Bankshot: Caching Slow Storage in Fast Non-Volatile Memory

Meenakshi Sundaram Bhaskaran Jian Xu Steven Swanson
Computer Science and Engineering
University of California, San Diego
{mbhaskar,jix024,swanson}@cs.ucsd.edu

ABSTRACT

Emerging non-volatile storage (e.g., Phase Change Memory, STT-RAM) allow access to persistent data at latencies an order of magnitude lower than SSDs. The density and price gap between NVMs and denser storage make NVM economically most suitable as a cache for larger, more conventional storage (i.e., NAND flash-based SSDs and disks). Existing storage caching architectures (even those that use fast flash-based SSDs) introduce significant software overhead that can obscure the performance benefits of faster memories. We propose Bankshot, a caching architecture that allows cache hits to bypass the OS (and the associated software overheads) entirely, while relying on the OS for heavy-weight operations like servicing misses and performing write backs. We evaluate several design decisions in Bankshot including different cache management policies and different levels of hardware, software support for tracking dirty data and maintaining meta-data. We find that with hardware support Bankshot can offer upto 5× speedup over conventional caching systems.

Categories and Subject Descriptors

D.4.2 [**Operating Systems**]: Storage Management

General Terms

Design, Performance

Keywords

Solid State Caching, Non-Volatile Memory, Storage Systems

1. INTRODUCTION

Emerging faster-than-flash non-volatile memories (NVMs) provide the potential to vastly increase IO system performance. However, their high cost (at least for now) means that in the near future these memories will likely find use as caches or tiering layers within larger storage systems comprised of hard drives and/or flash-based solid state drives (SSDs).

Most of the recent work on storage systems for fast NVMs such as phase change memory (PCM), spin-torque memories (STTM), and memristor [11, 12, 14, 5, 33] has focused on primary storage devices (i.e., stand-alone SSDs). At the same time, several groups have proposed systems that use flash-based SSDs as caches for conventional disks by either combining SSDs with cache-specific functions [29] with operating system support or by providing generic OS support for caching the contents of one block device on another [10, 27, 21, 30, 24].

Combining these two approaches to build a small cache out of fast NVMs should provide an economically feasible path to integrating fast NVMs into storage systems along with large performance benefits.

However, relying on the OS to access the cache and implement caching policy imposes software overheads and limits the performance improvements the cache can provide. For flash-based caches, these overheads are small compared to the SSD's access time, but for fast NVMs, the software overheads for cache hits can easily dominate total latency, dramatically reducing benefits of the caching layer.

This paper describes *Bankshot*, an SSD built to exploit the performance benefits of fast NVMs as a cache for slower, conventional block devices (e.g. flash-based SSDs and hard disks). Bankshot minimizes cache hit latency by allowing applications to access the cache hardware without operating system intervention. Operating system intervention is only required on cache misses, when an access to the slower, underlying storage device is inevitable. In addition, Bankshot provides hardware support to detect cache hits and misses, allow for recovery of data after power failure, collect data usage information for cache management policies and track dirty blocks for effecient write back.

We explore the design space of Bankshot by implementing caching functions (hit detection, metadata maintenance, and dirtiness tracking) in hardware, software, and a combination of the two. We find that on a wide range of storage access traces, a small amount of hardware support can provide significant reductions in both cache access time and cache miss rate. Interestingly, we find that hardware support for caching operations provides the greatest benefits for servicing misses.

The reminder of the paper is organized as follows. Section 2 provides motivation for reducing cache hit latency for emerging NVM. Section 3 and Section 4 describes the system and hardware, software components with detailed description of different design options. We present our evaluation of Bankshot in Section 5. Section 6 describes Bankshot with respect to previous work. Finally we present our conclusion in Section 7.

2. MOTIVATION

For the first time, faster-than-flash non-volatile memories are starting to become commercially available. Micron [2] sells PCM device in quantity, and Everspin [3] is sampling DDR3 DIMMs

INFLOW'13, November 3, 2013, Pennsylvania, USA.

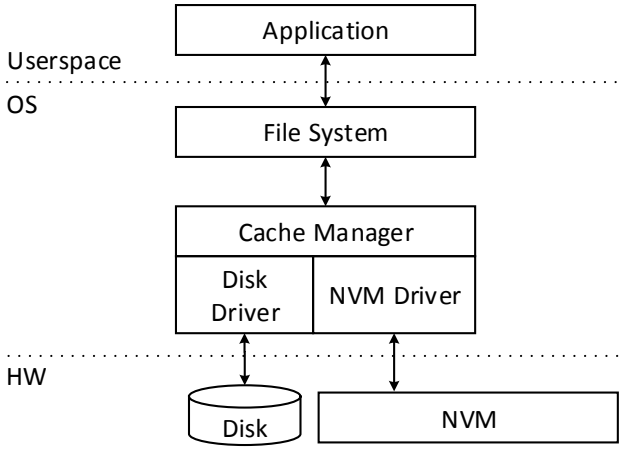

Figure 1: System Stack for SSD based caches: Depicts the system stack for conventional SSD caching system where the file system and operating system sets the policies for protection and sharing of data.

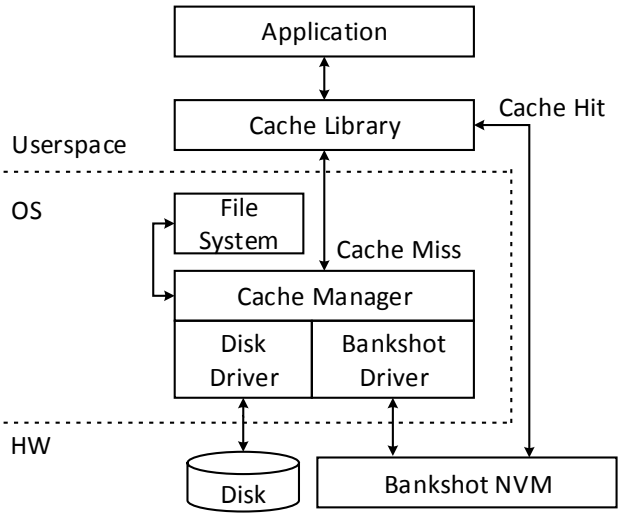

Figure 2: Bankshot system stack: User-space cache library services cache hits while Bankshot SSD provides primitives for protection and sharing.

