Everest: Scaling down peak loads through I/O off-loading

Dushyanth Narayanan Austin Donnelly Eno Thereska Sameh Elnikety
Antony Rowstron
Microsoft Research Cambridge, United Kingdom
{*dnarayan,austind,etheres,samehe,antr*}*@microsoft.com*

Abstract

Bursts in data center workloads are a real problem for storage subsystems. Data volumes can experience peak I/O request rates that are over an order of magnitude higher than average load. This requires significant overprovisioning, and often still results in significant I/O request latency during peaks.

In order to address this problem we propose Everest, which allows data written to an overloaded volume to be temporarily off-loaded into a short-term virtual store. Everest creates the short-term store by opportunistically pooling underutilized storage resources either on a server or across servers within the data center. Writes are temporarily off-loaded from overloaded volumes to lightly loaded volumes, thereby reducing the I/O load on the former. Everest is transparent to and usable by unmodified applications, and does not change the persistence or consistency of the storage system. We evaluate Everest using traces from a production Exchange mail server as well as other benchmarks: our results show a 1.4–70 times reduction in mean response times during peaks.

1 Introduction

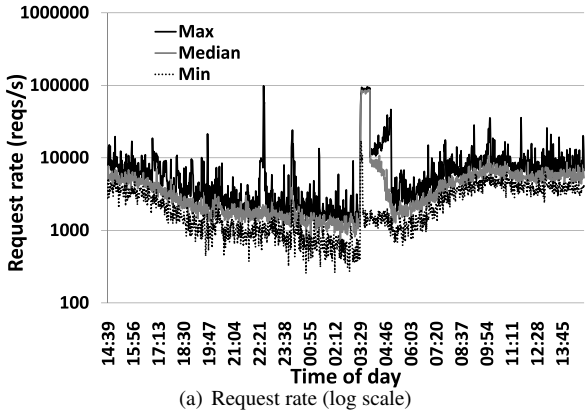
Many server I/O workloads are bursty, characterized as having peak I/O loads significantly higher than the average load. If the storage subsystem is not provisioned for its peak load, its performance during peaks degrades significantly, resulting in I/O operations having significant latency. We observe that workloads are usually unbalanced across servers in a data center, and often even across the data volumes associated with a single server. We propose Everest, a system that improves the performance of overloaded volumes by transparently exploiting statistical multiplexing across the storage bandwidth resources in the data center.

Everest monitors the performance of a data volume, and if the load on the volume increases beyond a predefined threshold, it utilizes spare bandwidth on other storage volumes to absorb writes performed to the overloaded volume. It does this by maintaining a *virtual short-term persistent store*, into which data is temporarily written, or off-loaded. The store is virtual in the sense that storage resources are not explicitly allocated to it; rather it is created by pooling idle bandwidth and spare capacity on existing data volumes either on a single server or across a set of servers in the same data center. In the common case, this can remove the majority of writes from the peak load, allowing the data volume under stress to serve mostly reads. When the peak subsides, the off-loaded data is lazily reclaimed back to the original volume, freeing the space in the Everest store. Everest handles short-term peaks and is not designed to handle long-term changes in load: these must be addressed by reprovisioning the storage subsystem and changing the data layout to match the new workload patterns.

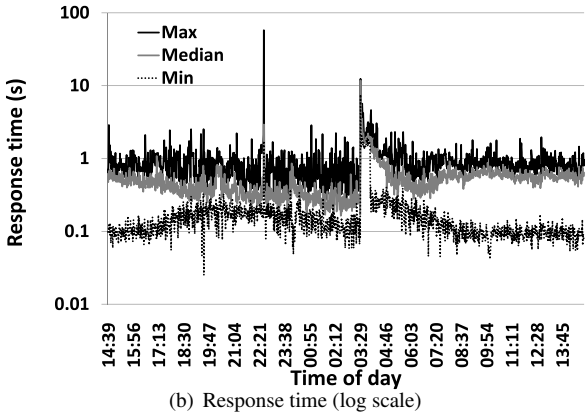
Everest thus provides a short-term, low-latency persistent store without the requirements for explicitly provisioned resources. Everest is interposed at the block I/O interface level, and is transparent to the applications and services running above it. It does not alter the persistence or consistency semantics of the storage subsystem, and unmodified applications can use Everest.

Two recent developments make storage pooling for peak I/O loads increasingly important. First, gigabit networking is ubiquitous in today's data centers, with servers configured with multiple high bandwidth NICs, and soon 10-gigabit networks will be common. This increase in network bandwidth relative to storage bandwidth is exploited by many storage technologies, such as Network Attached Storage (NAS) and Storage Area Networks (SAN). Everest also exploits it by allowing I/O off-loading across the network.

Second, as disk bandwidth increases more slowly than capacity, bandwidth rather than capacity increasingly determines the number of disks provisioned for an application. Peak off-loading avoids the need to provision each data volume individually for its peak load, which is often



(a) Request rate (log scale)



(b) Response time (log scale)

Figure 1: Exchange request rate and response time over a 24-hour period, 12–13 December 2007.

impossible (since peaks are unpredictable) and always expensive. Instead the volumes can be configured for the expected mean or 95th percentile I/O rate, and peak off-loading used to improve performance during periods when the I/O rate is higher than expected.

We evaluate the increase in performance Everest achieves using real world traces gathered from a production Exchange mail server, as well as database benchmarks and micro-benchmarks. The results show that Everest provides significant benefit. For example, for the mail server the mean response time during the worst peak is reduced by a factor of 70. In tests of database OLTP throughput, off-loading from a loaded machine to one idle machine increased throughput by a factor of 3, scaling up to a factor of 6 with 3 idle machines.

The rest of the paper is organized as follows. Section 2 provides further background, then Section 3 describes the design and implementation of Everest. Section 4 presents evaluation results, Section 5 related work, and Section 6 concludes the paper.

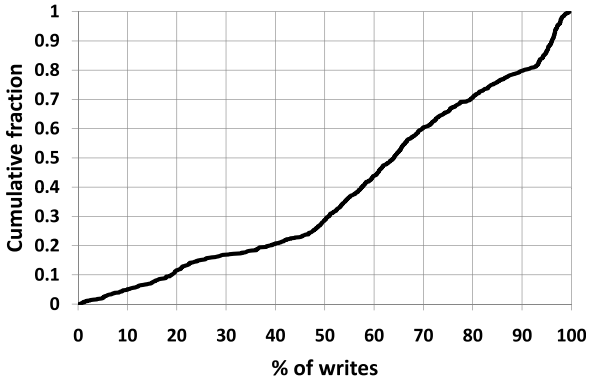


Figure 2: % of writes in Exchange: CDF over 1 minute intervals having mean response times above 1 second.

