Purity: Building Fast, Highly-Available Enterprise Flash Storage from Commodity Components

John Colgrove, John D. Davis, John Hayes, Ethan L. Miller, Cary Sandvig,
Russell Sears*, Ari Tamches, Neil Vachharajani, and Feng Wang
Pure Storage
Mountain View, CA, USA
coz,johnd,jhayes,elm,cary,sears,ari,neil,fwang@purestorage.com

ABSTRACT

Although flash storage has largely replaced hard disks in consumer class devices, enterprise workloads pose unique challenges that have slowed adoption of flash in "performance tier" storage appliances. In this paper, we describe Purity, the foundation of Pure Storage's Flash Arrays, the first all-flash enterprise storage system to support compression, deduplication, and high-availability.

Purity borrows techniques from modern database and key-value storage architectures, and introduces novel storage primitives that have wide applicability to data management systems. For instance, all writes in Purity are monotonic, and deletions are handled using an atomic predicate-based tuple *elision* primitive.

Purity's redundancy mechanisms are optimized for SSD failure modes and performance characteristics, allowing for fast recovery from component failures and lower space overhead than the best hard disk systems. We built deduplication and data compression schemes atop these primitives.

Flash changes storage capacity/performance tradeoffs: unlike disk-based systems, flash deployments are rarely performance bound. A single Purity appliance can provide over 7 GiB/s of throughput on 32 KiB random I/Os, even through multiple device failures, and while providing asynchronous off-site replication. Typical installations have 99.9% latencies under 1 ms, and production arrays average 5.4× data reduction and 99.999% availability.

Purity takes advantage of storage performance increasing more rapidly than computational performance to build a simpler (with respect to engineering, installation, and management) scale-up storage appliance that supports hundreds of terabytes of highly-available, high-performance storage. The resulting performance and capacity supports many customer deployments of multiple applications, including scale-out and parallel systems, such as MongoDB and Oracle RAC, on a single Purity appliance.

*Corresponding author

SIGMOD'15, May 31–June 4, 2015, Melbourne, Victoria, Australia.

ACM 978-1-4503-2758-9/15/05 ...$15.00.
http://dx.doi.org/10.1145/2723372.2742798

Categories and Subject Descriptors

D.4.2 [**Storage Management**]: Storage hierarchies; Secondary storage

Keywords

Storage Area Networks; Enterprise Flash Storage; Deduplication; Log structured storage; High availability; Scale up architectures

1. INTRODUCTION

Most "performance tier" applications—those that favor I/O rate over capacity—store their data in virtual block devices that are exposed over the network by enterprise storage arrays. Historically, such systems used hard disks, so random reads and writes were expensive, but raw storage capacity was cheap. As a result, large-scale storage resorted to scale-out architectures to achieve reasonable performance.

Flash reverses this capacity/performance tradeoff and, as flash storage capacity becomes cheaper, it has begun to displace disk for performance tier applications. However, enterprise users demand resiliency, high availability, and scalability in addition to performance, and want it all at no higher cost per gigabyte than they pay for high-performance disk. In this paper, we present Purity, a system that uses commodity solid state disks (*SSDs*) and servers to meet these requirements.

Enterprise storage users frequently make clones, snapshots, and off-site copies of volumes to provide data resiliency. To make these operations more efficient and less disruptive, we store data in *mediums*—coarse-grained mappings that we expose as virtual containers. This allows us to virtualize all storage resources, making many of the system features that conserve capacity easier to implement, at the expense of additional random reads.

We leverage flash's ability to perform fast random reads and sequential writes by compressing data and storing a single instance of duplicate blocks written to different logical addresses. Using these and other techniques, we significantly reduce the amount of flash capacity needed for a given workload. This provides us cost parity with disk capacity at significantly higher levels of availability and performance than disk.

Although they continue to improve, SSDs pay a large penalty for random writes [55], so Purity uses log-structured indexes and data layouts to ensure that data is written in large sequential chunks. We stripe the data written by these structures across many drives, and use Reed-Solomon encod-

ing for redundancy, allowing Purity to tolerate the loss of two SSDs without losing availability. In fact, we encourage potential customers to pull drives and unplug controllers as they evaluate Purity and competitive products.

Flash products for higher performance environments focus on direct-attached storage instead of redundant deployments. This causes them to become unavailable during machine failures, and prevents deduplication and load balancing across multiple tenants. In contrast, Purity allows quick failover from one controller to a live spare by having the spare "follow" the primary system, at significantly lower costs.

On average, these data reduction and fault tolerance techniques provide real-world applications with 5.4× more effective storage than the physical storage in the system, excluding gains from "thin provisioning," which would count unused space as application data. On average, our customers provision approximately 12× more virtual space than physical storage. We publish these numbers continuously[1] as averages across customer installations that are constantly monitored by our operations team.

This paper makes the following contributions: In Section 2 we examine the implications of flash storage on disk-based enterprise and scale-out storage architectures. Section 3 presents a set of design principles, motivated by these hardware trends and recent research results, upon which Purity is based. Next, in Section 4 we discuss Purity's implementation in more detail, showing that aggressive data reduction can be integrated with solid state storage while maintaining high performance. Finally, Section 5 discusses lessons learned from real-world deployments of Purity, and the implications of flash storage on application and database infrastructure.

2. BACKGROUND

Storage software designs are closely coupled to the performance and reliability characteristics of the underlying storage devices. As solid state disks replace hard disks, they are forcing a rewrite of legacy storage systems. Purity is one such redesign, aimed at efficiently using commodity flash drives to provide enterprise-quality block storage.

This section provides an overview of solid state disk (SSD) technology, and then describes the approaches that existed when we began development of Purity. We describe the impact of affordable SSDs on enterprise storage, and then argue that the vast majority of performance-oriented disk-based key-value applications are now better served by scale-up storage architectures.

2.1 Solid State Disks

SSDs are much faster and more reliable than mechanical disks, but they have performance quirks that prevent unmodified legacy storage systems from using them optimally. Purity and other SSD-optimized storage systems make up much of the difference by acting as a translation layer for existing applications.

An SSD consists of a set of flash chips that store data persistently and an ASIC with a purpose-built processor, error correction and redundancy hardware (Figure 1). This is driven by firmware that is at least as complicated as the operating system storage stack. The ASIC and firmware

[1] `http://www.purestorage.com/`

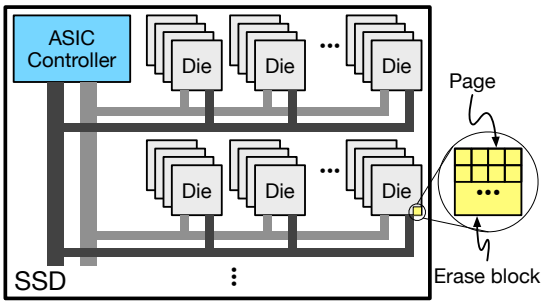

Figure 1: Architecture of an SSD.

combination is usually called a flash translation layer (FTL). Each flash chip contains a set of independent dies that can run in parallel; typical SSDs do not reach peak throughput with read queue depths less than 32.

Each die is broken into erase blocks of 2–16 MiB, which are, in turn, broken into pages of 512–4096 bytes. Pages are the minimum read/write unit. However, a page must be erased at some point before being written, and an entire erase block is erased in a single operation.

Because SSDs cannot efficiently overwrite data in place, the FTL must maintain mappings between logical sector addresses and physical locations. This process is complex and undocumented, and varies across vendors, models, and even firmware revisions, leading to performance quirks. Worse, while an SSD is erasing a block, it cannot read data from physically-related blocks, leading to read latency spikes. Purity goes to great lengths to expose simple append-only write workloads to the SSDs, and to avoid creating situations that expose these performance issues, as described in Section 4.4. We also carefully validate firmware revisions to ensure that this approach leads to deterministic, reliable behavior in practice.

