based Sensor Devices. In *FAST*, 2005.

[37] B. Zhu, K. Li, and H. Patterson. Avoiding the Disk Bottleneck in the Data Domain Deduplication File System. In *FAST*, 2008.

[38] M. Zukowski, S. Heman, and P. Boncz. Architecture-Conscious Hashing. In *DAMON*, 2006.