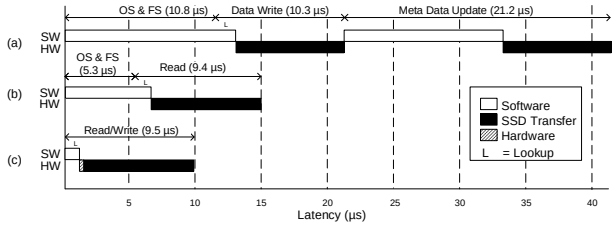

Figure 3: Breakdown of hit latency: Software and hardware components of the hit latencies for a 4 KB FlashCache Read (a) and FlashCache write (b) and Bankshot (c) read & write is shown when using a PCIe NVM emulating PCM with read and write latencies of 8.3 μs and 8.1 μs respectively. In FlashCache latencies introduced by the layers in the operating system contributes 6 μs for reads and 34 μs for writes.

populated with STTM. Also these memory technologies are not expected to replace main memory as they are order of magnitude slower than DRAM. But NVM provides a persistent store that is more scalable and consumes less idle power compared to DRAM. For now, atleast, these memory technologies are too expensive to serve as primary storage, but they can still play an important role in improving the performance of storage systems.

These memories are especially well-suited to serve as a persistent caching layer in front of conventional, slower storage (e.g., flash-based SSDs or hard disks). Recent study [7, 6, 23] of characteristics of large scale caching systems in a production environment show that most workloads have significant locality that a cache can exploit.

Existing SSD caching systems use an OS-resident cache manager (Figure 1) that exposes a generic block device interface and performs caching operations internally. The cache manager tracks dirty data and maintains a mapping between cache locations and the backing store. The cache manager is responsible for mapping cache contents while it relies on file system and operating system to implement protection and to mediate sharing of cached data.

This architecture leverages existing support for composable block devices in the kernel, but it also introduces extra software overheads and requires a operating system interaction on every access to either the cache or the backing store. These overheads are relatively small for systems that use lower-end SSDs as the caching device, but they are significant components for higher-performance SSDs. Figure 3 provides the latency overhead for servicing a 4 KB cache hit. For SATA SSDs used as cache, with typical latencies of 200 us and 1 ms for read and writes respectively, that software only increases latency by 3 %. However, high-end PCIe-based SSD [1, 31, 19] (with sub-100 μs latencies) and SSDs based on PCM or STTM technologies (with sub-10μs latencies [12, 33]) the software overhead will cripple cache performance.

The overheads are especially detrimental for cache hits, since those are the accesses that caches can speed up. By comparison, impact of software performance on cache misses is much smaller, since those accesses involve accesses to slow flash-based SSDs or disks.

With this in mind, Bankshot's primary design goal is to minimize hit latency, and the next section describes how Bankshot achieves that goal.

3. SYSTEM OVERVIEW

Bankshot addresses the overheads described above by leveraging an SSD built for fast NVMs and ensuring that caching software overheads only impact the latency of cache misses. Figure 2 depicts the architecture of the Bankshot system. Bankshot consist of three components. A user space library called *LibBankshot* that is responsible for servicing cache hits. A kernel driver called *Cache manager* to handle cache miss and interface with the file system. And a caching aware NVM called the *Bankshot SSD*.

Two aspects of the Bankshot design sets it apart from existing SSD caching layers. First, the application access the Bankshot hardware called directly from userspace via libBankshot. LibBankshot provides a POSIX compatible interface that intercepts file systems calls so that caching is transperent. Second, Bankshot operates on file extents rather than blocks. In particular, it leverages file system protection and file layout information to safely allow direct access from userspace and to detect cache misses in hardware.

Bankshot combines these two technologies with hardware sup-

port for tracking dirty data and mapping between data in the cache and data stored on backing store. The result is an SSD caching system with a unique set of properties:

- **Very fast cache hits** Since cache hits do not incur system call, operating system, or file system overheads, applications benefit fully from the short latencies of next-generation NVMs.
- **File system awareness** Bankshot caches file system extents rather than storage device blocks. This allows it to naturally prefetch contiguous regions of cached files and minimize the amount of metadata the cache must maintain. Despite this, Bankshot does not require any modifications to the file system.
- **Write-back caching** Tracking the mapping between cache contents and the backing store locations in hardware means that Bankshot is safe to use as a write-back cache.
- **Efficient cache eviction** The Bankshot SSD tracks which data in the cache is dirty and provides an efficient mechanism for querying that information.
- **Data usage tracking** The Bankshot SSD tracks data usage which the operating system leverages for LRU management policies.

Figure 2 depicts Bankshot's architecture. Bankshot comprises of the Bankshot SSD, the backing store (in this case a disk), a libBankshot, and the in-kernel cache manager.

To illustrate how these components work together, we describe the sequence of events that occurs for a miss without eviction, a hit, and miss requiring an eviction.

A miss to a cold cache First, we describe a miss that requires the system to load new data into the cache, but does not require an eviction. First, libBankshot intercepts a read() system call and checks whether it is a hit or a miss. LibBankshot maintains a per-process copy of the cache's layout for this purpose. This layout, or *logical-map*, maintains translation information for file offsets to the locations of those extents in the Bankshot SSD. LibBankshot uses the logical-map to look up the extent that the access targets and to determine whether it is present in the cache.

If the access is a miss libBankshot issues a system call to the cache manager. The manager queries the file system for application's file access permission and the file extents. Once verified it allocates the space for the requested extent, copies the data from the backing store to the Bankshot SSD, and installs a permission record (described below) in the SSD to allow the application to access the extent. It also records the reverse mapping (from cached extent to the backing store extent location) in the SSD. It then completes the system call by returning the location to libBankshot which can reissue the request (now a hit), and inform the applications that the access is complete.

A hit On a hit, libBankshot will find the mapping from the target file extent to a location in the logical-map, and issue a I/O to the SSD. To support user-space access, Bankshot builds upon a direct-access SSD architecture called Moneta-D [12]. The direct-access mechanism gives each application a virtual "channel" that allows it to issue IO commands and be notified of their completion.