Modern SSDs can store from 1–4 bits per cell. Storing more bits per cell increases data density, but decreases *endurance*—the number of times a given cell may be erased and still reliably store data. SLC SSDs, which store 1 bit per cell, are rated to support about 100,000 program/erase (P/E) cycles. Multi-level cell (MLC) SSDs store 2 bits per cell, and typically support 3000–5000 P/E cycles, while newer technologies such as TLC and QLC store 3 bits and 4 bits per cell, respectively, further increasing density at the expense of fewer P/E cycles. Purity systems currently store data in consumer-grade MLC SSDs. Because of the way Purity uses flash, P/E ratings significantly underestimate real-world endurance. Section 5.1 discusses the discrepancy and describes the flash failure modes we have encountered.

2.2 Disk-based enterprise storage

Enterprise applications can require tens of thousands of random I/O operations per second (IOPS), but a typical performance hard disk provides only a few hundred IOPS. To work around this mismatch, enterprise computing environments incorporate block storage appliances that aggregate large numbers of disks behind a small number of controllers, making them accessible to applications such as scale-up databases, virtual machines, and file servers via standard protocols such as iSCSI and FibreChannel. For example, EMC's midrange VNX-7500 supports 1,000 spinning disks in its largest configuration, while EMCs VMAX

Table 1: Comparison of Purity and a disk array.

Metric	Purity	Disk	Improvement
Peak IOPS @ 32 KB	200K	65K	3.08×
Latency	1ms	5ms	5×
Usable Capacity	40 TB	25 TB	1.6×
Rack Units (RUs)	8	28	3.5×
Installation	4 hours	40 hours	10×
Power	1240 W	3500 W	2.82×
Annual Power Cost	$13,034	$36,792	2.82×
$/GB	**$5**	**$18**	**3.6×**
IOPS/RU	**25,000**	**2321**	**10.7×**
IOPS/W	**161**	**18.6**	**8.6×**
IOPS/$	**1**	**0.144**	**6.9×**

arrays leverage custom silicon to scale to even larger configurations.

Enterprise arrays typically consist of a large number of RAID-protected disks, a battery-backed RAM to accelerate commit of data and logs to "stable" storage, and a large data cache for the hottest data. Database engines and operating systems assume disk is slow, and thus cache their working sets in client (host) DRAM, allowing the total amount of cache to scale linearly with the number of clients and avoiding network latencies for cache hits.

Although they do not provide high-availability and other important enterprise features, consumer-oriented SSDs deliver 100–1000× more IOPS than enterprise spinning disks, suggesting that drastically simpler deployments should be able to match the throughput of these thousand-disk arrays.

This intuition is borne out in Table 1, which compares the published specifications for the VNX and a Purity array [46]. The numbers are from a description of our reference architecture for Oracle databases, which includes benchmark results and explains how to deploy Oracle RAC atop a Purity array.

Although the Purity array outperforms the disk array across all our metrics, the difference in cost/performance is striking: Purity cuts the cost of an I/O by about an order of magnitude while decreasing the cost of usable capacity. This has changed the way users manage storage: IOPS cost less than careful performance provisioning, so most customers simply throw hardware at the problem rather than spending their time manually configuring the disk array to ensure that important applications are not starved for IOPS.

2.3 High-performance scale-out storage

Enterprise storage only scales up to a certain point. As companies attempted to leverage it for Internet workloads, they discovered that the largest available systems were unable to service consumer-facing web sites [11].

Unlike most enterprise applications, these services did not require strong isolation or consistency, so homegrown scale-out storage systems with relaxed consistency models were sufficient. As with enterprise storage, these systems targeted spinning disk. Given the advantages of solid state storage, it is natural to ask if they are still necessary or cost-effective.

In this section, we compare high-performance scale-out disk storage with a large Purity array that targets data center storage consolidation. The FA-450 provides clients with 200,000 32 KiB peak IOPS, and can be configured with up to 70 TB of physical (250 TB effective) storage. We pessimistically assume object sizes of 32 KiB, which is larger than typical key-value store records.

Conveniently, the YCSB benchmarking study provides us with directly comparable numbers for a wide range of key value stores [16]. They all achieve approximately 1600 ops/s per machine in the best case, which is less than 1% the throughput of a FA-450.

Where sufficient information is available, we repeat this calculation in Table 2 by looking at published numbers from real-world disk-based scale-out deployments, and, again, estimate that one Purity array can replace hundreds of disk-based machines. We use disk-based key-value store hardware requirements and scalability bottlenecks from the literature, focusing on 2010–2014, when SSD storage deployments began to become practical.

We believe our estimated 100–250:1 consolidation ratios apply to a wide-range of disk-based storage solutions, and our customers have reported similar ratios (Section 5.4).

Recall that application designers were forced to move to key-value storage systems because their applications were too demanding to be stored in a scale-up system. However, the cost of managing a large scale-out cluster is significant, pushing these applications to consolidate onto a small number of monolithic clusters.

The deployments in the table are meant to serve dozens to thousands of applications. The vast majority of such applications require only a small fraction of a scale-up flash storage system. This eliminates the need for extremely large consolidated storage infrastructure, and allows applications to migrate to simpler scale-up solutions that provide strong application-level consistency.

The logistical improvements associated with 100:1 consolidations are significant. It takes approximately four hours to unpack a Purity array, physically install it and finish provisioning volumes. We help customers run on-site evaluations of our products on their own workloads before making a purchase. This is simply not practical for custom-built scale-out storage clusters.

The transition from disk to flash is driving aggressive storage consolidation. Even the largest scale-out applications from just a few years ago are now better served by a few scale-up machines than by large-scale clusters. In turn, this is improving application semantics, reducing costs, and enabling new classes of applications.

3. DESIGN PRINCIPLES

The previous section describes the approaches taken by existing systems and the storage trends that led us to break from those designs. Purity was designed around several core principles that are based upon the use of SSDs rather than disks for the underlying storage. These design principles are used throughout Purity's implementation.

To recap, Purity exposes virtual block devices over local networks. Systems such as databases and virtual machine infrastructure use these block devices to store data. These applications run in highly-available environments, and standard protocols such as iSCSI allow them to coordinate access to a given block device amongst multiple machines.

Simple applications run in primary-secondary mode. One machine obtains an exclusive lease on the block device. If the primary fails, the secondary obtains the lease and continues operation. Higher-level issues (such as client session management) are handled by the applications deployed atop Purity.

Table 2: Key-value store deployment sizes and estimated Purity FA-450 consolidation ratios. Purity arrays improve data center storage densities by orders of magnitude, and can service multiple scale-out applications.

Service	Scale	Year	Scope	Apps	Nodes	≈FA-450's	$\frac{Nodes}{FA-450}$	$\frac{Apps}{FA-450}$
PNUTS [15, 16]	1.6M op/s (design target)	2010	Data center	-	1000	8	120	-
Spanner [18]	1-10 PB (design target)	2010	Data center	300	10^3-10^4	4-40	250	7.5-75
S3 [31]	1.5M op/s (peak)	2013	Global	-	-	6*	-	-
DynamoDB [32]	2.6M op/s (mean)	2014	Region	-	-	13	-	-

*S3 mixes large and small objects; our estimate ignores large objects.

In systems such as parallel databases, multiple machines simultaneously access the same block devices. The clients manage issues such as cache coherence, often by leveraging block-level leases on the underlying storage.

In addition to carefully managing faults, Purity generates and manages large amounts of metadata. Despite the relative simplicity of block-oriented storage, users demand high-level volume management features such as snapshotting, off-site replication and security. Furthermore, Purity relies on compression and deduplication to provide storage capacities at costs comparable to hard disks. Purity stores the resulting metadata in log structured relational structures.

For these reasons, Purity includes from-scratch implementations of locking and indexing primitives that should be familiar to database implementors. Indeed, Purity incorporates a wide range of recent advances in database implementation techniques.

3.1 Perform extra reads to conserve space

Flash capacity is more expensive than disk, requiring that we reduce the total amount of storage actually required as much as possible. Our system implements both compression and deduplication on the fly, cutting storage requirements by a factor of 3–10× or more, depending on the workload, while still providing very high performance. Purity's deduplication and error correction coding techniques incur additional random reads, but our use of SSDs makes the cost acceptable [54]. Purity employs compression judiciously, reducing the amount of data transferred between storage and RAM [34]. Update-in-place systems must align compressed data to fixed boundaries to allow future updates. In contrast, like other log-structured systems, Purity is able to pack data tightly, leading to simpler, more efficient compression techniques [12, 50, 53, 56].