2 Background

In order to understand the impact of peak I/O loads, we examined a trace of a production Exchange e-mail server running inside Microsoft [26]. Employee e-mail at Microsoft is supported by a number of such servers, which are provisioned and maintained by the corporate IT department. The trace records the I/Os issued by one such server, which supports 5000 users. The trace covers 8 data volumes with a total capacity of 7.2 TB, for a 24-hour period starting at 14:39 on the 12th December 2007.

For each minute in the trace, we measured the mean request rate for each of the volumes, and Figure 1(a) shows for each minute the maximum and minimum request rates across all volumes, as well as the request rate for the median volume. It should be noted that due to the large variations in rates a log scale is used. The load is extremely bursty, and also unevenly distributed across volumes during peak bursts. Across the entire trace the peak-to-mean ratio in the I/O load is 13.5, and during the highest peak 90% of the load is on a single volume.

To understand the impact this load has on response times, we calculated the mean response time over 1 minute intervals for each data volume. Figure 1(b) shows the maximum, median, and minimum response times across volumes, again using a log scale. As expected, the response times vary widely, from under a second in the common case to above 10 seconds during the peaks. Further, there is substantial variation across volumes when at peak. This implies that, even on a single server, there is scope for statistical multiplexing of disk bandwidth during peak load episodes.

For Everest to be able to provide benefit, the I/O peaks must contain writes as well as reads. We believe peaks in server workloads are likely to have a significant fraction of writes. Storage subsystems are typically well-equipped to handle large streaming read workloads, and small random-access reads will benefit from caching at

various levels of the system. Figure 2 shows the fraction of write requests for every 1 minute interval in the Exchange traces in which the mean response time exceeded 1 second, as a cumulative distribution. Fewer than 5% of these intervals have less than 10% writes.

Battery-backed non-volatile RAM (NVRAM) is sometimes used in storage systems to improve I/O performance, although its use is often limited due to the cost and the need for maintenance of the batteries. NVRAM can only provide benefit when the I/O peak's footprint is smaller than the NVRAM size. The Exchange mail server that we traced is configured with 512 MB of NVRAM shared across all volumes. Figure 1 clearly shows that the response times are high despite the use of NVRAM. In general, it is not cost-effective to provision sufficient NVRAM to handle worst-case peak loads.

3 Design

Off-loading in Everest is configured on a per-volume basis. Here by volume we mean any block storage device: e.g., a single disk, array of disks, or solid-state storage device (SSD). An Everest *client* can be associated with any volume, which we then call the *base volume* for that client. The client is interposed on all read and write requests to the base volume. When the base volume is overloaded, the client off-loads writes into a virtual store. This allows more of the base volume's bandwidth to be used for servicing reads. When the load peak subsides, the client *reclaims* the data in the background from the virtual store to the base volume.

The virtual store is formed by pooling many individual physical stores, referred to simply as *stores* in this paper. Each store has an underlying base volume and uses a small partition or file on this volume. Thus stores are not explicitly provisioned with resources but export spare capacity and idle bandwidth on existing volumes. When a store's base volume is idle, it is used opportunistically by clients whose base volume is heavily loaded.

Each Everest client is configured to use some set of stores, called its *store set*. The stores can be on different volumes on the same server as the client, or on different servers in the data center. A single volume can host a client, a store, or both. In general a client can use any store, and a store can be used by multiple clients. However, a client would not be configured to use a store having the same base volume as itself, since this does not contribute any additional disk bandwidth.

Figure 3 shows an example of a server in a data center using Everest. The server has two volumes, each of which is an array of many disks. Both volumes are configured with Everest clients, which interpose between the volumes and the file system and appear as standard block devices to the file system. Client 1 is configured to use

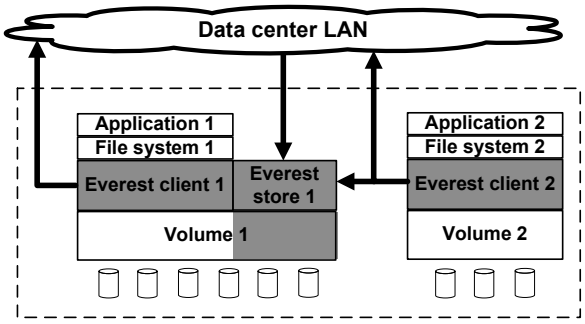


Figure 3: Example server running Everest.

Everest stores on other servers in the data center; client 2 is configured to use the store on volume 1 as well as stores on other servers.

The key challenges for Everest are to retain the consistency and persistence guarantees of the base volume while providing low latency and high throughput for off-loaded writes. This is challenging because Everest needs to maintain a consistent view of the data logically stored on each data volume, even though it may be physically distributed across several other volumes and servers when off-loaded. While data is being off-loaded there can be no synchronous writes of meta-data to the original volume, as this would increase its I/O load when already overloaded. Further, different versions of the same data could be off-loaded to multiple different locations during the same I/O peak, and a read request might span data that is off-loaded to different locations. The system must always return the latest version of the data in all cases. The persistent state of the system must also be recoverable after a failure.

Everest meets these goals through the combination of techniques described in the rest of this section. Much of the complexity in the Everest design is in the client, which is responsible for deciding when to off-load, where to off-load, and when to reclaim off-loaded data. The client is also responsible for consistency and fault-tolerance. We first describe the simpler Everest store and then the more complex client.

3.1 Everest store

An Everest store provides short-term write-optimized storage. It exports four operations: $write$, $read$, $read\_any$, and $delete$. Clients call $write$ when off-loading write requests. $read$ is invoked on a store when a client receives a read request for data off-loaded to that store. This happens rarely since blocks are only off-loaded for short periods, and are likely to remain in application and file system buffer caches during these periods. $read\_any$ and $delete$ are background operations used by clients to reclaim off-loaded data from the store.

The Everest store is optimized for low latency and high throughput of the frequently called foreground operation: $write$. The store uses a circular log layout to achieve sequential performance on writes. Each $write$ request results in a single write to the log head of a record containing both data and meta-data. The meta-data is also cached in memory for fast lookup. Background delete requests also cause records to be written to the log head; deletion records contain only meta-data and no data. If there are multiple concurrent $write$ requests, the store issues them concurrently to the disk to maximize performance. Write acknowledgements are serialized in log sequence order: a write is acknowledged only after all writes preceding it in the log are persistent on disk. This ensures that there are no holes in the log, and that all acknowledged writes are recoverable after a crash.

In the common case, the store absorbs a write burst, causing the head of the log to move forward. Subsequently clients reclaim and delete the data; this moves the tail forward and shrinks the log which eventually becomes empty. Thus both the head and the tail of the log move forward, wrapping around to the beginning when they reach the end of the file or partition. The head of the log is never allowed to wrap past the tail. In the common case the log does not fill up the allocated storage capacity. If the store does reach its capacity limit, it stops accepting write requests but continues to accept read and delete requests, which will eventually shrink the log.

