Mjölnir: Collecting Trash in a Demanding New World

Zev Weiss
University of Wisconsin-Madison

Sriram Subramanian
Swaminathan Sundararaman
Vinay Sridhar
SanDisk, Inc.

Nisha Talagala *
Parallel Machines

Andrea C. Arpaci-Dusseau Remzi H. Arpaci-Dusseau
University of Wisconsin-Madison

Abstract

As flash devices become ubiquitous in data centers and cost per gigabyte drops, flash systems need to provide data services similar to those of traditional storage. We present Mjölnir, a powerful and scalable engine that addresses the core problems that make efficient flash based data services challenging: multi-reference management and garbage collection. Additionally, by providing powerful primitives for address remapping, Mjölnir enables redesign of the I/O stack for greater efficiency and performance with flash. Mjölnir uses techniques from language runtimes for reference management and garbage collection; we show via prototype and experimental evaluation that this design can deliver predictable performance even with varied user workloads across a range of capacity and reference-count scales.

1. Introduction

Flash devices are now ubiquitous in data centers, improving both application performance and data center efficiency by providing an intermediate storage tier in between DRAM and Hard Disk Drives (HDDs) [2, 7, 25]. As flash increases in capacity and drops in cost, larger quantities of data are being stored in flash, leading to demand from users for the same data services available from traditional disk based storage systems. Prior work has observed the impact of flash on storage architectures while also noting that flash presents new challenges in the implementation of classic data services and the expectations placed on them [18, 19, 25, 27, 30, 34].

At the same time, studies have observed that flash presents an opportunity to rethink the overall architecture of the I/O stack, with designs that reuse powerful primitive functions to create composable data services [1, 16, 20–22, 25, 30]. For example, studies such as ANViL [30], FlashTier [25], NVMKV [21] and DFS [16] demonstrate that log structured stores, which are already well-suited to flash, can also provide address mapping capabilities which enable construction of applications and common data services (such as snapshots) with relatively little effort and minimal redundancy in the I/O stack.

In this paper we build upon and extend these key ideas of flexible and powerful address remapping to create Mjölnir, a system that provides the same core primitives as ANViL [30] and advocated by other studies [16, 21, 25, 27, 28]. While prior work focused on the power of the primitive operations themselves and the benefits achievable by redesigning the I/O stack using these primitives, in Mjölnir we focus on the challenges involved in creating a high performance and scalable implementation of the core engine that provides these capabilities.

Prior work has also shown the difficulty in implementing even basic data services at the speeds provided by flash devices [4, 11, 27, 30]. These studies illustrate the need for scalable reference management and garbage collection (GC). Once data sharing is introduced (with copy on write snapshots or in deduplication, for example), the core engine must be able to efficiently find unreferenced data blocks in order to coalesce free space, even while data references are changing simultaneously [10, 14]. While these needs have existed since the era of hard disk based systems, the high performance of flash brings new dimensions to the

* Work done while at SanDisk, Inc.

INFLOW'15, October 4, 2015, Monterey, CA.

http://dx.doi.org/10.1145/2819001.2819006

problem. Flash devices have enough bandwidth to fill their storage capacity in a very short amount of time. Therefore, previously off-line procedures, such as defragmentation, are now required to be done on-line by necessity [27].

Modern SSDs have become increasingly good at GC, simultaneously delivering predictable performance and millions of IOPS [23]. However, systems such as ioSnap have demonstrated that this performance is difficult to maintain in the presence of advanced data services such as snapshots [27]. The primary difference between the address mapping in an SSD and that required by a versioning system is the need to maintain and manage a variable number of references to each data block, which puts pressure on both the foreground and background operations of the system. The presence of frequent remapping operations, such as those generated by ANViL, further increase the performance demands on the GC system.

Scalable reference management has been done in other areas, such as disk based deduplication systems and language runtimes [6, 14]. We leverage insights and techniques from these areas and apply them to the flash domain, extending as necessary. For example, disk-based deduplication systems handle scale in reference count but not at the performance levels of flash [14]. Reference management and GC in managed language runtimes handle scale of reference and speed, but rarely have to operate at the capacity points required of storage systems.

In this paper, we describe the GC design in Mjölnir, a system specifically designed for high performance, scalable reference management and GC for flash storage. The paper makes the following contributions:

- Demonstrates a scalable and highly GC design for flash devices by borrowing concepts from programming language runtimes.
- Illustrates the design tradeoffs between CPU consumption, memory overhead, write amplification, and scalability in high performance GC.
- Shows that it is possible for version creation to remain fast by deferring work to background operations while also making these operations efficient.

The remainder of the paper is organized as follows. Section 2 provides background and an overview of Mjölnir. Section 3 discusses the need for GC and reference management in Mjölnir and explains why existing solutions are insufficient. In Section 4 we describe in detail the design of Mjölnir. Sections 5 and 6 provides an evaluation and conclusion, respectively.

2. Mjölnir background

Virtualization is a popular technique for managing and exploiting available resources in computing systems, from memory and processors [5] to entire machines [8, 12, 13]. Storage virtualization, though conceptually similar, remains fairly limited in use and highly restrictive [26], providing only specialized administrative features like deduplication, snapshots, and thin-provisioning. These operations do not provide any opportunity to the application to leverage storage virtualization directly. Mjölnir aims to address this problem.