3.2 Leverage logical monotonicity

Purity represents all persistent data as immutable facts (tuples). Deletions are represented as immutable retractions of facts, e.g., "X is no longer true." Facts incorporate sequence numbers, allowing us to represent arbitrary insertion and deletion schedules. This makes it easier to reason about complex parallel schedules and failure scenarios.

Over time, machines have become increasingly parallel, and SSDs have drastically lowered the price of I/O operations. However, the cost of synchronization primitives has remained relatively flat. In order to leverage the computational power and storage bandwidth of modern systems, we need to be extremely careful to avoid unnecessary synchronization across CPU sockets and cores. Similarly, Purity arrays contain multiple storage devices and caches. For performance, we process updates in parallel whenever possible.

Since insertions and deletions simply insert immutable facts into Purity relations, they are idempotent and commutative. This allows us to use relaxed synchronization and ordering primitives without sacrificing correctness. It also enables a novel approach to deletion we call *elision*. Elision allows us to atomically delete all tuples that match a predicate, and is discussed in more detail in Section 4.10.

Unlike standard key-value stores, Purity stores retractions in a separate table from normal facts, and readers are allowed to run in a relaxed consistency mode that simply ignores retractions, allowing them to observe tuples that no longer exist. Similarly, confused or lagging writers may safely reorder inserts and deletes to no ill-effect.

Our approach to insertions and deletions is similar to the programming methodology espoused by the CALM theorem [3], which states that eventually consistent programs are exactly those that can be built by monotone logic. Logically monotone programs are those in which facts that become true never become false. Sequence numbers violate this property because the current sequence number changes over time, so they act as a controlled source of non-monotonicity.

Purity uses this to implement semantics stronger than eventual consistency, such as total ordering of writes to each sector, atomic snapshots and crash consistency. Similarly, we can implement linearizable updates by having write requests wait until their corresponding sequence number has propagated throughout the system. See the CRON principle for a more theoretical grounding of this approach [4].

As far as we know, Purity is the first data management system to uniformly apply these principles across every layer of the stack. Indeed, Purity was implemented as the CALM conjecture and CRON principle were published. As we describe Purity's implementation in Section 4, we present practical applications of these ideas in more detail.

3.3 Carefully manage writes

Although SSDs provide higher raw write bandwidth than hard disks, SSD writes come with hidden costs. They wear out the underlying flash, and, despite good average performance, flash translation layers behave erratically when exposed to random writes [43]. In theory, FTL behavior should improve, but a few issues prevent this in practice.

First, consumer use cases drive FTL designs. We have characterized and validated SSDs from a range of vendors over the last five years. Enterprise performance frequently regresses, even as consumer benchmarking scores improve.

More fundamentally, our x86 controller has a global view of the workload and much more computational power than the SSD FTL, allowing it to apply optimizations and make global decisions that the drives are incapable of.

Purity manages writes in a few different ways. As discussed above, Purity uses log-structured layouts and data reduction techniques to translate application-level random

writes into compressed sequential writes. This simplifies the job of the FTL, though we still need to verify that each SSD revision we ship behaves predictably when confronted with sequential writes. Also, we schedule reads to avoid SSDs that are performing writes, reducing read latency variance.

Materialized aggregates are a significant source of writes in Purity. Queries such as *"How many references are there to this block?"* are common. Instead of keeping precise answers in flash, we keep approximations and then fix them up by issuing additional reads at runtime.

3.4 Virtualize resources

Purity provides an update-in-place abstraction atop log-structured storage primitives. This level of indirection enables optimizations such as compression and deduplication, and it also breaks the direct correspondence between application disk blocks and physical disk blocks. This allows us to choose data layouts that directly support features such as volume cloning and snapshots. When implemented carefully, using these abstractions to virtualize volumes adds negligible runtime overhead. We use mediums to implement such functionality.

4. IMPLEMENTATION

Based on the hardware trends and design principles outlined in the previous sections, we implemented a scale-up storage appliance capable of servicing all but the largest workloads. A scale-up appliance based on commodity servers combined with consumer-grade SSDs provided the right balance of cost, performance, reliability, and capacity. Our challenge was to implement Purity in an environment that could easily become CPU-bound, not I/O bound, as legacy disk-based systems are.

Significant software development is required to use performant, low-cost consumer SSDs in Purity. For example, we use several data reduction techniques to increase the effective system capacity and we trade SSD throughput performance for consistently low I/O latency. Likewise, the hardware platform provides multiple paths to storage to enable software to maintain 24/7 availability, even during systems upgrades and hardware failures. Finally, fully virtualizing and abstracting the storage resources enables low-overhead features such as snapshots and image cloning, which also dovetails well with our deduplication features.

This section starts at the lowest levels of Purity's design, and describes how we progressively layer higher levels of the stack upon the foundation we have already described. Purity borrows concepts heavily from the literature; we cite existing work when possible, and focus on novel aspects here.

4.1 Flash Array hardware

Pure Flash Arrays consist entirely of commodity hardware (Figure 2). Each array contains two controllers, which are standard dual-socket x86 machines. Like most commodity servers, these machines include standard redundancy features, such as dual power supplies. Purity provides access to virtual block devices in *active-active* mode, which means that clients treat network ports on the two servers interchangeably, allowing clients to transparently redirect traffic in the event of a controller failure.

However, only one Purity controller serves traffic at a time. The other forwards requests to the primary via internal InfiniBand connections. The InfiniBand links are the

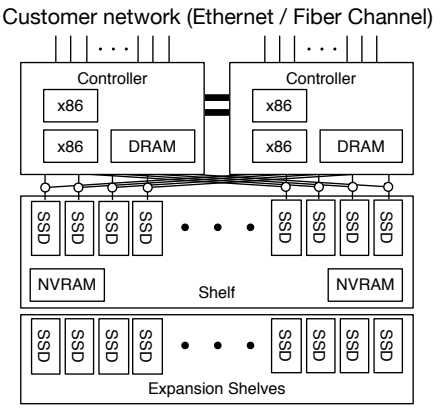

Figure 2: Flash Array consists of commodity components. At failure, interposers transfer control of the SSDs to the secondary. The controllers are stateless.

throughput bottleneck in our current arrays. As a side effect, Purity latencies improve slightly when the secondary fails. This is preferable to degrading application performance during failures, which risks losing availability.

Flash Array appliances connect to customer networks via multiple links. Currently, our largest model provides 7 GB/s of storage bandwidth via up to 12×16 Gb/s fiber channel links, 12×10 Gb/s Ethernet iSCSI links, or various combinations of the two technologies. All Flash Arrays include network replication ports.

The controllers connect to *shelves* that contain between 11 and 24 MLC drives. We use consumer grade SATA SSDs with SAS interposers. The interposers connect each drive to both controllers. When one of the controllers fails, the remaining controller immediately gains access to the SSDs.

In addition to the SSDs, shelves contain NVRAM devices for logging frequently updated structures. The NVRAM is part of the shelf so that the controllers themselves are stateless and easily replaced.

When Purity launched, NVRAM was not widely available. Instead, we use an extremely high-performance SLC flash part that provides us with bounded latency, and many more P/E cycles than the rest of the flash in the system. Below, we use the word "NVRAM" to refer to this device, since its performance characteristics are closer to NVRAM than to commonly deployed flash.

4.2 Physical storage layout

Purity stores data in *segments* (Figure 3) each of which is striped across multiple SSDs, using a high-performance software implementation of Reed-Solomon [45] to ensure that no data is lost even if two SSDs fail. In addition, Purity can leverage the parity pages *within* each SSD; flash translation layers can quickly recover a single corrupted page without the need to read data from the other drives in the segment.