The Bankshot hardware stores a per-channel permission table that describes the regions of the SSD that each channel can access. The operating system loads the channel permissions in the hardware when servicing the miss. If a channel tries to access data without proper permissions in place, the access will fail.

For write hits, the Bankshot SSD can automatically mark the extent as dirty. We explore the performance impact of this hardware support below.

A eviction and a miss If a miss occurs and the cache is full, Bankshot must evict an extent and, if needed, write back modified data to the backing store. The cache manager handles this process.

The first step is to identify one or more "victim" extents to evict from the cache to make room. We will assume only one victim is required. The first step in the eviction process is to remove any permission entries for that victim extent from Bankshot's hardware permission table. This effectively revokes the applications' access to the extent, causing accesses to the extent to fail and ensuring that only cache manager will be reading or writing the extent during the eviction process.

Next the cache manager must determine whether the data in the extent is dirty. If it is, it must write the data back to the backing store. Finally, the cache manager will load data for the new extent into the cache and install the corresponding permission entry.

We explore cache eviction policies, mechanisms for identifying dirty data, and mechanisms for maintaining persistent information about cache layout in Section 4.

4. DESIGN SPACE

We explore several design options for Bankshot that all rely on the same underlying, baseline architecture. This section describes the hardware and software components of a baseline Bankshot implementation and then describes the extensions in hardware and software that make trade-offs between complexity, the need for custom hardware support, and performance. In particular, we discuss options for tracking dirty data, maintaining cache metadata, tracking cache hits and maintaining data-structure for LRU.

4.1 Bankshot Hardware

The Bankshot hardware, Bankshot SSD, provides user space applications with direct access to an NVM-based SSD. Figure 4 shows the components of the Bankshot SSD. These implement a storage-like interface that applications can access directly. It supports read and write operations from/to arbitrary locations and of arbitrary sizes (i.e., accesses need not be aligned or block-sized).

The hardware includes the host-facing PIO and DMA interfaces, the request queues, the request scoreboard, and internal buffers. These components are responsible for accepting IO requests, executing them, and notifying the host when they are complete. It also includes the permission check mechanism that prevents applications from accessing data that the kernel has not granted them access to. Communication with the host occurs over a PCIe 1.1×8 interface, which runs at 2 GB/s, full-duplex.

The banks of non-volatile memory are at right in Figure 4. The SSD contains eight high-performance, low-latency non-volatile memory controllers attached to an internal ring network. The SSD's local storage address space is striped across these controllers with a stripe size of 8 KB.

The raw hardware can perform a 4 KB read or write access in 8.32/8.18 μs respectively and sustain 0.65 M random 4 KB IOPs.

4.2 Bankshot Software

The software in Bankshot consist of two components, the kernel-resident cache manager and the user space library, libBankshot, as shown in Figure 2.

4.2.1 Cache Manager

The cache manager provides two functions. First, it combines Bankshot SSD and another block device (the backing store) into

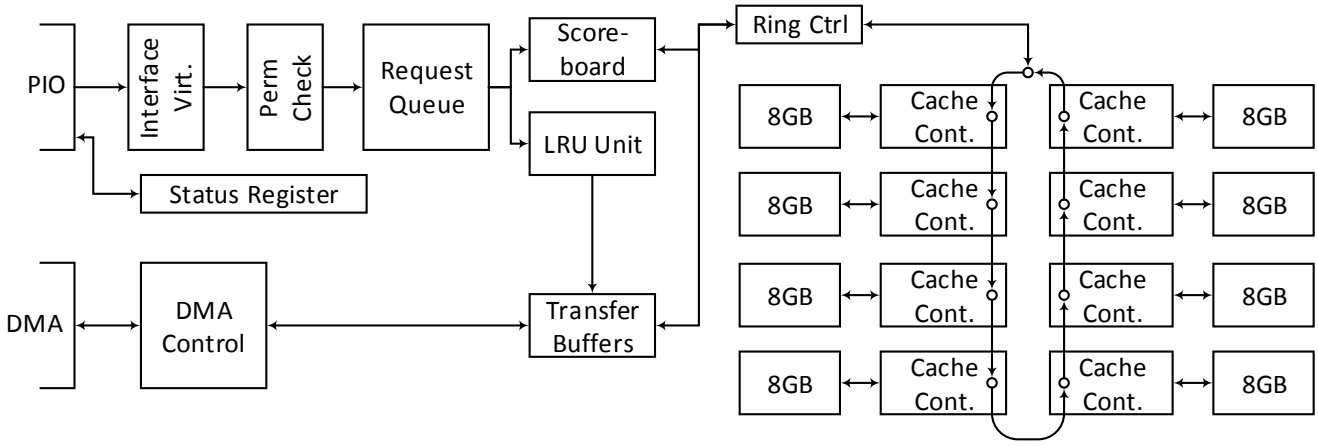

Figure 4: Bankshot SSD: Each memory controller has a cache logic that manages cache meta data and dirty bits. Cache LRU unit is responsible for tracking application access pattern to assist the kernel with eviction policy.

a single, reliable, cached block device. Second, it manages the Bankshot SSD's virtualized interface and the permission tables for each application.

To provide a cached block device, the cache manager divides the storage in Bankshot SSD into 4 KB blocks and uses a fully associative map for management of the cache contents. It uses a B+tree [15] of 8 B key and 4 B cache block number to store the *physical-map*. The physical-map maintains the mapping between backing store extents and the location of the extent in cache. Associated with each cache block is a 24 B in-memory meta-data that contains the state information for each cached extent. A reader/writer lock protects access to the B+tree while meta-data updates use atomic operations to provide efficient parallel access. If an I/O request spans multiple extents or only a subset of the request resides in the cache, the cache manager breaks up the request appropriately.

Bankshot caches file extents rather than disk blocks for three reasons. First, Bankshot must be aware of file system permissions and modern file systems maintain those permissions for extents, not blocks. Second, managing extents makes it easy to prefetch file data and ensures that write backs can result in long, sequential accesses. Third, extent-based mappings reduce the size of the maps that libBankshot and cache manager must maintain, improving performance and reducing memory requirements.