Mjölnir draws inspiration from ANViL and is a block device designed to run on top of fast, large flash devices [30]. In addition to providing a standard read/write block interface, Mjölnir also supports fine grained address space manipulation operations through the use of a translation layer (analogous to the address translation in FTLs). These operations help provide storage virtualization at a granularity and level of control that can support a wide variety of real-world applications [30]. We now briefly describe Mjölnir: features, implementation, and the various background work needed to support these operations.

Mjölnir provides a set of simple interfaces to a fine-grained block address map that allow virtualization to be exposed as a first class entity in storage devices. These operations (range copy, move, delete) have been designed to scale on multiple levels: size of extent being manipulated, the frequency of manipulation, and the overall number of such operations. ANViL also demonstrates the usefulness of these operations with features like high-performance volume snapshots, file snapshots and single-write journaling [30]. Mjölnir was designed from the ground up to provide the same address manipulation operations in a fast and efficient manner.

Mjölnir, like most modern storage systems, is log structured, with incoming writes always directed to the tip of the log, never overwriting data in place [24]. This also necessitates a translation layer (TL), where all block address remapping operations take place [3, 15]. Similarly to LFS, we manage physical space in units of segments, where data within a segment is sequential in time, but segments themselves need to be rearranged to reconstruct the time-ordered log. Finally, after a segment has been fully written, it is made immutable. But as the data written in a segment gets overwritten (logically, replaced by data elsewhere in the log), blocks within each segment become invalid. Segments gradually accumulate invalid data in this way and are garbage collected over time; garbage collection is thus a critical component for continuous efficient operation in Mjölnir.

3. GC Considerations for Mjölnir

Mjölnir requires a different background GC from a conventional SSD, primarily due to its many-to-one address map [24]. A GC for a traditional log-structured storage system like the one described in LFS is simple, with each block referenced by at most one logical location. Since Mjölnir aims to support much richer capabilities, a single physical data block may be referred to by more than one logical ad-

dress, with reference count ideally limited only by the available physical storage capacity. In this section, we outline our design goals and illustrate why traditional GC techniques are not directly applicable for Mjölnir.

3.1 Design Goals

Scale with Capacity: Modern storage systems are growing in capacity and now support terabytes to petabytes of data. The GC should be able to scale to large storage capacities.

Scale with References: Advanced virtualization support within storage devices increases the number of references to physical blocks. The garbage collector should be able to scale with both number of references and frequency with which these references are created.

Predictable Performance: Performance is improving with every generation of non-volatile memory devices, with a single modern flash drive capable of delivering anywhere from hundreds of thousands to millions of IOPS. Moreover, applications expect predictable performance from storage systems. As a result, background operations cannot take too long and must be able to keep up with foreground operations. If not, a huge backlog can accumulate and degrade foreground performance.

Manage Memory Usage: Memory is always a precious resource and the design of the GC must to be mindful of this issue. The design should be able to handle large-scale (in both capacity and reference count) storage systems even with relatively small amounts of available memory. This is very true especially in the case of an "off-load" device where virtualization support is embedded within device DRAM. The GC should be able to trade off CPU and GC efficiency against memory usage when necessary.

3.2 Alternate Approaches

There are many different ways of implementing a GC for a log-structured storage system. We now look at existing approaches and explain why Mjölnir requires a different (hybrid) solution.

3.2.1 Bitmaps

Bitmaps are perhaps the most obvious potential approach to GC. With bitmaps, tracking utilized and free blocks is very straightforward [24]. While bitmaps are efficient in both memory and CPU utilization, they are insufficient to track used/free status in the context of the many-to-one address map used by Mjölnir. For example, a simple set-on-map, clear-on-unmap bitmap-management algorithm would be inaccurate if one were to simply clone a live block's mapping to a new logical address and then unmap the original address (the block would have a live reference but incorrect bitmap state).

3.2.2 Reference Counting

Alternatively, one could use an array of reference counts to track the number of mappings to each block. In fact, a bitmap is simply a one-bit special case of a reference count array. If we generalize the bitmap approach to use multi-bit reference counts, we can address the inaccuracy problem inherent to bitmaps tracking a many-to-one address map, using a simple increment-on-map, decrement-on-unmap reference count management algorithm. This approach introduces a follow-on question to which there is no clear, obviously-correct answer: how large do we make the reference counts? Larger reference counts require more memory to store, but smaller ones impose undesirable limitations on the usage of the special features offered by Mjölnir. Even if this question is answered, it still does not address a significant need for GC. The GC in a multi-reference log-structured system needs to know not only the liveness status of physical blocks and reference count for each block, but also *where* those mappings are, so that it can update those mappings after copying data to a new location. Regardless of their size, reference counts simply do not provide this information, meaning that in addition to its expense in DRAM consumption, this would be at best an incomplete solution.

3.2.3 Synchronous Reverse Map

To overcome the limitations of reference counts, one could expand the GC's metadata-tracking to use a full reverse map (mapping each physical address to zero, one, or more logical addresses) in addition to the forward map in the TL. This would of course provide all the information provided by reference counts (and, depending on its exact implementation, likely avoid arbitrary limits on the number of references to a given block), and also be able to supply the necessary information for the GC to update the address map after copying data. However, a reverse map would require at least as much DRAM space as the forward map, and likely more, since the data structure mapped to by each physical address would be a set that would have to support reasonably efficient insertion and deletion. The cost of implementing this would simply be unacceptably high in terms of DRAM consumption in addition to the extra book-keeping work it would require during foreground I/O operations.