A segment is composed of one *allocation unit* (AU) from each of the drives across which it is striped; the specific drives may be different for each segment and are chosen when the segment is written. Allocation units, which are 8 MB in current systems, are the minimum allocation granularity for an SSD; future AUs could be different sizes. Within a segment, each SSD is written atomically in *write units*, which are currently 1 MB. A horizontal stripe of write units across the segment, called a *segio*, accumulates compressed

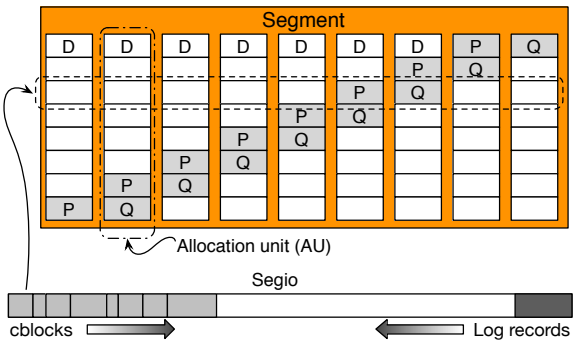

Figure 3: Data layout in Purity. Segios are parity protected and striped across devices. Data accumulates from the front of the segio, and log records (tuples) are written from the back. When the segio is full, it is flushed to the SSDs.

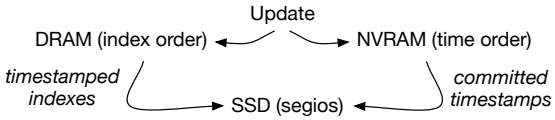

Figure 4: Monotonic write-ahead logging implementation. The segio layer joins the commit stream with a stream of time-bounded indexes, and then trims the DRAM and NVRAM (not shown).

user data from the front, and accumulates log records from the back (Figure 3). When the two sections meet, the segio is completed and marked for flush to SSD. However, Purity is very flexible; a segio may only contain user data or log records, as well.

Writing log entries is less expensive than re-writing entire data sections. However, it still involves writing multiple megabytes of data, and is too high-latency to be useful for committing application writes. Instead, we commit writes to NVRAM, and then write them back to segios (Figure 4). The commit path implementation is monotonic; commits are expressed in terms of immutable facts that flow through the system. We describe the benefits of this below, in the discussion of recovery and pyramids.

In the remainder of the paper, we ignore the existence of segios, and refer to them as segments instead. Segios are an optimization that allows us to flush data to SSDs in smaller chunks, so distinguishing between segios and segments would complicate our discussion considerably, to little benefit.

4.3 Recovery

At recovery, we scan segment headers. This allows us to discover facts that were persisted shortly before crash, since segment headers contain information about the sequence numbers they encode. Segments are self-describing, and early versions of Purity simply checked each header in the system at boot.

Segments are fairly large, and random SSD reads are inexpensive, so this approach worked reasonably well. However, it is linear in the size of the Purity array's storage. Although this scan was never the dominant cost of a cold boot, controller failover performs a segment scan, and its performance is critical. During failover, the Purity array

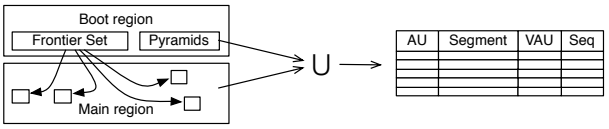

Figure 5: Purity always writes large sequential chunks, and therefore scatters log records amongst user data. Instead of scanning all AUs for log records, we maintain a *frontier set*—the list of AUs that we plan to use soon. Purity stores all its data as immutable facts, so recovery consists of computing set unions over the discovered tuples.

cannot respond to I/O requests, and typical clients time out after 30 seconds. Therefore, to avoid application failures during controller failure, we have to guarantee that recovery will complete in under 30 seconds.

It is tempting to synchronously ship the log to the secondary during normal operation. However, doing so would cause the system to become unresponsive if the secondary failed. Instead, the primary controller asynchronously warms the cache of the secondary, reducing the total amount of I/O required for failover. With this optimization, the scan takes 12 seconds on recent hardware; combined with other, less expensive, aspects of recovery, this brought previous versions of Purity dangerously close to the 30 second timeout.

Purity divides storage into two regions. The *main region* contains application data, medium tables, deduplication information, and the other metadata Purity maintains during forward operation. The *boot region* is a tiny percentage of the total storage, and contains the locations of the relations and allocator state for the main region.

Instead of scanning all segment headers to find log records at failover, we store a *frontier set* in Purity's boot region (Figure 5). This stores the set of free AUs that are ready to be reused. We constrain the allocator to allocate AUs from the currently persisted frontier set, allowing us to limit log record scans to segments in the frontier set. As an optimization, we also store *speculative* and *transition* sets that contain approximations of future frontier sets. These usually contain adequate approximations of future frontier sets, allowing us to rewrite the frontier set less frequently.

An early prototype took this idea one step further, and stored portions of the frontier set in log records. Ultimately, this was untenable: trimming segments could break the chain of log records, forcing us to block space reclamation on log updates. This was complicated and deadlock prone, so we abandoned chained frontier sets, and instead use the scheme in Figure 5.

In practice, frontier set writes consist of well under 1% of writes, and reduce startup scan times from 12 s to 0.1 s on our current arrays. This further reduces the impact of controller failover on applications, and keeps us well below the 30 second hard limit.

Scanning NVRAM and the log records provides us with a set of facts that might be missing from the persistent copies of Purity's indexes. We leverage logical monotonicity, and simply insert these facts into the indexes in question. Since all tuples in Purity are immutable facts, inserting stale or duplicate records is harmless.

Recovery must be robust against corrupted pages and missing drives. Since log records are stored in segments, they are automatically protected by Purity's ECC scheme.

4.4 I/O scheduling

Purity aims to service all application requests with less than a 1 ms tail latency. To this end, we implemented Purity in event-driven C++, with a single thread pinned to each core. Event handlers run with affinity to NUMA groups to avoid cache synchronization traffic. We carefully instrument and bound the latencies of each request handler in the system, preventing one application or background request from monopolizing an x86 core. Whenever feasible, the event handlers are lock-free, further reducing the chances that requests will lead to CPU stalls and impact the latency of unrelated requests.

Before shipping SSDs to customers, we carefully qualify both the physical hardware, and also the firmware revisions that are allowed into our appliances. This allows us to avoid drives with unpredictable request latencies.

As is common practice with other large-scale storage systems [19], we measure the latency of each request and use Reed-Solomon to reconstruct requested data whenever a request takes longer than our 95^{th} percentile latency.

Despite all of these efforts, the SSDs still cause stalls, and, as of Purity release 4.0, slow SSD requests are the leading cause of high latency application requests.

Since we cannot avoid using SSDs and FTLs with unpredictable latencies, we have to work around the underlying hardware. The vast majority of slow SSD reads happen while the SSD is in the process of servicing segment write requests. Therefore, we try to avoid writing to more than two SSDs per ECC group at the same time, and treat SSDs that are in the process of writing data as though they have failed.

Instead of reading from the busy SSDs, we rebuild the data using parity data. Purity uses $7 + 2$ Reed-Solomon encoding, with each segment written across a (potentially different) set of the 11 drives in a write group. In the worst case, we service $\frac{2}{11}$ of reads in this way, each of which requires reading 7 drives, increasing costs by $7 \times \frac{2}{11} \approx 1.3\times$ for write-heavy workloads.

With these optimizations, Purity provides an order of magnitude lower tail latencies than the best disk or hybrid disk-flash appliances, more than compensating for the throughput penalties.

4.5 Mediums

The vast majority of data stored in Purity is compressed block data from applications. Unlike metadata, this data is stored without regard for locality or careful ordering by key. Garbage collection strategies for unordered storage are well-understood, and are significantly less expensive than approaches for ordered indexes [48, 52, 58].

Purity exports a volume-based interface to users: blocks are addressed by $\langle volume, offset \rangle$. Internally, however, Purity maintains a single mapping structure for *all* user data, regardless of the volume on which it is stored. Rather than associate data with a particular volume, Purity logically stores user data in containers called *mediums*; each user-visible data block is thus accessed using a $\langle medium, offset \rangle$ key.