The Everest store writes meta-data in each log record which allows it to correctly recover its state after a crash. The meta-data contains the ID of the client that issued the write, the block range written, the version, and a data checksum. State is recovered after a crash by scanning the log from tail to head. A pointer to the log tail is stored at the beginning of the store's file or partition. A write or delete record written to the log head might cause older records in the log to become stale; when the tail record becomes stale, the tail pointer can be moved forward. Tail pointer updates are done lazily during idle periods to avoid contention with other requests on the volume.

The head of the log is the last valid record read during the recovery scan. Partially written log records are detected by verifying the checksum in the record header. Over time, the log head might cycle repeatedly around the storage area. Hence, in general, the disk blocks following the log head could contain arbitrary data. To distinguish arbitrary data from valid log records, the store uses a 128-bit *epoch ID* that is randomly generated every time the log head wraps around. Each record that is appended to the log contains both its own epoch ID as well as that of the previous record. This property is verified during recovery, ensuring that only valid records are used to reconstruct the state after a failure.

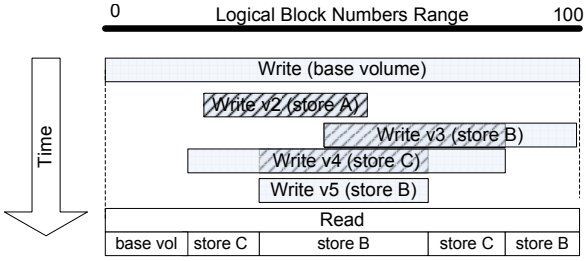


Figure 4: Example of range map holding versions: hashed areas represent stale/overwritten subranges.

3.2 Everest client

The Everest client is responsible for

1. off-loading writes to stores in its store set,
2. reclaiming data from the stores, and
3. guaranteeing persistence and consistency.

Achieving these goals is challenging for two reasons. First, when off-loading writes, the client must maintain consistency without writing either data or meta-data to the already overloaded base volume. Second, off-loaded data could be spread over multiple Everest stores; the client must correctly redirect read requests to the appropriate combination of base volume and/or Everest stores.

Since Everest interposes transparently above a block device, it provides the same consistency and persistence semantics as a local block device. The data returned for any read request is always the result of the last acknowledged write to that block, or of some issued but unacknowledged write. This property holds across transient failures, i.e., after a crash or reboot. Everest does *not* implement sharing of data across clients: this must be done in a higher layer if desired. At the Everest level a client mediates all access to its base volume and hence owns the namespace of blocks on the volume.

This section describes the key features of the Everest client: recoverable soft state, load balancing, reclaiming, and N-way off-loading for fault-tolerance.

3.2.1 Recoverable soft state

To avoid writing meta-data to the base volume when it is loaded, the client keeps almost all its meta-data as in-memory soft state. The only persistent state kept on the base volume is the store set: the list of Everest stores that the client can off-load to. This set changes infrequently, and is not changed during load peaks.

The soft state for each client contains an entry for each range of blocks off-loaded to an Everest store, which specifies the range, the store holding the latest version, and the version. The soft state is accessed on each I/O intercepted by the client, and is designed to be compact as well as efficient to query and update. It is maintained as

a *range map*, which contains ordered, non-overlapping ranges of byte offsets in a logarithmic search tree [20]. This supports fast lookup and update while using memory linear in the number of distinct block ranges off-loaded, rather than in the number of blocks. The range map can be queried to find the ranges that are currently off-loaded, and the Everest store holding the latest version. Range queries and updates can overlap arbitrarily, for example an update could partially overwrite an existing range with a newer version: the range map handles this by splitting ranges appropriately. Figure 4 shows an example of the in-memory state of a client after multiple overlapping writes have been completed.

On each read request, the client queries the range map to find out how to split the read between stores holding off-loaded blocks (if any), and the base volume. On each write request, the client queries the range map to see if the write request overlaps any currently off-loaded blocks. When an off-loaded write is completed, the client updates the range map before acknowledging the write completion to the higher layers.

An Everest client can correctly recover its soft state after a crash. Each off-loaded write sent to an Everest store is written along with meta-data that identifies the client, the base volume, the block range written, and the version. Keeping meta-data on the Everest stores allows the client to quickly and efficiently recover the soft state. After a failure the client retrieves meta-data in parallel from all the stores in its store set. This meta-data is cached by each store in a per-client range map. The responses from the stores are then merged to construct the client's state. If a store crashes and recovers concurrently with the client, the client waits for the store to recover by scanning its log, and then retrieves its meta-data.

3.2.2 Load balancing

In general, when an Everest client receives a write request, it can choose to send the write to the base volume or to any store in its store set. In making this choice, the client has four goals. First, it must always maintain correctness. Second, it should only off-load when the base volume is overloaded: unnecessary off-loading wastes resources and increases the chances of contention between workloads. Third, it should only off-load to stores on lightly loaded volumes, to avoid degrading the performance of workloads using those volumes. Finally, subject to these constraints, it should balance load between the available stores and the base volume.

Correctness requires that reads always go to the location holding the latest version of the data: the client splits read requests as necessary to achieve this. Most reads will go to the base volume, since reads of recently off-loaded data are rare. Most writes can be sent either to the base volume or to any available store in the store set. However a write that overwrites currently off-loaded data must also be off-loaded to some store, not written to the base volume. This is because the stores support versioning, but the base volume is a standard block device and hence cannot be assumed to support versioning. Hence if any version of the data is currently off-loaded, then the latest version must also be off-loaded to ensure that the client's state is correctly recoverable.

While the client generally has no choice on where to send read requests, it can redirect most write requests according to load. If a write overlaps a currently off-loaded range, then it is sent to the least loaded available store. Otherwise, if the load on the base volume is above a threshold T_{base}, and the least-loaded store has a load lower than another threshold T_{store}, writes are sent to the least loaded of the base volume and the available stores. Otherwise, writes are sent to the base volume.

Each Everest store periodically broadcasts load updates on the network and updates are also piggybacked on response packets sent to clients. The updates contain several block device level load metrics such as request rates, response times, and queue lengths. Thus a variety of load balancing metrics and policies are possible. Currently we use the simple policy described above, and we use the queue length — the number of I/O requests in flight — as the load metric.