3.3 Mark and Sweep

Mark and sweep is a GC approach categorized under the class of tracing garbage collectors [31]. A tracing GC is used to manage resources (commonly memory) by evaluating the reachability of memory extents (objects) starting from a set of root objects. In the context of programming languages (such as Java [29]), objects may have zero or more references at a given point in time; a live object will be reachable by at least one path. Objects without any references are ready for reclamation.

Category	Features	Accuracy	CPU	Memory
SSDs (Storage)	Low	High	Low	Low
Snapshotting Systems	Low	High	Med	High
Prog. Languages	High	High	High	High
Dedupe Systems	High	Med	High	High

Table 1. Comparison of garbage collecting systems

We compare various systems requiring garbage collection along four axes: features, accuracy, and CPU and memory utilization. "Features" refers to the data management features (pointers/references in a PL GC, snapshots in storage) supported by the system. Accuracy is the ability to identify reclaimable blocks most efficiently (relative to the CPU and memory overheads incurred).

Mark and sweep, as its name suggests, consist of two phases. The mark phase involves performing a complete reachability analysis on the entire object graph, during which all reachable objects are marked. The sweep phase then follows and reclaims all objects that were not marked in this manner.

Mark and sweep has also been explored in the context of storage systems [10, 14, 17]. For example, deduplication systems [14] have used mark and sweep to improve single node scalability. Similarly, BigTable [10] uses a mark and sweep based garbage collector to cleanup its SSTables.

3.4 Comparison with Other Systems

Even though Mjölnir is a storage system, given its features, its GC has more in common with those of programming language runtimes than most storage systems. Mjölnir supports a large number of references to its data objects (blocks) and reclaims space online, in real time. Also, as illustrated in Table 1, existing storage systems offer relatively few features (i.e., support only read/write operations) and so their garbage collectors are very accurate while consuming little CPU and memory. Mjölnir, being much more feature-rich, cannot use such techniques, so we look to programming languages for inspiration. As we will describe in subsequent sections, the garbage collector of Mjölnir adopts a mark and sweep approach, but provides the ability to control memory and CPU overheads during this process.

4. Design

Mjölnir uses a mark and sweep based GC [9]. The core design philosophy of Mjölnir is to make the common case operations as fast as possible and move the infrequent latency insensitive operations to the background [27]. This approach puts pressure on Mjölnir's background GC to reclaim data efficiently and ensure predictable performance to foreground operations.

4.1 Key Observations

There are a few key observations about multi-reference log-structured systems that we have used to design an optimized GC.

Translation Layer is Omniscient: The TL maintains the logical to physical address mapping and represents the entirety of the storage device's metadata; information about any physical block can be inferred via the TL. Moreover, TLs are almost always kept in DRAM (in host- or device-based FTLs) and thus offer much faster access (nanosecond latency) as compared to I/Os issued to the flash device.

Block Liveness: A block is live if there is at least one reference to it in the TL; block liveness can thus be inferred by traversing the TL and hence bitmap management for the entire storage device can be avoided. Once a block is dead (unreferenced), its status cannot change until the segment containing the block is recycled.

Reference Counting: Reference counts need not be actively maintained in the system to aid the GC. The simple boolean of block liveness is enough to decide which segments to reclaim. More importantly, the reverse map for a segment (i.e., the logical addresses mapped to blocks within a segment) can be constructed on-demand by traversing the TL.

Reclamation of Segments: Data in a log-structured system is always reclaimed in segments (large, physically contiguous extents). The validity of blocks within a segment helps determine if a segment could be a candidate for reclamation. The GC can thus use only a small amount of in-memory state to keep the information necessary for selecting segments as candidates for reclamation.

Keeping up with foreground I/O: In steady state operation, GC needs to reclaim space at the same average rate at which its being consumed. Though it is desirable to select segments optimally (perhaps using an algorithm that differentiates hot data from cold), it is more important reclaim space fast enough to avoid pauses in I/O traffic.

4.2 Mark Phase: Scanner

The goal of the *Mark Phase* is to identify segments for reclamation. The segments to be garbage collected may contain both valid and invalid data. Ideally, the goal is to find empty or near empty segments as this minimizes the amount of data copied forward, reducing write amplification [24]. There are many challenges to identifying candidate segments for GC (see Section 3).

The component of Mjölnir that performs this phase is the scanner. The scanner consists of one or more threads that periodically traverse the TL to collect valid data block references to identify segments for cleaning. Mjölnir leverages the observations described earlier and splits the mark phase into two parts. The first, *candidate selection*, scans through the TL to identify potential segments that fall below the desired data-validity threshold. This is accomplished by building a bitmap for immutable segments to get an accurate view of the amount of live data in each segment. The second part of the mark phase, *candidate preparation*, scans through the

TL again to reconstruct the reverse map for the live data blocks in the selected candidate segments.

The mark phase ultimately determines the overall write amplification introduced in the system: a poor choice of candidate segment may lead to inefficient space reclamation as well as device wearout caused by excessive writing. The mark phase also implicitly becomes a bottleneck on the sweep phase: without the mark completing, the sweep cannot proceed. Thus, the speed and the accuracy of the mark phase is critical. Scanning more quickly increases CPU utilization, while more careful candidate selection incurs greater memory consumption.