The medium table (Figure 6) is used to identify all possible keys that might be used to find the value for a given $\langle volume, offset \rangle$ lookup. Figure 6 shows examples of snapshots (14 is a snapshot of 12, 20 is a snapshot of 18, and 22 is a snapshot of 21) and clones (15 and 18 are clones of part of 12). In medium 22, it also shows that the table facilitates shortcuts: since blocks 500–999 haven't been written since the snapshot was taken of 12, medium 22 can refer *directly* to 12, allowing for fewer lookups. Purity uses additional flags to optimize lookup and reduce the number of references that need to be made to the table, including flags that indicate when no blocks have been written to a range and flags that terminate recursion.

Source		Target		
Medium	**Start:End**	**Medium**	**Offset**	**Status**
12	0:3999	none		RO
14	0:3999	12	0	RW
15	0:999	12	2000	RW
18	0:999	12	2000	RO
20	0:999	18	0	RO
21	0:999	20	0	RO
22	0:499	21	0	RW
22	500:999	12	2500	RW
22	1000:1999	none		RW

Figure 6: Information maintained for each medium.

4.6 Cblocks

Existing storage protocols dictate a minimum block size of 512 B, which we use as the minimum unit of deduplication and compression in Purity. Mediums store blocks of application data in a compressed format called a *cblock*. Each medium contains a set of cblocks that act as a patch to be applied to the underlying medium. Although mediums could be stacked arbitrarily, Purity's garbage collector rewrites trees of mediums in a flattened form so that application reads never have to access more than three cblocks.

Most applications perform I/O operations much larger than 512 B: across our customer installations, I/O requests are $\approx$ 55 KiB on average. Most flash and hard disk arrays force administrators to set a single optimal block size per volume. Such systems attempt to align application pages with RAID stripe geometries. This does not work well in practice, since most systems perform I/Os of multiple sizes. For example, customer telemetry data tells us that Oracle aggressively prefetches pages, leading to a minimum effective I/O size well above the configured database page size. Also, it chooses different transfer sizes for log and data accesses, forcing customers using other arrays to configure multiple virtual block devices with different RAID geometries for each database instance.

We went to great lengths to avoid exposing tuning knobs to customers. Purity is often used to consolidate multiple workloads, making it extremely difficult to manually tune. Also, Purity's simplicity of installation and administration is an important selling point to most customers.

Instead of having administrators guess optimal block sizes, Purity infers optimal transfer sizes by observing I/O requests. The vast majority of read requests access data using the same alignment and block size as the write request that created the data in question. Although Purity cblocks can represent as little as 512 B of data, they are sized to match application writes, up to 32 KiB. This improves compression ratios and also access times. Since the data is likely to be retrieved with the same granularity as the write, small read requests generally retrieve a single cblock.

4.7 Deduplication

Purity tracks deduplication blocks at 512 B granularity, but only records the hash value of every eighth block written, using hashes no larger than 64 bits. While only 1/8 of all hashes are recorded, *all* hashes are looked up. If a block's hash value matches one already stored, Purity performs a byte-level check to confirm that the block is a duplicate, allowing us to use shorter hashes with a collision rate of 10^{-6} or worse, since collisions only cost a data block comparison and do *not* affect correctness. We then use the duplicate block as an "anchor" to find other, nearby duplicates. This approach detects most duplicate sequences of at least 8 blocks (4 KiB), regardless of alignment, and is a reasonable tradeoff between occasionally missing duplicates and maintaining a small index of block hash values. For duplicates, Purity records a mapping from the new logical address to the $\langle segment, offset \rangle$ of the existing data.

Purity performs inline deduplication, allowing it to detect duplicates before writing the duplicate value somewhere on SSDs. We use a number of heuristics to improve inline deduplication performance. For instance, inline deduplication only checks for duplicates of recently written data and frequently deduplicated data; in practice, inline deduplication finds most duplicates. Later, as garbage collection scans SSDs in the background, it performs a more expensive deduplication pass, and deduplicates the blocks we did not have time to process earlier. Garbage collection also attempts to segregate deduplicated blocks into their own segments, since blocks with multiple references are less likely to become completely unreferenced due to overwrites.

4.8 Log structured indexes

Purity stores its metadata in relations that are indexed by zero or more log-structured merge trees called *pyramids*. Pyramids are tightly coupled to our NVRAM and segment persistence schemes.

Insertions are handled by *persist* operations that assign a sequence number to a batch of tuples and insert them into NVRAM. Each of these batches is also cached in DRAM, where it is sorted and indexed in key order.

The segment (segio) writer (Figure 4) subscribes to two event streams. One contains a list of sequence numbers that have been persisted to NVRAM; the other contains *patches*, index-ordered data that is annotated with sequence numbers. The segment writer enforces write-ahead logging by ensuring that indexes are not written to segments until after the corresponding sequence numbers are persisted to NVRAM.

Patches are analogous to *levels* or *components* in other LSM-Tree implementations, and describe differences between the previous version of the pyramid and the new one. We track key ranges and sequence numbers for each patch. *Merge* operations combine patches with contiguous sequence numbers, and *flatten* operations replace old patches with the newly merged ones.

The implementation of merge and flatten are idempotent, allowing everything below the top level of the pyramid to be implemented in a lock-free fashion. Furthermore, merge and flatten operations are always safe, ensuring that index maintenance is deadlock free, and simplifying recovery from failures during in-progress merge operations. LSM-Tree merge strategies are covered in detail elsewhere [10, 35, 39, 41, 44, 47, 51]; we refer readers to those discussions for more information about LSM-Tree algorithms and implementations.

In addition to the medium table and elide tables described below, important Purity tables include the segment table, deduplication tables, and link tables that manage relationships between segments for garbage collection.

4.9 Metadata layout

Purity stores metadata in tables that are compressed using formats similar to those used in column stores [1, 33, 50]. Each page has a "dictionary" header used to compress or decompress the tuples in the page. The dictionary has an entry for each tuple field that contains *bases* $b_0, \ldots, b_{B-1}$ and a width W for the offset from the base encoded in each tuple. A tuple value $v = b_x + o$ is then encoded as $\langle x, o \rangle$, where b is a $\lceil \lg B \rceil$-bit integer and o is a W-bit integer. Note that W can be 0, as can $\lceil \lg B \rceil$. This encoding is a highly efficient variant of run length encoding, allowing us to include extra fields that aren't needed: as long as their value is the same for every tuple, the extra fields take up no space. Another advantage is that the system can scan a page for a particular value without decompressing the tuples, all of which are the same length in bits. Instead, we treat the page as a bit stream and look for a particular bit pattern based on the compressed representation at the appropriate offset from the start of each tuple.

As a special case, Purity also implements extremely efficient range encoding schemes. These are used to bound the size of the elide tables.

4.10 Elision

Most LSM-Trees implement delete by inserting *tombstone* records into the tree. Tombstone records contain the key of the record to be deleted. Upon encountering a tombstone, LSM-Trees lookups ignore any older tuples with the same key. When tombstones reach the oldest level of the tree they can be discarded.

Purity takes a different approach. Each table has a set of *elide rules* associated with it, which refer to a set of *elision tables*, and explain the table's deletion policy. Typical tables have a single elision table, which is indexed by an auto increment key or sequence number. Tuples in this table represent deletion predicates, and Purity treats all tuples that match the predicate as though they have been deleted. Unlike tombstones, elide records do not need to be keyed in the same way as the table.

Mediums are the motivating example for elision: to drop all of the pages in a given medium, we simply insert a single record into the appropriate elide table. Traditional techniques, such as hierarchical locking [26] would obtain medium-level locks. We wish to avoid such synchronization primitives in Purity, since the data in question is accessed using lock-free operations. Also, inserting an elide record is much less expensive than deleting one cblock at a time.

As with all other tuples in Purity, elide records are immutable facts. Elide operations are idempotent and do not need to be protected by a locking protocol. Requests read the base data as normal. In circumstances in which they must return failure if the records have been deleted, they query the elide table for predicates that match the records they are about to return, and filter their results accordingly. Such readers simply read immutable data, and do not need to acquire locks or synchronize with other threads.