3.2.3 Reclaiming

When the base volume load is below the threshold T_{base}, the Everest client *reclaims* off-loaded data to the base volume in the background. The client issues low-priority $read\_any$ requests to all stores holding valid data for it. Each lightly-loaded store returns valid data and meta-data (if any) for some range of blocks off-loaded by the client to that store. In general a store's log might contain many records corresponding to a given client; the store chooses the record that is closest to the log tail and contains some valid (non-stale) data for the client. The client writes the data to the base volume, and then sends a deletion request to the store. The client sends such deletion requests to a store whenever data on it becomes stale, either due to a reclaim or due to a newer version being written to a different store.

In some rare cases the reclaim process results in wasted I/O; however consistency is always maintained and all off-loaded data is eventually reclaimed. For example, while a block is being reclaimed, an application might issue a new write for the same block: any work done to reclaim the old version will be wasted. Correctness is maintained through three invariants. First, the client will send a deletion request for a version v only if the corresponding data has been written to the base

volume, or a version $v' > v$ has been written to some store. Second, if a version v of a block is currently off-loaded, then the client will off-load any new write to that block, with a higher version $v' > v$. Third, the client will not send a deletion request for the highest currently off-loaded version of any block until all older versions of the block have been deleted from all stores.

The Everest client performs multiple concurrent reclaim operations for efficient pipelining of disk and network bandwidth. Depending on the support for low-priority I/O in the underlying system, reclaim I/O could have some impact on foreground I/O performance. In Everest this tradeoff is controlled by setting the concurrency level of the reclaim process, which limits the number of outstanding reclaim I/Os per Everest client. This is currently a fixed value that is configured per-client: it could also be dynamically regulated using a control process such as MS Manners [8].

3.2.4 N-way off-loading for fault tolerance

Off-loading introduces a failure dependency from a client to the store to which it has off-loaded data. If the store fails, the off-loaded data will become unavailable. To maintain the same availability and persistence as the non off-loading case, Everest masks this failure dependency by adding support for fault-tolerance. Broadly, there are two classes of failure: transient (crash) failures causing temporary data unavailability, and permanent disk failures causing data loss. We first discuss transient failures and then permanent disk failures; this paper does not consider Byzantine failures.

Everest provides fault-tolerance through N-way off-loading. Each off-loaded write is sent to N stores on N different servers, where N is a per-client configurable parameter. In general the client's store set will be larger than N, allowing the client to choose the N least-loaded stores and minimizing the impact on other workloads. Since writes are versioned by Everest, each write can be sent to any N stores independently of previous writes. Note that the client still maintains a consistent view of all the data, and all access to the data is mediated through the client. Hence there is no need for any consensus or co-ordination protocol between the stores.

N-way off-loading can mask up to $N-1$ store failures. If an Everest store fails, the client continues to service requests using the remaining stores. However data held on the failed store now has only $N-1$ copies and is more vulnerable to subsequent failures. The client reclaims such vulnerable data before any other data. The amount of vulnerable data to be reclaimed is bounded by limiting the amount of data off-loaded to any single store.

When an off-loaded version on a store becomes stale, the client normally sends a deletion request to that store. However if the store has failed, the client must ensure that any stale versions on it are eventually deleted. In this case the deletion request is queued locally and persistent, versioned deletion records are written to N other stores in the store set. These ensure that if the client crashes it can correctly reconstruct the outstanding deletions. When the failed store becomes available, the client sends it the queued deletion requests. When these are acknowledged the client garbage-collects the deletion records.

If a volume hosting a store is permanently decommissioned, Everest clients using the store must delete it from their store set. The clients then garbage-collect all deletion records pertaining to that store. Deletion of a store from the clients' store sets can be done manually by the administrator when decommissioning a volume, or automatically by clients when a store has been unresponsive for a certain amount of time.

Permanent disk failures: N-way off-loading also serves to protect against permanent disk failures on stores by adding redundant copies. An alternative approach is to ensure that Everest stores themselves are resilient to disk failures, for example, if the underlying storage uses RAID. In this approach, the system administrator configures each client to use only stores whose base volumes have at least as much redundancy as the client's base volume.

Reading and reclaiming: The Everest client load-balances reads of N-way off-loaded data by sending each read to the least loaded store holding the data. However, read requests for off-loaded data are rare. While reclaiming, low-priority $read\_any$ operations are issued concurrently to multiple stores: the first store to respond sends a cancellation request to the other stores.

3.3 Implementation

The current Everest prototype is implemented at user level. The store is implemented as an RPC server exporting the four store operations. The client is implemented as two library layers. The virtual store layer exports the Everest store operations for the virtual store, implemented by load-balancing across the Everest stores in the store set. The policy layer exports a standard block read/write interface and issues requests to the base volume and to the virtual store layer as appropriate.

The user-level library is sufficient for testing Everest with block-level traces and micro-benchmarks. We are also able to use the user-level prototype with unmodified binary Windows applications such as SQL Server. To do this, we intercept the application's file I/O calls using DLL redirection [13]; a policy layer then maps these calls to virtual store operations.

4 Evaluation

The main aim of the Everest design is to improve I/O performance under peak load. The first part of this section quantifies this improvement by measuring the impact of peak off-loading on I/O response times, using block-level traces of a production Exchange mail server.

While trace replay from a production server gives us a realistic evaluation of I/O response times, application benchmarks let us measure the improvement in end-to-end application throughput. The second part of the evaluation is complementary to the first: it uses an OLTP benchmark with an unmodified SQL Server application to measure the performance of off-loading in various configurations, and identifies the individual performance benefits of off-loading and of using a log-structured store. Finally it shows how application performance scales as more idle spindles are added to the network as well as when more load is added.

Finally, this section uses micro-benchmarks to evaluate the sensitivity of the base Everest performance to the read-write ratio of the workload, since Everest is designed to off-load write load but not read load. It also uses micro-benchmarks to test the scaling of the I/O performance as idle spindles are added, as well as the limits on efficiency of the reclaim process.

All the results in this evaluation are based on experiments run on a hardware testbed running Everest. The testbed consists of four HP servers, each with a dual-core Intel Xeon 3 GHz processor and an HP SmartArray 6400 series RAID controller connected to a rack-mounted disk enclosure with a SCSI backplane. Each enclosure contained 14 high-end enterprise disks: 15K RPM Seagate Cheetahs. The servers were connected to each other by a switched 1 Gbps Ethernet. Communication between Everest clients and stores uses in-process shared memory within a single machine; TCP for unicast communication across machines, and UDP subnet broadcast for periodic broadcasts of store load metrics. For N-way off-loading Everest clients use a combination of TCP and UDP to implement reliable multicast.

4.1 I/O response times: Exchange

In Section 2 we described the I/O traces from a production Exchange mail server, which motivated peak off-loading. Here we evaluate Everest against those traces by replaying them on our testbed.