The cache manager in Bankshot is responsible for performing cache lookups, managing hardware permissions, evicting/allocating cache extents and recovering cache data on crash.

Cache Lookup LibBankshot invokes the cache manager only on a cache miss. There are two types of misses: data misses and permission misses. For both types of misses, the cache manager first validates file descriptor permissions with the file system. It then retrieves the file extents to backing store extents information using the file system FIEMAP ioctl(). Using the backing store extent it lookups up in the physical-map to determine if any portion of the extent exist in Bankshot SSD and handles the request as a data or protection miss as described previously.

Permission Management The permission table in Bankshot SSD can only hold 8192 entries. If Bankshot used one permission entry per extent, the system could quickly run out of permission table

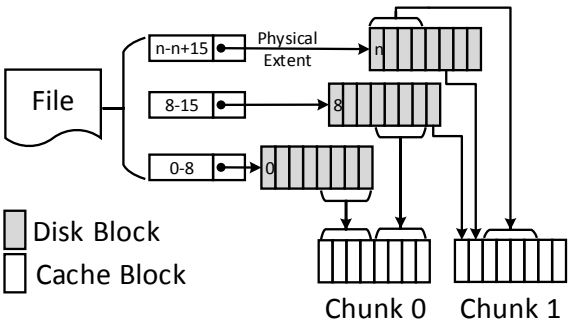

Figure 5: Permission Management: chunks map extents from the same file in Bankshot

entries, leading to unnecessary misses that would occur when a request to the hardware fails, even though the data is present.

To avoid this problem, Bankshot divides the cache into *chunks* such that there is one chunk per permission entry, eliminating the possibility of unnecessary misses. In our prototype implementation, each chunk is 2 MB.

Each chunk can contain multiple extents (not necessarily ordered) from a single file, and a single file (or even a single extent) can occupy multiple chunks as shown in Figure 5. Bankshot evictions occur at the granularity of chunks, so a single eviction may require multiple write-back operations, one for each extent in the chunk.

Cache Eviction In Bankshot evictions happen at chunk granularity. By removing all the cached data in a chunk Bankshot prevents untrusted user space applications from modifying a files mapped previously by the chunk. On a capacity miss, the cache manager identifies a candidate chunk and evicts the data by a multi-step process. First, the cache manager marks the state of the cache blocks as EVICTION_IN_PROGRESS to prevent servicing request to in-flight cache regions until the data is copied out to the backing store. Second, it removes all the permission entries for the chunk in the hardware to prevent libBankshot from servicing cache hits. Third it identifies dirtiness information for the chunk and writes back the dirty blocks. Finally, it clears the EVICTION_IN_PROGRESS flag and marks the state of evicted blocks as INVALID in the meta-data

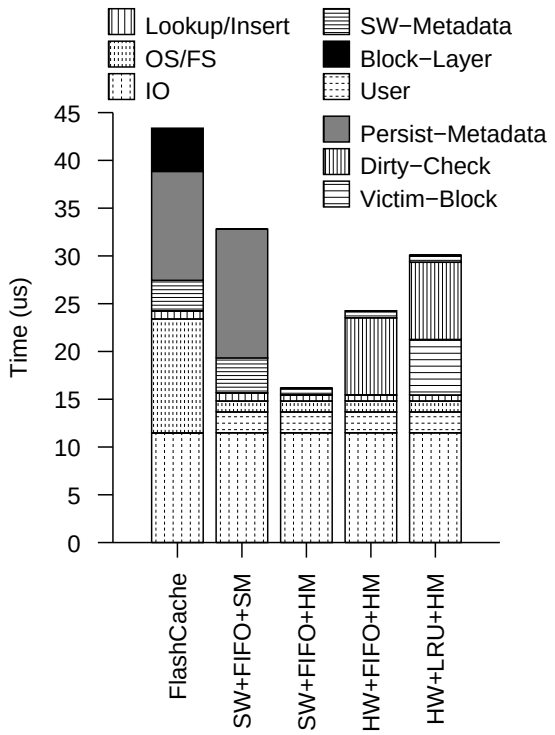

Figure 6: Latency breakdown for cache miss: Hardware optimization for metadata reduces the latency for a cache miss by 20 μs compared to FlashCache. Hardware dirty bit support and LRU optimizations to improve cache hit rate add 14 μs in total for identifying victim chunk for eviction. Dirty blocks require disk write back which is in the order of milliseconds

table and deletes the keys from the physical-map. Bankshot optimizes the write-back by first sorting the extents within a chunk based on backing store address and then merging extents that are contigious both in backing store and the Bankshot SSD.

Cache Recovery Bankshot stores the 64-bit backing store address associated with each cache block persistently in the NVM. When the cache manager loads, it scans the meta-data in the Bankshot SSD to reconstruct the in-memory physical-map (i.e B+Tree). In case of system crash, the cache manager on initalization writes all the dirty data in the the Bankshot SSD to the backing store, thereby recovering the cached data.

4.2.2 LibBankshot

LibBankshot transparently intercepts IO system calls (e.g., open, close, read, and write) made by unmodified application using LD_PRELOAD. LibBankshot is responsible for servicing all cache hits in Bankshot, registering applications with the cache manager and requesting cache manager to handle cache misses. It maintains a mapping from file offsets to cache locations for data that the application has accessed, eliminating the need to invoke file system to locate data on cache hits.

4.3 Design options

The baseline design described above provides the core caching functionality that Bankshot requires, but there are many aspects of caching that might benefit from hardware support in the SSD. This section describes implementation options for cache replacement policies, dirty data tracking, and meta data management. The next section evaluates the performance benefits of each option.

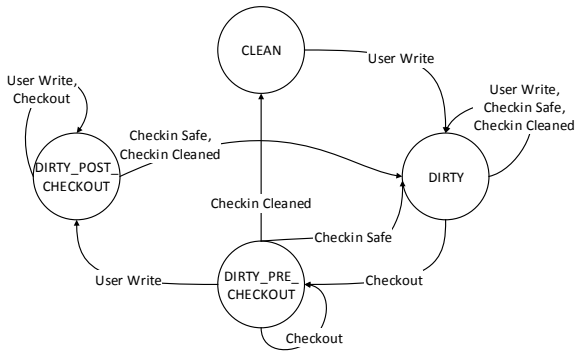

Figure 7: Hardware Dirty Tracking: Each cache logic stores 2-bits per 4 KB block for dirty information tracking.