4.2.1 Selective Segment Tracking

The scanner is required to construct reverse maps for candidate segments for use during the sweep phase. As explained in Section 3, reverse maps are expensive; a full-system reverse map would cause significant memory bloat. Thus, it is important to control the memory consumed by these maps.

The only way by which memory consumption can be reduced without compromising data integrity is to construct reverse maps for fewer segments. The problem is thus to select a small subset of segments based on overall space pressure and some limit of permissible memory consumption. A given segment must undergo two full scan cycles before it can be reclaimed. The first scan populates a bitmap for each segment to determine how much valid data it contains. Once the utilization gets low enough and it is selected as a candidate for reclamation it is treated specially in the following scan, during which a reverse map is constructed for all valid blocks in the selected segments. One could consider other segment selection policies, but for simplicity we have employed a greedy policy, which has shown itself to be effective.

4.2.2 Pipelined Scanning

For performance, the two scan phases are pipelined (i.e., they can be run concurrently for different segments). It is important to note that the second scan (reverse-map construction) is only dependent on the first (candidate selection) for a given individual segment. As a result, on any given scan cycle, the scanner can be performing the work of the first scan on one set of segments and the work for the second on another (disjoint) set of segments, effectively pipelining them. While pipelined scanning does increase ”latency” of the reclamation of any individual segment (since it must take two complete trips through the scanner), this is not an important metric for Mjölnir, for which GC throughput (which is not negatively affected by pipelined scanning) is much more critical. Figures 1 (a) and (b) provide an overview of the scan stage.

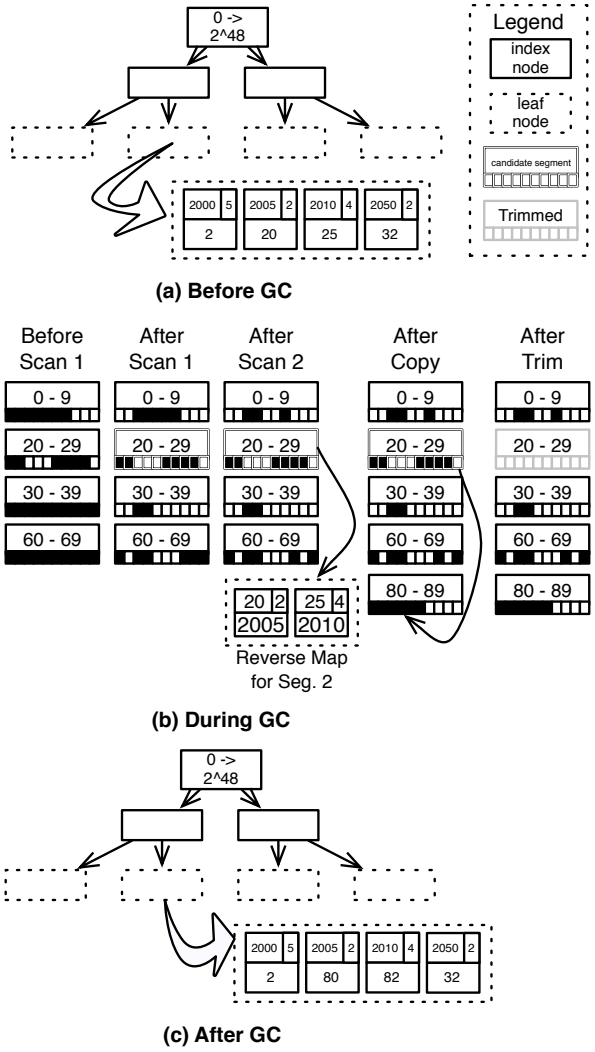

Figure 1. Mjölnir’s GC in action

The above figures illustrate Mjölnir’s mark and sweep GC in action. Figure (a) shows a btree-like TL that maps a logical space ranging from address 0 to 2^{48}. The solid boxes represent index nodes and the dotted lines indicate leaf nodes. In this simplistic example, we show a single leaf node with 4 mapping entries. Each entry contains a logical address (top left), extent length (top right), and a physical address (in the box below). For example, the first entry represents a mapping from logical address range [2000, 2004] to physical addresses [2, 6]. For simplicity, each segment is shown as having 10 blocks. Figure (b) illustrates the construction of bitmaps and reverse maps during the scan. We show four segments in the left most ”before scan 1” segment. The addresses of these segments are listed within each rectangle with a bitmap below (dark areas represent valid addresses). After Scan 1, segment [20, 29] is identified as a candidate. Scan 2 helps build a reverse map for this segment using the entries it found in the leaf node from Figure (a). These entries are then copied to new locations in segment [80, 89] and finally segment [20, 29] is trimmed and reclaimed. Figure (c) shows the state of the same leaf node after the GC cycle has completed.

4.2.3 Parallel Scanner Threads

Because the work of a scan cycle is entirely CPU-bound, potentially very large, and parallelizable, the scanner is multi-threaded. Each thread is given a subset of the logical address space to scan. How to divide the logical address space among these threads, however, is a surprisingly difficult question. The logical address space is sparsely populated, and the scanner only traverses mapped addresses in the TL. In order to spread work evenly among scanner threads, each thread should scan approximately the same number of mappings. The scanner, however, has no high-level picture of the distribution of mapped addresses within the logical address space and as such it is not trivial to divide up the work evenly.