We selected three episodes from the 24-hour trace shown in Figure 1 that covered the three highest peaks in request rates (measured on a 1 min time scale). The storage capacity of our testbed machines is much lower than that of the original server, which was 7.2 TB across 8 data volumes. Hence for each episode we selected three volumes: the volumes having the maximum, median, and the minimum number of requests during the episode. This corresponds to a configuration where off-loading is restricted to the three chosen volumes, i.e. the three volumes do not off-load to, or accept off-loads from, the other 5 volumes. The three selected volumes were then mapped onto volumes on one of our testbed servers. To achieve the required capacity, each testbed volume was configured as a RAID-0 array with four 146 GB Seagate Cheetah disks; the original production server uses redundant configurations such as RAID-5.

	Time	Requests	Read %	Peak rate
1	22:28–22:58	444345	27%	98206 reqs/s
2	03:16–04:01	5072789	91%	95179 reqs/s
3	10:05–10:35	620519	24%	35568 reqs/s

Table 1: Peak episode traces.

	Mean		99th percentile	
	Original	Testbed	Original	Testbed
1	1.17 s	1.53 s	8.8 s	11.2 s
2	2.08 s	0.06 s	20.0 s	0.6 s
3	0.43 s	0.04 s	3.1 s	0.4 s

Table 2: Peak episode response times.

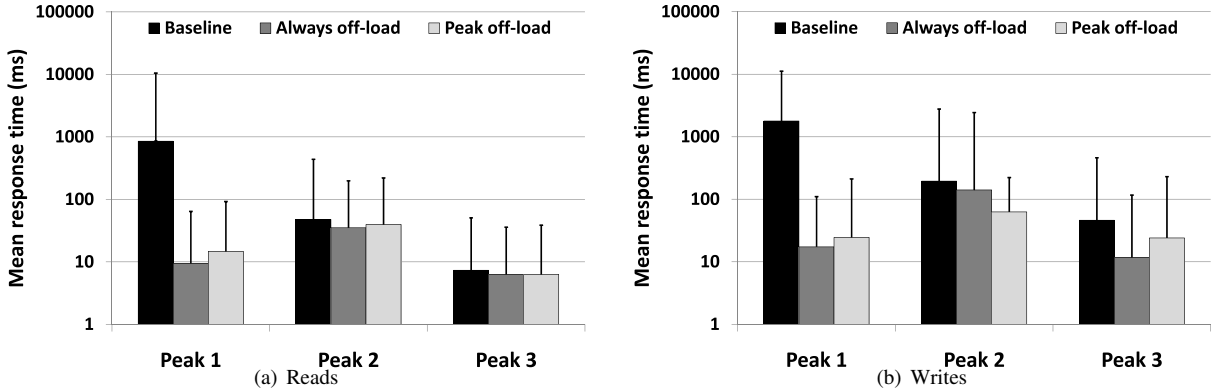
Table 1 shows the three peak episodes covered. The original trace files are divided into 15 min segments; each episode consists of the segment containing the request rate peak, as well as the following segment, to ensure that the trace would be long enough to cover any reclaim activity. Peak episode 2 was extended by an additional 15 min in order to cover all the reclaim activity in all the experiments. Table 2 shows the response times from the original trace as well as those measured on our testbed. The testbed response times are similar to the original hardware for peak 1, and lower (but still high) for the other two peaks.

We evaluated three configurations:

- *Baseline*: no off-load enabled.
- *Always off-load*: all writes are off-loaded, and data is never reclaimed.
- *Peak off-load*: off-load and reclaim are enabled with the queue length based policy described in Section 3.2.2, with $T_{base} = T_{store} = 32$.

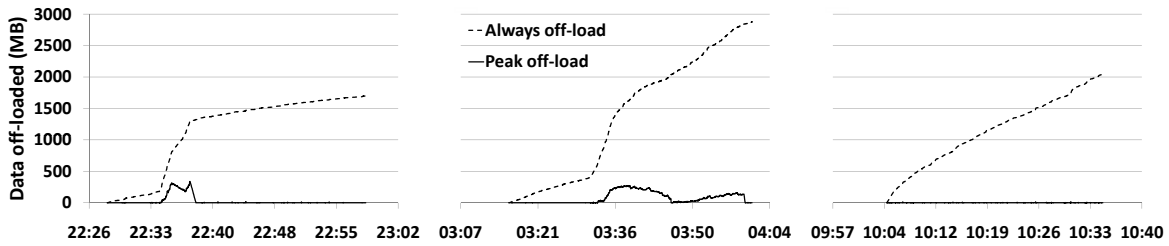
In the two off-load configurations, each volume was configured with an Everest client. Each volume also hosted an Everest store on a small partition (less than 3% of the volume size) at the end of the volume. The off-load configurations used 1-way, single-machine off-loading: for off-loading between RAID volumes on a single server we expect this to be the typical configuration.

Response time: Figure 5 shows the mean and 99th percentile read and write response times achieved by the



(a) Reads (b) Writes

Figure 5: Effect of off-load on mean response time (log scale). Error bars show 99th percentile response times.



(a) Peak 1 (b) Peak 2 (c) Peak 3

Figure 6: Amount of off-loaded data over time.

three configurations for all three peaks. Note that the y-axes are on a log scale. Off-loading improved both read and write response times by almost two orders of magnitude for the worst peak (peak 1), and significantly for the other two peaks. Writes benefit by off-loading from the heavily loaded volume to the lightly loaded ones, and reads benefit by having fewer writes to contend with.

For the write-dominated peaks 1 and 3, "always off-load" has better performance than "peak off-load". This is because it balances load more aggressively than the "peak off-load" policy, which only off-loads when the base volume is heavily loaded and some Everest store is lightly loaded. The disadvantage of "always off-load" is that if the Everest store's underlying disks are busy, then the resulting contention will hurt performance for both contending workloads. This effect can be seen clearly for peak 2, where the write performance of "always off-load" is worse than "peak off-load".

Reclaim time: It is important to know how much data is off-loaded and for how long, since this affects space usage on Everest stores as well as vulnerability to failures. Figure 6 shows for each peak episode the amount of off-loaded data over time, summed across all three clients, for both the "peak off-load" and the "always off-load" policy. With "always off-load", data is off-loaded more aggressively and never reclaimed. Hence, the amount of off-loaded data increases at a much higher rate and never decreases. Interestingly this resulted in more off-loaded data for peak 2 than for the other two; although peak 2 only has 9% writes, in absolute terms it has more writes than the other two peaks. By contrast, the "peak off-load" policy off-loads less data and reclaims it using idle bandwidth: this keeps the amount of off-loaded data low. For peak 3, reclaiming using idle bandwidth is so effective that the amount of off-loaded data is always close to zero. The difference between the two policies shows the importance of off-loading selectively, and of reclaiming off-loaded data.