4.3.1 Replacement Policies

Cache replacement policies affect the cache hit ratio and hence the performance of a cache. Bankshot implements two replacement policies, FIFO and LRU.

In a FIFO replacement policy the cache manager evicts chunks in the order of allocation. FIFO eviction is easy to implement, but it can cause the cache manager to evict chunks even if they are under intensive use. However, in Bankshot FIFO policy is the only deterministic replacement policy that the cache manager can implement purely in software. Even though libBankshot is aware of per process data access pattern, gathering access information from userspace is unsafe. Implementing usage-aware caching policies in software is challenging since the cache manager is unaware of accesses that the application makes to the SSD. To overcome this limitation, the SSD needs to collect information about access patterns on behalf of the cache manager.

With some hardware support Bankshot can implement an LRU replacement policy. It maintains a doubly-linked list of chunks in LRU order, and updates the list on every access. The updates occurs off the critical path of normal accesses. The Bankshot SSD exposes a DEQUEUE command to return the least recently used chunk off the list. Querying of the LRU information happens only a cache miss and each query to the hardware incurs only 5 μs of latency (Figure 6) compared to the several ms it takes to move data to and from backing store.

4.3.2 Tracking Dirty Data

Since Bankshot is a write-back cache, it must know when data in the cache has changed. Tracking dirty data presents a problem similar to tracking LRU information: Since applications access the Bankshot SSD directly, the cache manager is, by default, unaware of writes to cached data. There are two options: First, we can use the Bankshot SSD's permission mechanism to detect the initial write to an extent. Second, we can add hardware support to the Bankshot SSD to track dirty data directly.

Tracking dirty data in software Each cache block in a chunk has a bit in the cache manager that indicates the dirtiness of the block. Cache manager sets the bit as clean or dirty based on the type of miss serviced. When allocating cache blocks to service read misses the cache manager first inserts only read permission in the hardware. On the first write, the hardware denies the modify request to the chunk. This permission failure forces libBankshot to request the cache manager to service a write permission miss. The cache

manager then sets the block as dirty and inserts write permission to the Bankshot SSD permission table.

Since Bankshot manages hardware permissions on a per-chunk basis, the cache manager assumes the entire chunk as dirty and will write back all the cached blocks in the chunk on eviction.

Tracking dirty data in hardware Adding hardware support for tracking dirty data can eliminate needless write backs and avoid the need for an extra permission update for the first write to a chunk.

The Bankshot SSD stores two persistent dirty bits for every 4 KB cache block. On every write from the application the SSD updates the dirty bits. During an eviction, the cache manager queries the dirty bits to determine which blocks it needs to write back.

The cache manager can proactively "clean" the cache by writing back dirty data. In this case, the cache manager must query the dirty bits, write back the dirty data, and then clear the bits. A race condition can occur if the application modifies the data between the write back operation and clearing the dirty bits. The Bankshot SSD uses two dirty bits per cache block to allow concurrent cleaning of cache.

Figure 7 shows how four dirtiness states can eliminate the race condition. Rather than just querying the dirty bits, the cache manager issues a CHECK_OUT for the dirty bits in a chunk. Th SSD moves the dirty blocks from DIRTY to DIRTY_PRE_CHECKOUT state. A write to a checked out block marks the block as DIRTY_POST_CHECKOUT. Once the cleaning operation is complete, the cache manager issues a CHECK_IN command. The check in clears dirty pages to CLEAN and leaves DIRTY_POST_CHECKOUT pages as DIRTY.

Unlike software dirty bit, hardware dirty bit allows applications to modify data in cache without taking a cache miss. However with hardware tracking, eviction of chunks would require the cache manager to issue IO to the Bankshot SSD to query the dirty bit information. As seen in Figure 6 hardware dirty bit incurs only 8μs of latency overhead to query the dirty information of all the blocks in a chunk while the backing store write back takes several ms.

Hybrid Hardware/Software Dirty Bits We can combine the hardware and software dirty data tracking mechanisms to provide the best of both world. The cache manager can rely on the software mechanism to determine when a chunk is not clean, and then use the hardware dirty bits to determine which blocks within the chunk are actually dirty.

4.3.3 Meta Data Management

Caches track the backing store address to cache address translation information using per block meta data. Write back caches that provide durability guarantee persist the meta data to survive system crashes. Flash SSD based caching systems delay the meta-data updates until a write changes the state of cache block from clean to dirty. In Bankshot as applications can modify the data directly, the cache manager needs to persist the meta-data on every cache allocation. B+Tree index and meta-data table in the host memory track cache contents and in order to survive crashes the cache manager implements two different schemes to write the meta-data table to hardware.

Software Meta-data On each cache allocation, the cache manager first performs necessary evictions and fills the cache with valid data. It then writes a 64 bit backing store address record for each cache block in a extent to a mapping table stored in the Bankshot SSD.

Hardware Meta-data The Bankshot SSD can record the mapping information on its own and guarantee that both data and meta data update occur atomically. Cache manager on allocation stores the data and 64-bit backing store address using a special CACHE_FILL_WRITE command. The command updates the 64-bit address in a mapping table stored in dedicated portion of the Bankshot SSD's persistent memory. The command does not complete until both the data and meta-data updates have completed, ensuring atomicity. We envisage the Bankshot SSD to have small supercap that provides sufficient backup power to FPGA and DRAM to write upto 64 KB of data (8 KB for each memory controller).