To address this issue, we devised a dynamic space reassignment algorithm. This algorithm is based on the insight that there is no actual need for the division of logical address space between threads to be statically determined at the start of each scan cycle. When any thread finishes its partition, it sets a global flag requesting that the remaining scanning work be redistributed. Each running thread checks this flag periodically, and upon observing it having been set, records the progress it has made and proceeds to wait at a barrier. When all threads have reached the barrier, a leader thread then splits up the remaining work among any idle threads. The scanner threads are then released from the barrier and begin scanning their newly-reassigned address space, repeating the reassignment process when any threads finish their work.

4.3 Sweep Phase: Cleaner

The goal of the cleaner, Mjölnir's *Sweep Phase*, is to actually reclaim the unused space in candidate segments. The cleaner must efficiently move all valid data identified by the scanner out of candidate segments and into new locations, updating the TL accordingly before freeing the segments. Here we describe these steps in detail.

Copy-Forward Valid Data: The cleaner needs to relocate all valid data (if any) in a candidate segment to a new location on the log in order to reclaim the segment. To relocate data, the cleaner first issues reads to all valid data blocks identified by the scanner. When these reads complete, the cleaner allocates new space (at the head of the log) and writes the data out to these new locations. It is important to note that, even if the copy-forward fails for some reason, the old data still remains and can be used to service foreground read operations.

Update Translation Layer: The cleaner updates the TL only after the data has been successfully moved to its new location. After a successful write but before this point, both the old and new location of the data are valid and either could be used to service reads. Thus, even with multiple blocks of valid data to be moved, the TL can be updated one mapping at a time without compromising data integrity. Figures 1 (a) and (c) illustrate how mappings are updated after cleaning.

Reclaim Segment: After the TL has been successfully updated for the valid data within a segment, the segment can be trimmed and returned to the space allocator for reuse. Once the segment is trimmed, the old data no longer exists and any reference to such a segment would be invalid. Figures 1 (b) and (c) provide an overview of some of the cleaner's operation.

4.3.1 Callbacks to Reduce Write Amplification

GC always leads to some write amplification in a log-structured system and a large body of existing work already describes policies to minimize it [24, 32, 33]. The aspect of write amplification that is unique to Mjölnir is introduced by the two phases of mark and sweep. The reverse map generated during the mark phase for a segment represents all valid data within the segment at the time the reverse map was constructed by the scanner. But this state could potentially change with user writes or address manipulation operations. Moreover, by the time the cleaner can act on a segment during the sweep phase, the amount of valid data in the segment may have changed. Note however that the amount of valid data within a segment can only decrease, never increase.

To ensure the cleaner does not perform any wasted work, we use callbacks on our TL that are triggered when ever an existing piece of data is overwritten. The old mapping information available to the callbacks is used to check if it belongs to a segment that is being cleaned and if so, remove the corresponding entries on the fly.

4.3.2 Rate Limiter for Predictability

The act of GC forward-copying data necessarily interferes with the foreground user I/O traffic by using some of the available bandwidth of the underlying storage device. This could potentially introduce I/O pauses or latency spikes. Mjölnir tackles this problem using an explicit rate limiter. The job of the rate limiter is to determine what fraction of the total available bandwidth is given to the foreground and background (GC) traffic.

The rate limiter takes into account the overall available free space in the system (number of free segments), the invalidity of the segments tracked by the scanner, and the amount of work that is outstanding for the user and the cleaner. At a high level, the fraction of bandwidth given to the GC increases as free space drops (as long as the cleaner has useful work). Finally, the rate limiter also decides when the system has entered ”panic” mode; when the system is critically low on space it suspends user traffic and dedicates all available bandwidth to the GC. In interest of space, we do not describe the rate limiter in detail in this paper.

5. Evaluation

Mjölnir is a complex system (approximately 100K lines of code), with GC contributing significantly to its complexity. In this section, we will restrict our evaluation to only the GC and examine its functionality, predictability, scalability, and interaction with foreground traffic. To better understand GC in Mjölnir, we will also dig deeper into the various aspects that contribute to the cost of reclaiming a segment. We will also demonstrate Mjölnir's ability to scale with references (i.e., snapshots). For our evaluation, we use SLES 11SP2 with a 3.0 Linux kernel running on an HP DL380 server with 64GB of RAM, two 6-core (12-thread) Intel Xeon processors, and a 1.2 TB SanDisk ioMemory PCIe flash drive as Mjölnir's backing storage. Mjölnir is configured to run with a segment size of 128M.

5.1 GC in Action

Here we demonstrate the real-time operation of the GC. Specifically, we show how the GC slowly ramps up its activity as device utilization increases. To illustrate this behavior, we artificially reduced the capacity of the drive to 320 GB. We ran a fio workload with 32 threads each writing 10 GB of data in 32K blocks, with about 50% overwrites. Figure 2 shows the progress of the system over time.