Sensitivity to parameters: Figure 7 shows the sensitivity of off-load performance to three parameters: T_{base}, the base volume load below which the client will not off-load; T_{store}, the maximum load on a store beyond which clients will not off-load to it; and $N_{reclaim}$, the maximum number of concurrent reclaim I/Os per client. In each case one parameter was varied while the others were set to their default values: $T_{base} = 32$, $T_{store} = 32$, and $N_{reclaim} = 256$. The graphs show the mean response time and the mean amount of off-loaded data, across all clients and all peaks. Performance is generally insensitive to T_{base} and T_{store}: when T_{store} is high, off-loaded I/Os begin to contend with non off-loaded I/Os, but this effect is small compared to the order-of-magnitude re-

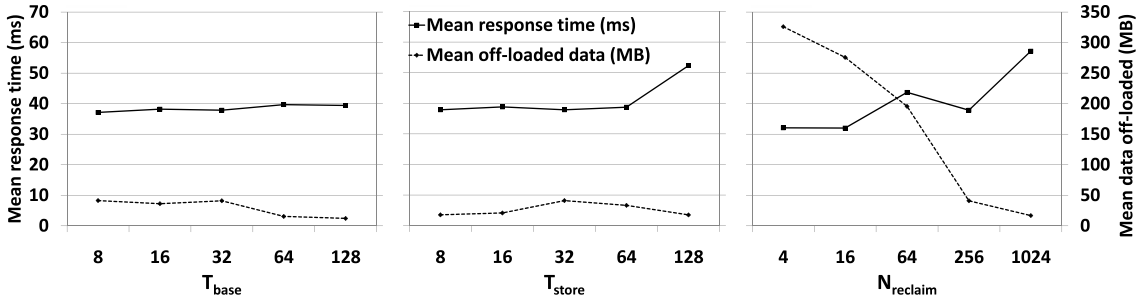
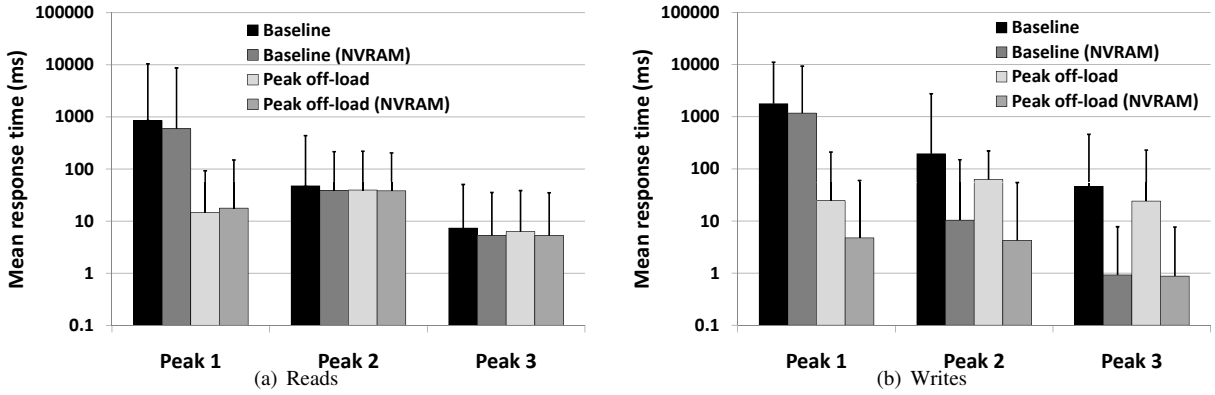


Figure 7: Sensitivity to off-load and reclaim parameters.



(a) Reads (b) Writes

Figure 8: Effect of off-load with NVRAM.

ductions achieved by off-loading. Performance degrades slightly with higher $N_{reclaim}$, due to contention between reclaim I/Os and foreground I/Os. However, for this particular workload, increasing $N_{reclaim}$ from 64 to 256 reduced the fraction of reads that was off-loaded from 0.4% to 0.1%, resulting in a slight decrease in overall response time. The main effect of $N_{reclaim}$ is on the amount of data left off-loaded: decreasing $N_{reclaim}$ below 256 underutilizes the available idle disk bandwidth and it takes substantially longer to reclaim data.

NVRAM: Enterprise RAID controllers are often augmented with battery-backed non-volatile RAM (NVRAM), which can potentially improve write performance by persisting writes temporarily in the NVRAM. The original traced server had 512 MB of NVRAM shared across all its volumes and yet had high response times under peak load. To confirm this, we enabled the NVRAM on our test server. This was 128 MB shared across 3 volumes, set to the default configuration of 50% for read acceleration and 50% for writes. Figure 8 compares the baseline and peak off-load performance with and without NVRAM. Even with NVRAM, the largest peak (peak 1) still shows very high read and write response times, because the amount of data written exceeded the capacity of the NVRAM. Peak off-loading by contrast substantially reduces response times for peak 1.

Max log size	982MB
Max log records	218505
Max valid data	193MB
Max recovery time	14.5 s
Total meta-data (compressed)	247 KB

Table 3: Worst-case recovery statistics.

Overheads: The main CPU overhead of Everest is the lookup and update of in-memory meta-data, which are logarithmic in the size of the meta-data. In our experiments these added an average CPU overhead of 56 μs per off-loaded write, which translated to an average CPU consumption over all 3 peaks of 0.4%. The meta-data also adds a memory overhead; the high watermark of the meta-data memory usage was 11.7 MB. With a more optimized implementation these overheads could be further reduced. Since our testbed was limited to four machines we did not measure the scalability of using subnet broadcast for load updates. However we note that the broadcasts are limited to a subnet, and ethernet multicast could be used to further limit their scope. The broadcast frequency is also low: at most once every 100 ms.

Recovery times: Off-loading occurs during relatively infrequent peak load episodes; hence usually the amount of off-loaded state to be recovered after failure will be

very small. We measured recovery times for Everest clients and stores in a worst-case scenario where the Exchange server fails when the amount of off-loaded data is highest. For our traces, this was 10 min into peak episode 1. Table 3 summarizes the results. Since Everest stores recover concurrently, we show statistics for the store which was the slowest to recover. The recovery time is essentially the time to scan the entire log on disk sequentially. In the single-machine scenario the Everest client can recover as soon as all stores in its store set have recovered. In the network off-load case, the stores must also transfer meta-data over the network to the client; however, as shown in Table 3, the meta-data is small.