Purity's garbage collector understands elide records, and consults elide tables as part of the merge process. Tuples that have been deleted are immediately dropped. This allows space to be reclaimed more quickly than tombstone-based approaches, which must wait for the tombstone to propagate to the level that stores the data to be deleted. Timely space reclamation is important, because it allows us to immediately begin reclaiming segments after snapshots and other large structures have been dropped. This reduces the risk of deadlock due to running out of space inside the garbage collector.

However, it leaves us with a new problem: We must make sure that elide records do not accumulate over time, permanently leaking space. Rather than introduce another deletion mechanism, we encode elide records as ranges, and merge contiguous ranges. In the worst case, this forces us to store one range for every elide tuple. Since we use dense, monotonically increasing numbers as the keys for elide tables, the ranges collapse rapidly, preventing the elide tables from growing without bound. Moreover, the number of elide ranges can be no larger than the number of *valid* tuples, which itself is limited by the amount of physical or logical storage in the system, depending on the table. Since we never reuse sequence numbers, there is never a need to remove an elide record to allow new records to be written.

5. CUSTOMER DEPLOYMENTS

In Section 2, we contrasted Purity's design goals with those of state-of-the-art performance disk systems. Here, we talk about real-world deployments of Purity, and compare to currently deployed solutions.

Although an alpha version of Purity began shipping in 2010, it began to hit critical mass in 2013 and 2014. At that point, Purity had mature support for the most important enterprise storage features.

5.1 Reliability

We sell service plans that include continuous health and telemetry monitoring of customer arrays, as well as installation services for failed components. In areas with sufficient customer density, our operations team offers customers a four hour hardware replacement SLA.

When an array phones home with a hardware failure, we dispatch a technician to pick up the necessary replacement equipment from a local warehouse, arrive at the customer site, and install the part.

Telemetry data provides us with detailed insights into customer workloads, including I/O request rates, request sizes, volume sizes, and deduplication ratios. We do not expose tuning knobs to customers. Instead, if a Purity array's performance begins to degrade, we perform a root cause analysis (remotely, or on site), and contact the customer before taking action to mitigate the problem. Along with Purity's architecture, this proactive approach to array maintenance has led to 99.999% availability across our customers' production arrays.

For example, a bug in our garbage collection routines was causing suboptimal deduplication performance at a few of our customer sites. We detected the issue, patched and validated a new version of Purity, and then contacted the impacted customers so they could install the patch at a time that minimized the risk of disruption to their production applications.

SSDs are much more reliable than we anticipated. At this point, Purity arrays have replaced hundreds of EMC VMAX installations. Across all Purity installations, only two SSDs have failed. Of course, this number will increase over time, but the chances that a Purity array will encounter a failed drive are extremely low. Instead, components such as fans and DIMMs account for the lion's share of hardware replacement calls.

Unlike hard disks, which fail catastrophically, SSDs simply begin losing pages as they age. Even this is rare, as most enterprise workloads are read heavy, and typical customers never approach the P/E ratings of the consumer MLC drives we sell. We are confident enough in this statement to offer free SSD replacements to customers that actually manage to wear out SSDs in arrays covered by support contracts.

In the process of validating Purity, we built an array out of worn-out flash. We first used synthetic data to overwrite drives until they reached their rated number of P/E cycles. Next, we formatted the drives using Purity, and ran stress tests against the resulting system. We did not encounter any application-level hardware errors in this test.

In addition to all of the error correction techniques that Purity employs, it periodically scrubs the underlying storage to proactively detect data loss. Worn-out flash leaks charge faster than new flash, and P/E ratings are calculated under the assumption that the SSD will be powered off for a year.

Periodically scrubbing and rewriting data ensures that the worn-out flash is rewritten more frequently than the P/E calculations assumed, allowing our arrays to run well past rated wear out. In practice, we do not let customer arrays reach such a state, and instead, proactively replace the few worn SSDs we encounter in production.

5.2 Database deployments

Our customers routinely deploy databases atop our arrays, and we have published detailed descriptions of reference installations of popular options, including Oracle, SQL Server and MongoDB. At the high end, customers deploy parallel, shared-disk database solutions such as Oracle RAC against one of our arrays. However, it is much more common for customers to deploy dozens or even hundreds of independent database instances on top of each Purity array.

Instead of recapping details about these installations, we discuss the implications of all-flash arrays on well-known database performance models.

5.2.1 Transaction rollback rates

Stateless clients scale linearly with request latencies and parallelism. Typical Purity latencies are $\frac{1}{10}$ those of disk based solutions, reducing the number of concurrent requests customer compute boxes must track. Similarly, Purity provides much higher throughput than disk-based systems, allowing more requests to be processed per unit time. Taken together, these effects can significantly reduce memory usage on modern database application servers. However, such servers run stateless applications, and are typically compute bound.

Contrast the case of stateless servers with database transactions, which must roll back in the face of conflicts. As latencies increase, so too does transaction concurrency and runtime, increasing the probability of transaction rollbacks. It is well known that these effects lead to non-linear increases in rollback rates [25].

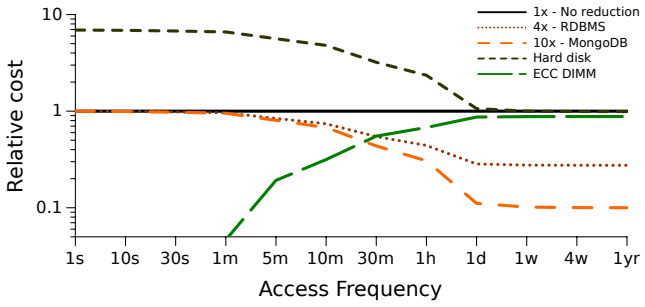

Figure 7: The relative cost of storing data in Purity arrays, disk arrays and main memory. Data reduction significantly changes the tradeoffs between disk, flash and RAM.

A number of mitigating techniques exist, including running transactions at lower consistency levels, rewriting applications from scratch without transactions, and purchasing extremely expensive database licenses.

We frequently encounter customers facing this dilemma, and offer an alternative solution: Purity decreases request latencies by an order of magnitude, potentially reducing rollback rates by more than 10×. In practice, this allows customers to avoid upgrading to more complicated database infrastructure, or even to migrate from proprietary solutions to simpler open source systems such as Postgres and MySQL.

Interestingly, these effects cause customers to underestimate Purity's potential throughput improvements. For instance, given a production database running at 60% CPU time and 40% I/O wait time, one would not expect more than a doubling of throughput due to storage improvements. In contrast, replacing disk-based storage with Purity often leads to speedups closer to 10× in such scenarios.

5.2.2 *The five minute rule*

The database literature is well-poised to answer another common customer concern: *Given the rapid changes to storage hardware, what should we purchase?*

The "five minute rule" provides a rule of thumb for provisioning storage systems [24]. Traditionally, data accessed more often than once every five minutes belonged in RAM, and colder data belonged on disk.

The cost of storing data can be divided into the price of the capacity the data occupies, and the price of the device time it takes to transfer the data to storage. Small, fast devices, like RAM are a good fit for hot data, and large, slow devices are better for cold data.

Using Table 1, and information from customer deployments, we computed relative costs for Purity and hard-disk arrays: Typical Purity deduplication ratios are 3–8× for relational databases and 10× for document stores such as MongoDB. I/Os are 55 KiB on average. We assume a price of $1000 for 64 GiB of ECC LR-DIMM RAM.

Based on the results in Figure 7, we present the following rules of thumb:

1. Performance disk is dead.
2. Without data reduction, store everything you can afford to lose in RAM (or non-redundant direct attached disks).
3. With data reduction, never cache data that is accessed less frequently than every half hour. It is cheaper to store it in fault tolerant storage than RAM.
4. Important data follows a ten-minute rule: For data colder than that, the cost of the second cached copy is comparable to the cost of accessing the storage.