Initially, when the device utilization is under 50%, the foreground traffic is granted the full device bandwidth and performs its work at full throttle. Once the utilization crosses 50%, the GC activates, starts performing scans of the TL and begins moving blocks as is observed after approximately 50 seconds. We observe a gradual decrease in the fraction of bandwidth given to the foreground traffic to allow the GC to do its work. More importantly, we also observe that the GC is keeping up with foreground traffic, as evidenced by the roughly equal rates of segment allocation and reclamation. From the backing-device I/O graph, we can see that the presence of the GC introduces reads, which also has the implicit effect of reducing the overall write bandwidth.

An interesting behavior of flash devices is also revealed in Figure 2. In the pure write workload, the device can achieve full bandwidth. But in the presence of a mixed workload, especially the combination of large writes with small reads, the write bandwidth suffers disproportionately. From the graph, we can also see that the rate-limiter attempts to minimize this impact. This, unfortunately, is the behavior of the drive even in the absence of Mjölnir.

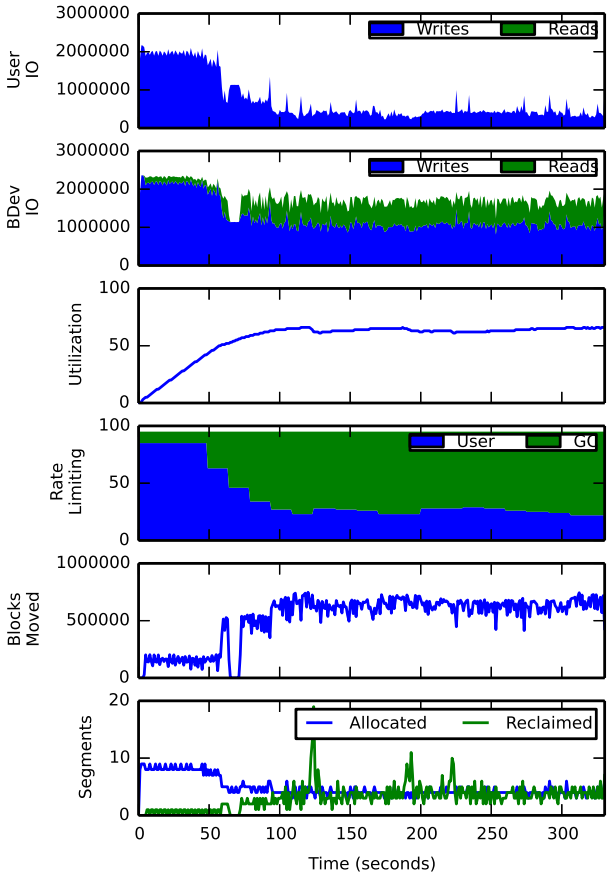

Figure 2. GC in action

This figure shows the operation of the GC at various utilization levels. The graphs are (from top to bottom): user and backing device I/O rates (sectors per second), device-wide segment utilization, bandwidth share between GC and foreground I/O, sectors per second moved by GC, and segment allocation and reclamation rates.

Data	Mappings	Time to reclaim
128M	3,712	329ms
8G	266,952	28.3s
64G	2,152,134	179.4s
128G	4,037,073	363.7s

Table 2. GC scaling with capacity

This table illustrates how Mjölnir scales with capacity. We populate the device with some data and alter the GC to clean segments even though they contain only live data. The cost in time and mappings scanned thus represents the time spent by the GC in moving all of the data that was originally written.

5.2 Scaling GC with capacity

It is important for the GC to scale gracefully with the volume of data being managed. We measure the time taken by the GC to reclaim the valid data from a set of segments. In these experiments, we issue 4K sequential writes to the device and allow the GC to start processing after all the writes complete. We adjust the GC's candidate-selection threshold to make it reclaim any segment, regardless of its utilization. In this particular experiment, as we have no overwrites in our initial workload, the segments contain only valid data.

Table 2 shows the number of mappings discovered during the scan and the time taken by the GC to reclaim all segments (representing all the data that was written); the scan time increases linearly with the quantity of data. Linear scaling

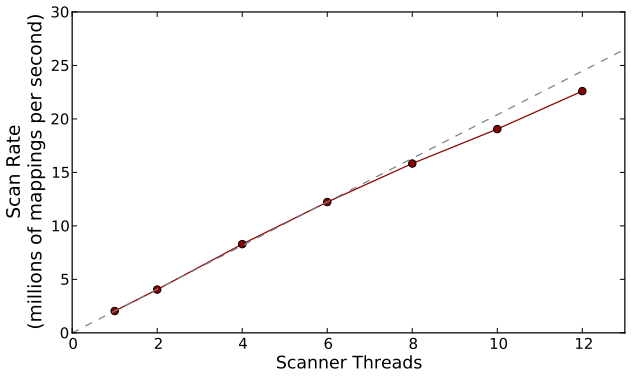

Figure 3. Scanner scalability

This figure illustrates the scalability of the scanner, showing scanning performance at varying thread counts. The scanner achieves near-linear scaling up to 12 threads (the number of available CPU cores).

may not always be desirable when it comes to large backing devices. In the future, we can apply work saving techniques like preferential scanning of modified regions to reduce scan space and control scan times.