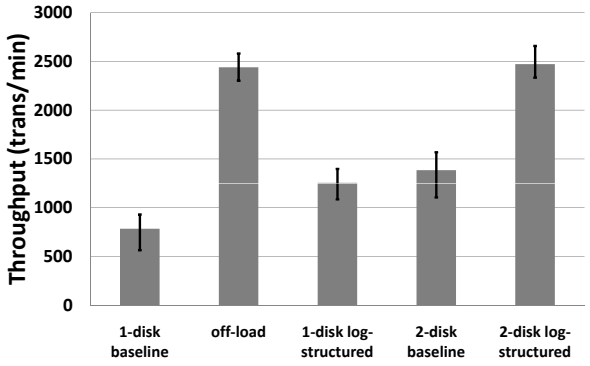


Figure 9: Single-server OLTP throughput.

4.2 Application throughput: SQL Server

The previous section showed substantial improvements in I/O response time from peak off-loading, based on trace replay. In this section we use a standard benchmark to measure the effect of off-loading on the end-to-end throughput of the application. Our test application is Microsoft SQL Server running an OLTP (TPC-C like) benchmark. The figure of merit is the transaction throughput measured as the number of "new order" transactions successfully completed per minute. We measure the saturation throughput, i.e. we increase the concurrency level until the server can support no more connections, and then measure the achieved throughput.

In the OLTP workload 43.5% of the transactions are updates ("new order" transactions), which results in about 27–32% writes at the disk I/O level. The disk workload has poor locality: when the database is larger than the available buffer cache memory, the performance is limited by the random-access I/O performance of the underlying storage. In our experiments we used a database size of 7.5 GB (corresponding to 100 warehouses) and a SQL Server buffer cache size of 256 MB.

The database server used was an unmodified SQL Server 2005 SP2, and we intercepted the server's file I/O using DLL redirection. Following standard practice, the database file and the transaction log file were stored on two separate volumes; each of these contained a single Seagate Cheetah 15K disk. Off-loading was enabled only for the database file, since it was the bottleneck device: we measured the utilization of the log disk and found that it was 7.6% on average and under 20% always. The policy used was the queue length based policy described in Section 3, with $T_{base} = T_{store} = 32$.

All results shown are the average of 5 runs; error bars show the minimum and the maximum values observed. Each experiment was run for 10 min to warm up the cache, and another 10 min to measure the throughput.

In our first experiment, we measured the benefit of off-load for OLTP on a single server. We compared the baseline throughput with no off-loading enabled, to that obtained when off-loading was enabled with a single Everest store on a different disk on the same server. The first two bars in Figure 9 show that off-loading provides a 3x increase in throughput over the baseline. We repeated the same experiment with the Everest store on a different testbed machine to the client; the results were quantitatively the same, indicating that the network was not a bottleneck for this configuration.

Two factors contribute to this benefit: the additional spindle provides additional I/O throughput, and random-access writes are converted to sequential-access writes on the Everest store's log. To separate out the contributions of these two factors, we evaluated two more configurations, shown in the third and fourth bars of Figure 9: a 1-disk log-structured configuration, and a 2-disk striped (RAID-0) array with the default (non-log-structured) layout. Each does roughly 2x better than the baseline case. This shows that for this workload, the benefit of log-structured writes is roughly equal to that of adding a second spindle.

The last bar in Figure 9 shows the performance of a configuration which is log-structured and uses a 2-disk striped array. We see that the performance of off-loading is comparable to that of this configuration. In other words, off-loading gets the full benefit both of log-structured writes and of an additional spindle, but by opportunistically using the second spindle for off-load rather than explicitly provisioning it.

Impact of unreclaimed data: By only off-loading to lightly loaded volumes, Everest ensures that off-loaded writes will not severely degrade the performance of the applications using those volumes. However reads of off-loaded data must be sent to the Everest store that holds the latest version of the data. In the common case, off-loaded data is reclaimed before being read, i.e. foreground reads on the Everest store are rare. Additionally, if N-way off-loading is enabled, the client is likely to find at least one lightly loaded replica to read.

However, in the worst case, there can be contention for

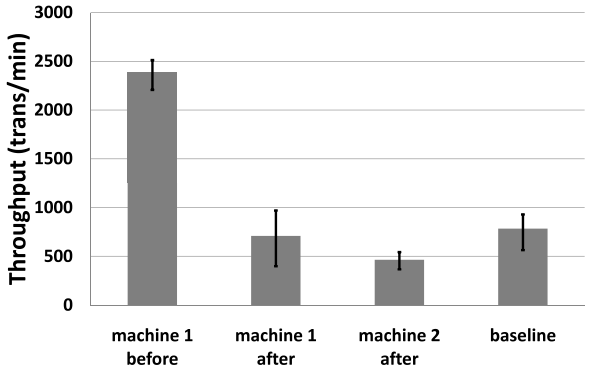


Figure 10: Off-loading worst-case scenario.

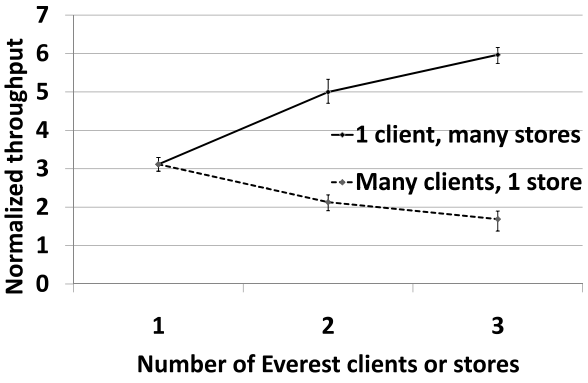


Figure 11: Scaling of OLTP off-load performance.

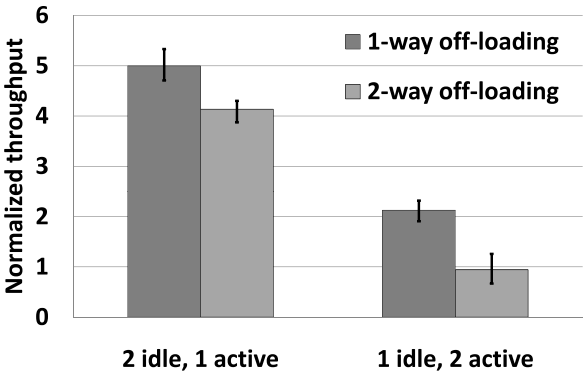


Figure 12: OLTP throughput with 1-way and 2-way off-loading.

disk bandwidth between an Everest store satisfying read requests and applications using the store's base volume. To quantify this effect, we measured a "worst-case" scenario using two machines both running SQL Server and OLTP. Machine 1 was configured for off-load, and machine 2 hosted an Everest store on the same disk as the OLTP database. For the first 10 min of the experiment, only machine 1 was active and it achieved the expected performance improvement. At the end of 10 min, machine 1 had off-loaded 936 MB (13% of the database) to the store on machine 2. At this point machine 2 also became active. Due to this competing load, machine 1 stopped off-loading fresh data onto machine 2; however, accesses to already off-loaded data still went to the Everest store on machine 2. Figure 10 shows the performance of the machines before and after machine 2 became active, compared to the baseline (no off-loading) case. Machine 1's performance dropped to 93% of baseline: relatively few of its I/Os were redirected to the contended disk. Machine 2's performance dropped to 59% of the baseline value: all its I/Os went to the contended disk.