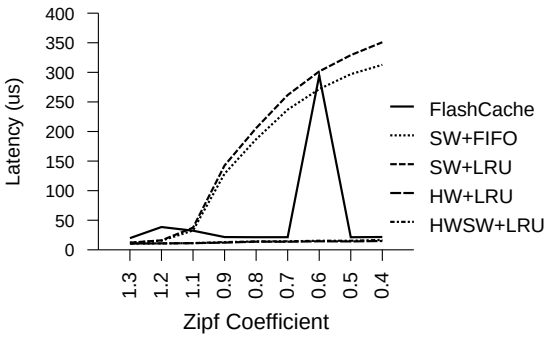

Figure 8: Bankshot Average Write Latency

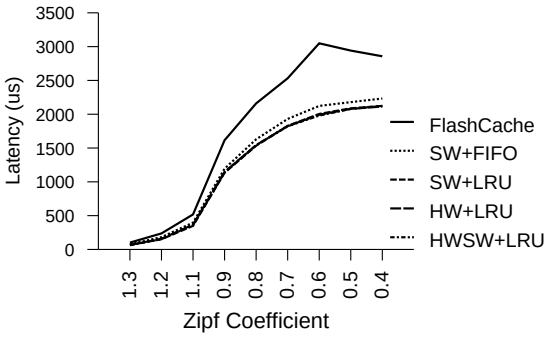

Figure 9: Bankshot Average Read Latency

5. RESULTS

Bankshot reduces the software overhead in servicing cache hits using libBankshot and the Bankshot SSD. In this section we measure the performance of Bankshot using microbenchmarks and block traces and evaluate the different design options discussed previously.

5.1 Experimental setup

We implemented the Bankshot SSD on the BEE3 prototyping platform [9]. The BEE3 provides four FPGAs which each host 16 GB of DDR2 DRAM and one PCIe interface. The design runs at 250 MHz. To emulate the performance of PCM using DRAM, we use a modified DRAM controller that allows us to set the read and write latency. We use the latencies from [22] âĂŤ 48 ns and 150 ns for array reads and writes, respectively.

We ran all our experiments on an 16-core 2.9GHz Intel Xeon X5647 equipped with a 250GB 7200RPM Seagate hard drives as the backing store. We compare Bankshot against Facebook's FlashCache [30].

FlashCache is an open source cache driver for Linux. It utilizes the Linux kernel device mapper functionality to register an SSD as a write back or write through cache for disk. FlashCache divides the cache address space into sets of 512-4 KB blocks and uses a hash table with linear probing within sets for management.

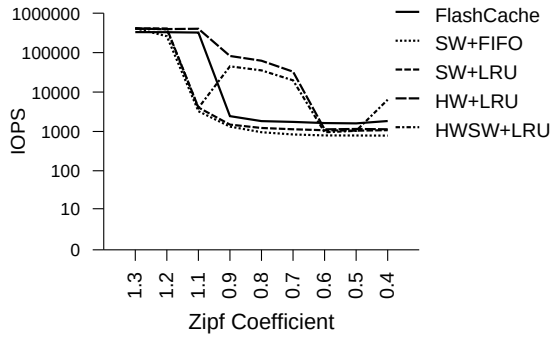

Figure 10: Bankshot Read Bandwidth

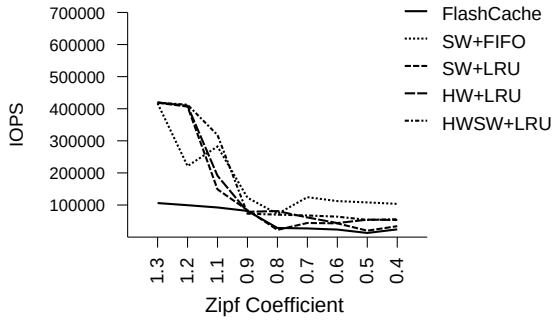

Figure 11: Bankshot Write Bandwidth

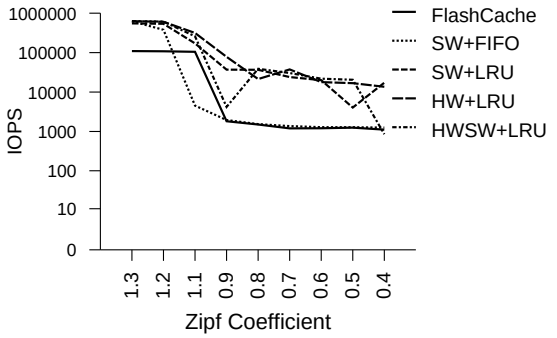

Figure 12: Bankshot Read/Write Bandwidth

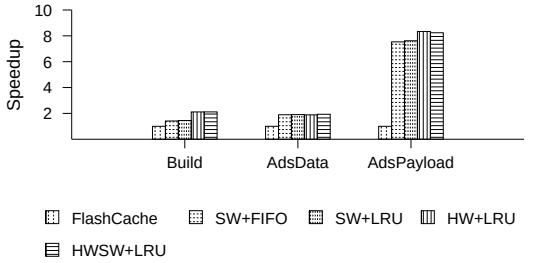

Figure 13: Bankshot Traces Workload

Associated with each block is 24 B meta data stored in the SSD. FlashCache also periodically scans the meta-data table and cleans blocks within a set when the dirty block threshold of the set exceeds a limit.

To keep runtime manageable, we limit the cache size to 16 GB in our experiments. We use the XFS file system in all cases.

5.2 Microbenchmarks

In order to understand the read write performance of Bankshot for different cache hit ratios we use Flexible IO Tester, FIO [17]. Fio can issue random requests according to a Zipf distribution [4]. The Zipf distribution takes a parameter (the "Zipf coefficient") that increases spatial locality by focusing accesses on a small region of the backing store. Higher Zipf coefficients correspond to more spatial locality and represent higher hit rates. We vary the Zipf coefficient from 1.3 to 0.4 and measure the cache latency and bandwidth.

To measure cache latency we first warm the cache with random data from a 24 GB file. We then run FIO for 20 minutes to measure the steady state performance of the cache. We report the average by running the experiment 3 times. To keep runtime manageable, we limit the Bankshot SSD to 16 GB in our micro-benchmark experiments. We use the XFS file system in all test cases.

Latency Figure 8 shows the average write latency and Figure 9 shows the average read latency for a 4 KB IO. For a zipf coefficient of 1.3 (98.3% hit ratio) Bankshot reduces the write hit latency from 19 μs to 10 μs. Unlike FlashCache which suffers from both capacity and conflict misses, Bankshot's fully associative mapping eliminates conflict misses to provide a consistent write latency. Hardware support for dirty bit improves the cache latency by avoiding redundant write backs for smaller zipf coefficients (or lower hit rates). Further as seen from Figure 9, direct user-space access reduces the cache hit latency for reads from an average of 100 μs to 62 μs. The sharp increase in access latency occurs as the hit rate drop from 82% to 45%.