With Purity, it makes sense to move warm data from RAM to persistent storage, reducing the amount of RAM necessary to support a given set of compute nodes. This has second-order benefits. High-density, low profile DIMMs are roughly 50% more expensive than physically larger models. Reducing memory capacities can allow customers to purchase less expensive processors with fewer memory channels, and to fit more machines into a given number of rack units. These effects increase the incremental cost of storing data in RAM in ways we ignored in our computation.

5.3 Virtualization environments

Virtualization is another common use case for Purity, and can be divided into two classes of workloads: server and desktop virtualization. Server virtualization allows companies to reduce the number of physical machines required to service their backend infrastructure, and has side benefits, such as facilitating hardware upgrades and centralizing backup and other storage management tasks.

Customers often run hundreds or thousands of virtual machines atop a single Purity array. Many are database servers, while others run database applications, mail servers, file servers and other such services. Typical deduplication ratios for these systems are 5–10×.

Scale-out database applications are commonplace, and typically run atop stateless clones of the same operating system image. Purity can efficiently clone such instances. Alternatively, as updates are rolled out to the cloned images, Purity aggressively deduplicates data modified by the updates. Effects such as these sometimes lead to much higher deduplication rates.

We see similar effects with virtual desktop deployments that manage thousands of similar virtual machine images, which are in turn used to power cash registers, terminals, and standardized software environments. Depending on the exact use case, it is possible to achieve deduplication ratios in excess of 20× with such workloads.

5.4 Cloud service providers

Section 2.3 estimates storage requirements of large scale-out storage installations, and concludes that Purity arrays and scale-out performance storage clusters have comparable capabilities at drastically different price points. Indeed, we have begun selling Purity arrays to cloud service providers.

Our customers report data center footprint reductions on the order of 40% when moving from spinning disk to Purity, with the majority of the remaining footprint allocated to compute nodes. Client-visible request latencies improve dramatically, as does throughput. In a representative deployment, virtual machine provisioning times went from nine minutes to forty five seconds (12× faster), and the customer plans to back 500 relational database instances with a single FA-420 array.

Based on customer experiences and proof of concept installations, we have found that an eight-rack-unit Purity appliance provides service providers with similar capabilities as 160 rack units of white-box OpenStack storage nodes. This

is similar to the footprint reductions reported by our customers, as well as the estimates we presented in Section 2.3.

Cloud service providers resell fractions of Purity arrays to customers that ultimately provision their own resources. Purity includes automation tooling that is appropriate for this task, and has been certified to work well with the major infrastructure stacks in use at cloud service providers. As with our other use cases, customer-facing simplicity and usability are major differentiators here. Providers resell Purity storage to a large number of customers. To the extent that the array automates performance tuning and provisioning, providers can focus manpower on other tasks.

Similarly, data reduction leads directly to improvements in the providers' cost per gigabyte, allowing our customers to charge rates comparable to their competitors' non-redundant cache storage rates.

In such environments, Purity's tail latencies and 99.999% availability have become key differentiators for cloud service providers. These benefits translate directly to improved application-level availability and performance, and cloud service providers are able to charge a premium for this.

6. RELATED WORK

Purity builds upon an extremely large body of related work, including LSM-Trees and other sorted [2, 10, 13, 39, 40, 44, 51], and unordered indexes [5, 37, 43].

Gupta, *et al.* use flash for content-addressable storage in a way that improves locality [30]. We do this for our index. While interesting, memory optimized indexes such as SILT have write amplifications that are too high for our purposes [41]. Debnath has published a number of other interesting small-memory flash structures [21–23]. Salus is another flash optimized persistent block store [57].

There are a large number of scale out distributed systems; we mention a few that target high performance workloads and modern storage hardware here. FAWN takes the opposite approach as Purity, and uses extremely inexpensive servers to build scale out flash storage [6]. Similarly, RAMcloud targets main-memory scale-out storage [49]. Like Purity, its implementation makes heavy use of logs and sequence numbers. Strata is another system in this category [17].

A number of key-value workload studies provided us with insight into scale-out storage performance [7, 16].

Our use of sequence numbers for consistency is reminiscent of the approach taken in Corfu, which implements a large, high-throughput distributed flash log [8]. Tango [9] layers persistent data structures atop this simple primitive.

We were informed by insights from a wide range of studies of flash hardware limitations. Flash performance is known to degrade in the face of random writes [29, 43]. Purity already uses horizontal and vertical erasure coding [38]; over time, we expect flash densities to increase at the expense of error rates, and adaptive techniques that handle variable error rates will probably be necessary [27, 28].

Chen, *et al.* found that coalescing writes extends flash lifetime [14]. We do so in our compression, deduplication and garbage collection algorithms, and use SLC SSDs (our "NVRAM") to absorb low latency writes that we cannot coalesce. Deferred writes have been applied elsewhere [20].

Deduplication studies show that, depending on workloads, whole-file deduplication performs well [42]. However, we target virtual machine deployments which are known to provide opportunities for deduplication [36], and therefore perform block-level deduplication. It is well known that database workloads compress well [34], and we use techniques from the column store compression literature [1, 33].

7. CONCLUSION

We have presented Purity, the first all-flash enterprise storage system to support compression, deduplication and high-availability. Hardware trends are disrupting existing storage system architectures, and enabling rapid consolidation of existing storage infrastructure as well as new classes of applications.

Purity leverages these trends, and a wide body of work from the literature to provide applications with order of magnitude performance and density improvements at comparable or lower cost per gigabyte than existing systems.

8. ACKNOWLEDGMENTS

Purity is the work of the entire Pure Storage team. Our conversations with Chas. Dye about customer deployments were invaluable, as were Costa Sapuntzakis's descriptions of Purity. Thanks to Marco Sanvido for the description and motivation for frontier sets, which were developed by Marco Sanvido, Rich Hankins, Nidhi Doshi and Huihui Cheng. We thank our shepherd, Carlo Curino, and Christopher Douglas for their feedback on earlier drafts of this work.

9. REFERENCES

[1] D. Abadi, S. Madden, and M. Ferreira. Integrating compression and execution in column-oriented database systems. In *Proc. SIGMOD Conf.*, pages 671–682, 2006.

[2] D. Agrawal, D. Ganesan, R. Sitaraman, Y. Diao, and S. Singh. Lazy-adaptive tree: An optimized index structure for flash devices. In *Proc. 35th VLDB Conf.*, Aug. 2009.

[3] P. Alvaro, N. Conway, J. Hellerstein, and W. R. Marczak. Consistency analysis in Bloom: a CALM and collected approach. In *Proc. CIDR*, pages 249–260, 2011.

[4] T. J. Ameloot and J. Van den Bussche. Positive Dedalus programs tolerate non-causality. *Journal of Computer and System Sciences*, 80(7):1191–1213, 2014.

[5] A. Anand, C. Muthukrishnan, S. Kappes, A. Akella, and S. Nath. Cheap and large CAMs for high performance data-intensive networked systems. In *Proc. 7th NSDI Symp.*, 2010.

[6] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan. FAWN: A fast array of wimpy nodes. In *Proc. 22nd SOSP Conf.*, pages 1–14, Oct. 2009.

[7] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *Proc. 2012 SIGMETRICS*, pages 53–64, 2012.

[8] M. Balakrishnan, D. Malkhi, V. Prabhakaran, T. Wobber, M. Wei, and J. D. Davis. CORFU: A shared log design for flash clusters. In *Proc. 9th NSDI Symp.*, Apr. 2012.

[9] M. Balakrishnan, D. Malkhi, T. Wobber, M. Wu, V. Prabhakaran, M. Wei, J. D. Davis, S. Rao, T. Zou, and A. Zuck. Tango: Distributed data structures over a shared log. In *Proc. 24th SOSP Conf.*, pages 325–340, Nov. 2013.

[10] M. A. Bender, M. Farach-Colton, J. T. Fineman, Y. R. Fogel, B. C. Kuszmaul, and J. Nelson. Cache-oblivious streaming B-trees. In *Proc. 19th Symp. on Parallel Algorithms and Architectures*, pages 81–92, 2007.