The scan time can be reduced substantially by parallelizing the scan across multiple threads. We populated the drive with 128 MB of data (one segment's worth) and created 10,000 snapshots of it (to fill up the TL with five times as many mappings as would be used by 256 GB of non-snapshotted data). We then measured the performance of the scanner with varying numbers of threads; the results are shown in Figure 3. We can see from this data that Mjölnir's scan performance scales well with increasing numbers of scanner threads, allowing Mjölnir to meet its goal of scaling to large numbers of data references by leveraging the plentiful CPU cores of modern systems.

5.2.1 Reclamation Time Breakdown

The time required to reclaim space includes the time spent scanning the TL and the time taken to physically move the data and update the TL. To understand how this time is spent, we split the time into various sub-components.

As mentioned previously, our segment size is configured to 128 MB. We first look at the cost of reclaiming a single segment with 128 MB of data (all valid). The scanner found 3,712 mappings and this phase took 340us. The cleaner took 329ms to move the data and perform the updates. It is not feasible to separate the data movement from the TL updates costs as these occur asynchronously and in parallel. Thus, as expected, data copying consumes most of the time spent on reclaiming a given segment.

5.3 Scaling with References

The final piece of our evaluation demonstrates the GC's scaling with large numbers of data references. We measure the cost of cleaning a single segment of data while varying the number of references to the data (snapshots). We populate the device with 128 MB of sequentially-written data and configure the GC to reclaim any and all segments. Table 3 shows snapshot time, number of mappings scanned, and time to reclaim a single segment with varying numbers of mappings.

Snapshots	Mappings	Time to reclaim
1	7,420	338.40ms
200	731,990	2.38s
2,000	7,287,642	29.6s

Table 3. GC Scaling with snapshots

Here we evaluate the cost of cleaning a single segment of data (128 MB) while varying the number of data references (snapshots). We populate the device with 128 MB of sequentially-written data and configure the GC to reclaim any and all segments; the time to reclaim a segment scales with the number of snapshots, due to the time spent scanning the TL.

The time to reclaim includes the time to move the data and update mappings. Comparing the time to reclaim with varying numbers of snapshots, we see that the system can handle an increasing number of snapshots quite comfortably. This is facilitated by a set of IO completion threads (out of scope for this paper) that handle most of the data moving and TL-update work.

Our evaluation has demonstrated the operation and scalability of Mjölnir's GC. There are however several pieces of evaluation that we omit due to space constraints, including scanner self-scaling (adjusting its CPU consumption to an appropriate level), memory consumption, and performance tradeoffs.

6. Conclusions

Mjölnir demonstrates the feasibility of an online mark and sweep style GC for log-structured storage systems supporting large numbers of data references. Mjölnir borrows ideas from programming language runtimes to create a unique GC that is tuned to scale with both capacity and references while still controlling CPU and memory overheads. We hope Mjölnir can inspire new storage systems to consider new methods for GC to adapt themselves to the demands of fast flash and persistent memory devices.

Acknowledgements

We thank the anonymous reviewers for their insightful feedback. Zev Weiss, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau are supported by NSF grants CNS-1421033, CNS-1319405, and CNS-1218405. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and may not reflect the views of NSF or other institutions.

References

[1] Native Flash Support for Applications. http://www.flashmemorysummit.com/.

[2] ioCache. http://www.fusionio.com/products/iocache, 2012.

[3] AGARWAL, N., PRABHAKARAN, V., WOBBER, T., DAVIS, J. D., MANASSE, M., AND PANIGRAHY, R. Design Tradeoffs for SSD Performance. In *Proceedings of the USENIX Annual Technical Conference (USENIX '08)* (Boston, Massachusetts, June 2008).

[4] ANDERSEN, D. G., FRANKLIN, J., KAMINSKY, M., PHANISHAYEE, A., TAN, L., AND VASUDEVAN, V. Fawn: A fast array of wimpy nodes. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles* (2009), SOSP '09, pp. 1–14.

[5] ARPACI-DUSSEAU, R. H., AND ARPACI-DUSSEAU, A. C. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 2014.

[6] BACON, D. F., ATTANASIO, C. R., LEE, H. B., RAJAN, V., AND SMITH, S. Java without the coffee breaks: A nonintrusive multiprocessor garbage collector. In *ACM SIGPLAN Notices* (2001), vol. 36, ACM, pp. 92–103.

[7] BADAM, A., PARK, K., PAI, V. S., AND PETERSON, L. L. Hashcache: cache storage for the next billion. In *Proceedings of the 6th USENIX symposium on Networked systems design and implementation* (2009), NSDI'09.

[8] BUGNION, E., DEVINE, S., AND ROSENBLUM, M. Disco: Running Commodity Operating Systems on Scalable Multiprocessors. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)* (Saint-Malo, France, Oct. 1997), pp. 143–156.

[9] CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W. C., WALLACH, D. A., BURROWS, M., CHANDRA, T., FIKES, A., AND GRUBER, R. Bigtable: A Distributed Storage System for Structured Data. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)* (Seattle, Washington, Nov. 2006), pp. 205–218.

[10] CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W. C., WALLACH, D. A., BURROWS, M., CHANDRA, T., FIKES, A., AND GRUBER, R. E. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS) 26*, 2 (2008), 4.

[11] DAS, D., ARTEAGA, D., TALAGALA, N., MATHIASEN, T., AND LINDSTRÖM, J. Nvm compression—hybrid flash-aware application level compression. In *2nd Workshop on Interactions of NVM/Flash with Operating Systems and Workloads (INFLOW 14)* (Broomfield, CO, Oct 2014).