Figure 3 shows the breakdown of software and hardware components for a 4 KB hit in FlashCache and Bankshot. Userspace access to cache reduces the overheads for read hits from 14 μs to 9 μs while for write hits from 42 μs to 9 μs.

Bandwidth Figure 10, 11, 12 shows the read, write and mixed (50% read and 50% write) performance of Bankshot and FlashCache. With hardware support for LRU and dirty bit Bankshot achieves 4$\times$ performance improvement over FlashCache for writes and 5$\times$ performance improvement for mixed workloads. Eliminating the operating system and file system overheads is beneficial as it reduces the contention for spinlocks within the kernel that are especially expensive for writes. LibBankshot reduces the lock contention by having threads aquire only reader locks to service cache hits.

5.3 Application Workload

To study the application performance of Bankshot we use three traces collected from Microsoft Production servers [20]. Table 1 describes the characteristics of these three traces. "Build" is a trace collected from server used for building windows operating system in production environment. "AdsData" and "AdsPayload" are backend servers providing storage for user id caching and ads display data respectively. All the traces contain information about the process, thread ID, completion and request time for each IO. As seen in [20], these traces have a average system interarrival time to average disk interarrival time ratio as 1, i.e. the disk I/O completion time limits the system I/O request rate. The average IO size is different for the three traces allowing us to evaluate extent based caching. Also the traces exhibit different read to write distribution.

In order to replay the traces we use FIO in replayer mode. As the IO completion time limits request rate, we use multiple threads with each thread issuing a unique IO from the trace. We analyzed the traces and identified the average number of processes in each 15-minute interval of the trace to determine the number of replay

Trace	Reads (M)	Writes (M)	Avg. Read Size (KB)	Avg. Write Size (KB)	Replay threads
Build	6.02	5.8	15.71	11.53	32
AdsData	1.3	0.14	29.73	5.73	12
AdsPayload	0.6	0.47	60.85	8.68	9

Table 1: Server traces:Traces collected from Microsoft production servers for 24-hour period. The ratio of average system iterarrival time to average disk interarrival time is 1 for all the three traces.

threads. Table 1 provides the number of replay threads used for each trace.

As seen from Figure 13 Bankshot achieves harmonic mean speedup of 2.6× over FlashCache for different application traces. For AdsPayload server a software only version of Bankshot achieves nearly 7.5× performance improvement over flashcache by reducing the software overhead. Also file extents based caching contributes to the increase in performance by allowing the cache manager to perform sequential write backs to disk. Hardware dirty bit support enables the cache manager to intelligently write back only those blocks that are dirty in a chunk which further improves the performance of the system by 10% over software only version.

All of the Bankshot configurations exhibit nearly 2x speedup over FlashCache for AdsData server. The speedup is uniform for different schemes because of two reasons. First, it is a read intensive workload (9:1) which causes the permission miss overhead in software only scheme to account for only 0.03% of the disk access time. Second, the working set size of the trace is around 4 GB resulting in no cache evictions.

Similarliy for Build server traces software only version of Bankshot achieves 1.5x speedup over FlashCache. But adding hardware support increased the performance by nearly 50% over software scheme.

6. RELATED WORK

Flash based SSDs Caches Many storage vendors provide provide caching solutions [18, 32, 26] that utilize PCIe Flash based SSDs as write-back or write through cache. Cache managers for SSDs are also available in the open source community [30, 8]. Recently Linux 3.9 included the support for using SSDs as caches for disk using the device mapper framework. However all these systems introduce significant software overhead to service a cache hit.

In addition, previous flash-based write-back caches [30, 8] suffer from two drawbacks. First, caching systems that use flash based SSDs implement complex schemes to coalesce meta data information across multiple cache blocks to leverage the page write property of NAND. These management schemes result in varying cache latencies. With emerging NVMs that provide a byte addressable in-place update interface and several orders of magnitude better endurance than NAND flash, the complex schemes are software overheads that contribute to nearly 37% of NVM latency (Figure 6). Secondly, delayed and discrete IO for data and meta data introduce inconsistencies between cache contents and cache metadata [30]. Bankshot's specialized hardware eliminates these problems.

Specialized Caching interface Bankshot design draws and extends from previous work in providing specialized interface to SSDs as storage and caches. Marvell has ships DragonFly PCIe SSDs with caching specific logic in the controller [25]. FlashTier [29] proposes a specialized interface for caching specific services in flash based SSDs. FlashTier reduces the multiple of levels of address translation by combining cache address map and the FTL of the SSD. It provides a consistent and durable cache through specialized SSDs commands and utilize the garbage collector within the SSD to perform silent eviction of clean data. While Bankshot extends the specialized interface it differs in two ways.

First, Bankshot explores the interface required to use emerging non-volatile memories such as PCM and STT-RAM as caches that do not suffer idiosyncrasies for NAND flash reads and writes. Recent work [28] has show that these memory technology do not require complicated FTL for management. Further they provide a byte-addressable memory with in-place updates eliminating the need for grabage collection.

Second, Bankshot implements all cache address space management functionality in the operating system. Unlike FlashTier, Bankshot utilizes the host CPU and DRAM to perform lookups and inserts. This allows Bankshot to leverage the file level information available in the kernel and simple LRU logic in the hardware to make better eviction decisions.

Bankshot extends the virtualized interface from Moneta-D [13]. Similar approaches for hardware permission and user space library have been explored in distributed, networked file systems. Direct Access File System (DAFS) [16] moves operating system functionality to user-space by using specialized NIC with virtualized interface (VI) to support userspace Remote Direct Memory access (RDMA). DAFS achieves low latency access by reducing memory copies. While DAFS uses a user-space library on the client side to transfer data, it relies on a kernel driver in the server side for RDMA. Further, unlike Bankshot which verifies each request in the hardware, DAFS implicitly trusts the client library request.