[11] E. A. Brewer. Lessons from giant-scale services. *IEEE Internet Computing*, 5(4):46–55, Aug. 2001.

[12] M. Burrows, C. Jerian, B. Lampson, and T. Mann. On-line data compression in a log-structured file system. In *Proc. 5th ASPLOS*, pages 2–9, Oct. 1992.

[13] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. In *Proc. 7th OSDI Symp.*, Nov. 2006.

[14] F. Chen, T. Luo, and X. Zhang. CAFTL: A content-aware flash translation layer enhancing the lifespan of flash memory based solid state drives. In *Proc. 9th FAST Conf.*, Feb. 2011.

[15] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. PNUTS: Yahoo!'s hosted data serving platform. In *Proc. 34th VLDB Conf.*, Aug. 2008.

[16] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Proc. 1st ACM Symp. on Cloud Computing*, 2010.

[17] B. Cully, J. Wires, D. Meyer, K. Jamieson, K. Fraser, T. Deegan, D. Stodden, G. Lefebvre, D. Ferstay, and A. Warfield. Strata: High-performance scalable storage on virtualized non-volatile memory. In *Proc. 12th FAST Conf.*, Feb. 2014.

[18] J. Dean. Designs, lessons and advice from building large distributed systems. Keynote from LADIS, 2009.

[19] J. Dean and L. A. Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, 2013.

[20] B. Debnath, M. F. Mokbel, D. J. Lilja, and D. Du. Deferred updates for flash-based storage. In *Proc. 26th IEEE Conf. on Mass Storage Systems and Technologies*, 2010.

[21] B. Debnath, S. Sengupta, and J. Li. ChunkStash: Speeding up inline storage deduplication using flash memory. In *Proc. 2010 USENIX Annual Technical Conference*, June 2010.

[22] B. Debnath, S. Sengupta, and J. Li. FlashStore: High throughput persistent key-value store. In *Proc. 36th VLDB Conf.*, Sept. 2010.

[23] B. Debnath, S. Sengupta, and J. Li. SkimpyStash: RAM space skimpy key-value store on flash-based storage. In *Proc. SIGMOD Conf.*, 2011.

[24] J. Gray and G. Graefe. The five-minute rule ten years later, and other computer storage rules of thumb. *ACM Sigmod Record*, 26(4):63–68, 1997.

[25] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *Proc. SIGMOD Conf.*, pages 173–182, June 1996.

[26] J. N. Gray, R. A. Lorie, G. R. Putzolu, and I. L. Traiger. Granularity of locks and degrees of consistency in a shared data base. In *Proc. IFIP Working Conf. on Modelling in Data Base Management Systems*, pages 365–394, 1976.

[27] K. M. Greenan, D. D. Long, E. L. Miller, T. J. E. Schwarz, S.J., and A. Wildani. Building flexible, fault-tolerant flash-based storage systems. In *Proc. 5th Workshop on Hot Topics in System Dependability*, June 2009.

[28] L. M. Grupp, J. D. Davis, and S. Swanson. The bleak future of NAND flash memory. In *Proc. 10th FAST Conf.*, Feb. 2012.

[29] L. M. Grupp, J. D. Davis, and S. Swanson. The harey tortoise: Managing heterogeneous write performance in SSDs. In *Proc. 2013 USENIX Annual Technical Conference*, June 2013.

[30] A. Gupta, R. Pisolkar, B. Urgaonkar, and A. Sivasubramaniam. Leveraging value locality in optimizing NAND flash-based SSDs. In *Proc. 9th FAST Conf.*, Feb. 2011.

[31] J. Hamilton. Why scale matters and why the cloud is different. AWS re:Invent, 2013.

[32] J. Hamilton. AWS innovation at scale. AWS re:Invent, 2014.

[33] A. L. Holloway, V. Raman, G. Swart, and D. J. DeWitt. How to barter bits for chronons: compression and bandwidth trade offs for database scans. In *Proc. SIGMOD Conf.*, pages 389–400, 2007.

[34] B. R. Iyer and D. Wilhite. Data compression support in databases. In *Proc. 20th VLDB Conf.*, pages 695–704, 1994.

[35] C. Jermaine, E. Omiecinski, and W. G. Yee. The partitioned exponential file for database storage management. *Proc. VLDB Endowment*, 16:417–437, 2007.

[36] K. Jin and E. L. Miller. The effectiveness of deduplication on virtual machine disk images. In *Proc. SYSTOR 2009*, May 2009.

[37] W. K. Josephson, L. A. Bongo, K. Li, and D. Flynn. DFS: A file system for virtualized flash storage. *ACM Trans. on Storage*, 6(3), Sept. 2010.

[38] Y. Kang and E. L. Miller. Adding aggressive error correction to a high-performance compressing flash file system. In *Proc. 9th ACM & IEEE Conf. on Embedded Software (EMSOFT '09)*, Oct. 2009.

[39] LevelDB: A fast and lightweight key/value database library by Google. https://code.google.com/p/leveldb/.

[40] Y. Li, B. He, Q. Luo, and K. Yi. Tree indexing on flash disks. In *Proc. 25th Int'l Conf. on Data Engineering*, 2009.

[41] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky. SILT: a memory-efficient, high-performance key-value store. In *Proc. 23rd SOSP Conf.*, Oct. 2011.

[42] D. T. Meyer and W. J. Bolosky. A study of practical deduplication. In *Proc. 9th FAST Conf.*, Feb. 2011.

[43] C. Min, K. Kim, H. Cho, S.-W. Lee, and Y. I. Eom. SFS: Random write considered harmful in solid state drives. In *Proc. 10th FAST Conf.*, Feb. 2012.

[44] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The log-structured merge-tree (LSM-tree). *Acta Informatica*, 33:351–385, 1996.

[45] J. S. Plank, K. M. Greenan, and E. L. Miller. Screaming fast Galois field arithmetic using Intel SIMD instructions. In *Proc. 11th FAST Conf.*, Feb. 2013.

[46] Pure Storage reference architecture for Oracle databases, 2014.

[47] RocksDB: A fork of LevelDB by Facebook. https://github.com/facebook/rocksdb/.

[48] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Trans. on Computer Systems*, 10(1):26–52, Feb. 1992.

[49] S. M. Rumble, A. Kejriwal, and J. Ousterhout. Log-structured memory for DRAM-based storage. In *Proc. 12th FAST Conf.*, Feb. 2014.

[50] R. Sears, M. Callaghan, and E. Brewer. Rose: Compressed, log-structured replication. In *Proc. 34th VLDB Conf.*, pages 526–537, Aug. 2008.

[51] R. Sears and R. Ramakrishnan. bLSM: A general purpose log structured merge tree. In *Proc. SIGMOD Conf.*, pages 217–228, May 2012.

[52] D. J. Sheehy and D. Smith. Bitcask. a log-structured hash table for fast key value data. Technical report, Technical report, Basho Technologies, 04 2010, 2010.

[53] V. Sikka, F. Färber, W. Lehner, S. K. Cha, T. Peh, and C. Bornhövd. Efficient transaction processing in SAP HANA database: the end of a column store myth. In *Proc. SIGMOD Conf.*, pages 731–742, 2012.

[54] K. Srinivasan, T. Bisson, G. Goodson, and K. Voruganti. iDedup: Latency-aware, inline data deduplication for primary storage. In *Proc. 10th FAST Conf.*, Feb. 2012.

[55] R. Stoica, M. Athanassoulis, R. Johnson, and A. Ailamaki. Evaluating and repairing write performance on flash devices. In *DaMoN*, pages 9–14, June 2009.

[56] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik. C-store: a column-oriented DBMS. In *Proc. 31st VLDB Conf.*, pages 553–564, 2005.

[57] Y. Wang, M. Kapritsos, Z. Ren, P. Mahajan, J. Kirubanandam, L. Alvisi, and M. Dahlin. Robustness in the Salus scalable block store. In *Proc. 10th NSDI Symp.*, pages 357–370, 2013.

[58] H. Yadava. *The Berkeley DB Book*. Apress, 2014.