[12] DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T., HO, A., PRATT, I., WARFIELD, A., BARHAM, P., AND NEUGEBAUER, R. Xen and the Art of Virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)* (Bolton Landing, New York, October 2003).

[13] GOLDBERG, R. Survey of Virtual Machine Research. *IEEE Computer 7*, 6 (1974), 34–45.

[14] GUO, F., AND EFSTATHOPOULOS, P. Building a high-performance deduplication system. In *USENIX Annual Technical Conference* (2011).

[15] GUPTA, A., KIM, Y., AND URGAONKAR, B. DFTL: a Flash Translation Layer Employing Demand-Based Selective Caching of Page-Level Address Mappings. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIV)* (Washington, DC, Mar. 2009), pp. 229–240.

[16] JOSEPHSON, W., BONGO, L., LI, K., AND FLYNN, D. Dfs: A file system for virtualized flash storage. In *Usenix Conference on File and Storage Technologies* (February 2010).

[17] JUUL, N. C., AND JUL, E. Comprehensive and robust garbage collection in a distributed system. In *Memory Management*. Springer, 1992, pp. 103–115.

[18] KIM, J., LEE, D., AND NOH, S. H. Towards slo complying ssds through ops isolation. In *13th USENIX Conference on File and Storage Technologies (FAST 15)* (Santa Clara, CA, February 2015), USENIX Association, pp. 183–189.

[19] LEE, C., SIM, D., HWANG, J., AND CHO, S. F2fs: A new file system for flash storage. In *13th USENIX Conference on File and Storage Technologies (FAST 15)* (Santa Clara, CA, February 2015), USENIX Association, pp. 273–286.

[20] LU, Y., SHU, J., AND WANG, W. Reconfs: A reconstructable file system on flash storage. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST 14)* (Santa Clara, CA, 2014), USENIX, pp. 75–88.

[21] MARMOL, L., SUNDARARAMAN, S., TALAGALA, N., AND RANGASWAMI, R. Nvmkv: A scalable, lightweight, ftl-aware key-value store. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)* (Santa Clara, CA, July 2015), USENIX Association, pp. 207–219.

[22] OUYANG, X., NELLANS, D., WIPFEL, R., FLYNN, D., AND PANDA, D. K. Beyond block i/o: Rethinking traditional storage primitives. In *High Performance Computer Architecture* (Feb 2011).

[23] RESEARCH, H. World's Fastest SSD. https://www.hgst.com/press-room/press-releases/HGST-Research-Demonstrates-World-s-Fastest-SSD-at-Flash-Memory-Summit-.

[24] ROSENBLUM, M., AND OUSTERHOUT, J. The Design and Implementation of a Log-Structured File System. *ACM Trans. Comput. Syst. 10*, 1 (Feb. 1992), 26–52.

[25] SAXENA, M., SWIFT, M. M., AND ZHANG, Y. Flashtier: A lightweight, consistent and durable storage cache. In *Proceedings of the 7th ACM European Conference on Computer Systems* (2012), EuroSys '12, pp. 267–280.

[26] SIVATHANU, G., SUNDARARAMAN, S., AND ZADOK, E. Type-Safe Disks. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)* (Seattle, Washington, Nov. 2006).

[27] SUBRAMANIAN, S., SUNDARARAMAN, S., TALAGALA, N., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Snapshots in a flash with iosnap. In *Proceedings of the Ninth European Conference on Computer Systems* (2014), EuroSys '14, pp. 23:1–23:14.

[28] VASUDEVAN, V., KAMINSKY, M., AND ANDERSEN, D. G. Using vector interfaces to deliver millions of iops from a networked key-value storage server. In *Proceedings of the*

Third ACM Symposium on Cloud Computing (New York, NY, USA, 2012), SoCC '12, ACM, pp. 8:1–8:13.

[29] VENNERS, B. *Inside the Java virtual machine*. McGraw-Hill, Inc., 1996.

[30] WEISS, Z., SUBRAMANIAN, S., SUNDARARAMAN, S., TALAGALA, N., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. ANViL: Advanced Virtualization for Modern Non-Volatile Memory Devices. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST '15)* (Santa Clara, CA, February 2015).

[31] WILSON, P. R. Uniprocessor garbage collection techniques. In *Memory Management*. Springer, 1992, pp. 1–42.

[32] YANG, J., PLASSON, N., GILLIS, G., AND TALAGALA, N. Hec: improving endurance of high performance flash-based cache devices. In *Proceedings of the 6th International Systems and Storage Conference* (2013), ACM, p. 10.

[33] YANG, J., PLASSON, N., GILLIS, G., TALAGALA, N., AND SUNDARARAMAN, S. Don't stack your log on my log. In *2nd Workshop on Interactions of NVM/Flash with Operating Systems and Workloads (INFLOW 14)* (Broomfield, CO, Oct 2014), USENIX Association.

[34] ZHENG, D., MHEMBERE, D., BURNS, R., VOGELSTEIN, J., PRIEBE, C. E., AND SZALAY, A. S. Flashgraph: Processing billion-node graphs on an array of commodity ssds. In *13th USENIX Conference on File and Storage Technologies (FAST 15)* (Santa Clara, CA, February 2015), USENIX Association, pp. 45–58.