7. CONCLUSION

In this paper we describe Bankshot a caching system for emerging NVMs. Bankshot reduces the operating system and file system overhead incurred while servicing cache hits by allowing applications directly modify the data in the cache. Hardware support for dirtiness and metadata tracking reduces the cache miss latency significantly. With minimal support from hardware Bankshot achieves upto 3× performance improvement over a wide variety of workloads.

8. REFERENCES

[1] Fusion-io ioDrive2. http://www.fusionio.com/products/iodrive2.

[2] Phase Change Memory - Micron Technology, Inc. http://www.micron.com/products/multichip-packages/pcm-based-mcp.

[3] Spin Torque MRAM - Everspin Technologies, Inc. http://www.everspin.com/spinTorqueMRAM.php.

[4] L. A. Adamic and B. A. Huberman. Zipf's law and the Internet. *Glottometrics*, 3:143–150, 2002.

[5] A. Akel, A. M. Caulfield, T. I. Mollov, R. K. Gupta, and S. Swanson. Onyx: a protoype phase change memory storage array. In *Proceedings of the 3rd USENIX conference on Hot topics in storage and file systems*, HotStorage'11, pages 2–2, Berkeley, CA, USA, 2011. USENIX Association.

[6] D. Arteaga, M. Zhao, P. V. Riezen, and L. Zwart. A trace-driven analysis of solid-state caching in cloud computing systems. http://www.cloudvps.com.

[7] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, pages 53–64, New York, NY, USA, 2012. ACM.

[8] Bcache. http://bcache.evilpiepirate.org/.

[9] BEE Cube. http://beecube.com/products/.

[10] S. Byan, J. Lentini, A. Madan, L. Pabon, M. Condict, J. Kimmel, S. Kleiman, C. Small, and M. Storer. Mercury: Host-side flash caching for the data center. In *Mass Storage Systems and Technologies (MSST), 2012 IEEE 28th Symposium on*, pages 1–12, 2012.

[11] A. M. Caulfield, A. De, J. Coburn, T. I. Mollov, R. K. Gupta, and S. Swanson. Moneta: A high-performance storage array architecture for next-generation, non-volatile memories. In *Proceedings of the 43nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 43, pages 385–395, New York, NY, USA, 2010. ACM.

[12] A. M. Caulfield, T. I. Mollov, L. A. Eisner, A. De, J. Coburn, and S. Swanson. Providing safe, user space access to fast, solid state disks. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 387–400, New York, NY, USA, 2012. ACM.

[13] A. M. Caulfield, T. I. Mollov, L. A. Eisner, A. De, J. Coburn, and S. Swanson. Providing safe, user space access to fast, solid state disks. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '12, pages 387–400, New York, NY, USA, 2012. ACM.

[14] A. M. Caulfield and S. Swanson. Quicksan: A storage area network for fast, distributed, solid state disks. In *ISCA '13: Proceedings of the 40th annual international symposium on Computer architecture*, 2013.

[15] D. Comer. Ubiquitous b-tree. *ACM Comput. Surv.*, 11(2):121–137, June 1979.

[16] M. DeBergalis, P. Corbett, S. Kleiman, A. Lent, D. Noveck, T. Talpey, and M. Wittle. The Direct Access File System. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, FAST '03, pages 175–188, Berkeley, CA, USA, 2003. USENIX Association.

[17] Flexible I/O Tester. http://freecode.com/projects/fio.

[18] Fusion-io directCache. http://www.fusionio.com/data-sheets/directcache/.

[19] http://www.intel.com/content/www/us/en/solid-state-drives/ssd-910-series-specification.html.

[20] S. Kavalanekar, B. Worthington, Q. Zhang, and V. Sharda. Characterization of storage workload traces from production windows servers. In *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*, pages 119–128, 2008.

[21] T. Kgil and T. Mudge. Flashcache: a nand flash memory file cache for low power web servers. In *Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, CASES '06, pages 103–112, New York, NY, USA, 2006. ACM.

[22] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger. Architecting phase change memory as a scalable dram alternative. In *ISCA '09: Proceedings of the 36th annual international symposium on Computer architecture*, pages 2–13, New York, NY, USA, 2009. ACM.

[23] A. W. Leung, S. Pasupathy, G. Goodson, and E. L. Miller. Measurement and analysis of large-scale network file system workloads. In *USENIX 2008 Annual Technical Conference on Annual Technical Conference*, ATC'08, pages 213–226, Berkeley, CA, USA, 2008. USENIX Association.

[24] Z. Liu, B. Wang, P. Carpenter, D. Li, J. S. Vetter, and W. Yu. Pcm-based durable write cache for fast disk i/o. In *Proceedings of the 2012 IEEE 20th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, MASCOTS '12, pages 451–458, Washington, DC, USA, 2012. IEEE Computer Society.

[25] M. T. G. Ltd. Drangonfly virtual storage accelerator, 2011. http://www.marvell.com/storage/dragonfly/.

[26] Netapp flash cache. http://www.netapp.com/us/products/storage-systems/flash-cache/index.aspx.

[27] T. Pritchett and M. Thottethodi. Sievestore: a highly-selective, ensemble-level disk cache for cost-performance. In *Proceedings of the 37th annual international symposium on Computer architecture*, ISCA '10, pages 163–174, New York, NY, USA, 2010. ACM.

[28] M. K. Qureshi, J. Karidis, M. Franceschini, V. Srinivasan, L. Lastras, and B. Abali. Enhancing lifetime and security of pcm-based main memory with start-gap wear leveling. In *MICRO 42: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 14–23, New York, NY, USA, 2009. ACM.

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

[30] M. Srinivasan. Flashcache : A Write Back Block Cache for Linux. https://github.com/facebook/flashcache.

[31] Violin memory 6000 series flash memory arrays. http://www.violin-memory.com/products/6000-flash-memory-array/.

[32] Flashmax connect. http://www.virident.com/products/flashmax-connect/.

[33] J. Yang, D. B. Minturn, and F. Hady. When poll is better than interrupt. In *Proceedings of the 10th USENIX conference on File and Storage Technologies*, FAST'12, pages 3–3, Berkeley, CA, USA, 2012. USENIX Association.