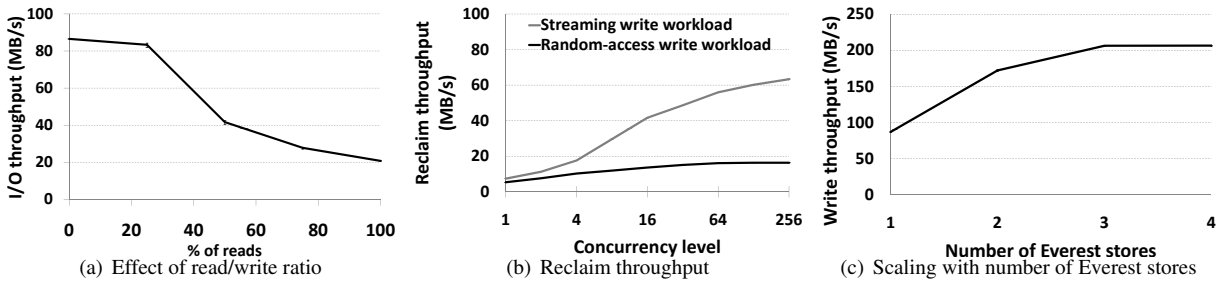
This experiment shows a worst-case scenario of continuous load with no idle periods. Everest is not designed for such scenarios, but rather to improve performance when load is bursty and unbalanced. To avoid off-loading during long periods of high load, the amount of data off-loaded by each client can be bounded. This limits the amount of contention in a worst-case scenario.

Performance scaling: In general, one or more Everest clients can share one or more Everest stores. A single client can improve performance by off-loading to more than one idle store; alternatively multiple clients can off-load to a single idle store.

To measure the effect of using multiple idle stores, we ran the OLTP benchmark on one of the SQL Servers with off-loading enabled, and added 1–3 machines hosting Everest stores but otherwise idle. The solid line in Figure 11 shows the throughput achieved by SQL Server. The throughput is normalized to the baseline throughput of SQL Server with no off-loading (the baseline case is not shown on the graph). With one idle store we get a 3x improvement over the baseline, as seen before. The throughput scales linearly up to 3 idle stores, at which point SQL Server becomes CPU-bound.

To measure the effect of multiple clients sharing a single idle store, we configured one machine to host an Everest store and added 1–3 machines running OLTP and configured for off-load. The dashed line in Figure 11 shows the normalized average throughput achieved by the active machines. As more load is added, per-server throughput decreases slightly, since the benefit of the additional spindle must be shared. However, even with 3 SQL Servers sharing a single Everest store, we get an average speedup of 1.7x.

N-way off-loading: Everest supports N-way off-loading to create redundant copies for fault-tolerance. In general this adds overheads due to the bandwidth used to write additional copies. We compared the performance of 1-way and 2-way off-loading on a configuration with one machine running OLTP and two idle machines hosting Everest stores. The throughput of these two cases, normalized to the 1-disk baseline throughput, is shown by the first two bars in Figure 12. We see that 2-way



(a) Effect of read/write ratio (b) Reclaim throughput (c) Scaling with number of Everest stores

Figure 13: Micro-benchmark results.

off-loading incurs a performance penalty since it uses twice as much write bandwidth on the stores; however the penalty is relatively small since the stores are write-optimized. The second pair of bars in Figure 12 represents a configuration with two active SQL Servers and one idle machine. In this case, with 1-way off-loading the two clients share the performance benefit of off-loading to one extra spindle. With 2-way replication we get performance equivalent to the baseline. This is because neither Everest client can find two lightly loaded Everest stores and hence no off-loading occurs.

We also see that 2-way off-load to two stores does better than 1-way off-load to one store; this is because although the write load per store is the same, the read load is halved in the former case. However, in general when using Everest for short-term peak off-loading, we would not expect a high read load on the Everest stores: the main purpose of N-way off-loading is to provide fault-tolerance rather than read performance.

4.3 Micro-benchmarks

In this section we use synthetic micro-benchmarks to measure the sensitivity of Everest performance to the workload read/write ratio and the reclaim concurrency level, as well as the scaling of performance with the number of clients and stores. The results presented here all use 64 KB reads and writes, and are based on the average of 5 runs of each experiment, with error bars showing the maximum and minimum values. Each experiment runs for 20 seconds. All configurations used a single Everest client with a single disk, and one or more Everest stores each with a single disk. The Everest client is configured with an "always off-load" policy.

Figure 13(a) shows the throughput achieved by an Everest client with a single store for a random-access workload, as a function of the workload's read/write ratio. With a write-only workload, Everest achieves a throughput equal to the streaming write bandwidth of the disk (observed to be 90 MB/s), since it converts random-access writes to sequential accesses. As the fraction of reads increases, the throughput is increasingly dominated by the reads: since the workload has no temporal locality, almost all reads go to the base disk (observed random-access throughput for this disk type is 21 MB/s). However, off-loading continues to show benefit until the workload is 100% read-dominated.

Figure 13(b) shows the reclaim throughput achieved by Everest with an otherwise unloaded base disk and Everest store, for random-access and streaming write workloads. Since off-loaded records are reclaimed in log order, the writes to the base disk during reclaim have the same locality pattern as the original write workload. For a random-access workload, reclaim throughput is dominated by the random-access write performance of the base disk, which increases with the concurrency level to a maximum of about 20 MB/s. For a streaming workload, the throughput is limited by the Everest store, which alternates between reading from the tail of the log and writing deletion records to the head. At higher concurrency levels, this effect is amortized by batching, and we can achieve a throughput of up to 70% of the streaming bandwidth of the disk.

We conclude the evaluation section by measuring the scaling of client write throughput as a function of the number of available stores. The results are shown in Figure 13(c). Scaling is linear up to 2 stores. With 3 stores the system becomes CPU-bound, achieving 206 MB/s, or 3300 IOPS. About 9% of the CPU time was spent in Everest meta-data operations; the remainder was due to our user-level trace replay infrastructure, which requires at least one user-kernel transition and two memory copies per I/O.

5 Related work

Most research on storage system performance has focused on achieving good performance when I/O loads remain within expected limits. In contrast, Everest mitigates performance degradations from short-term unpredictable bursts, and is orthogonal to these efforts. We are unaware of much work that exploits pooled storage as a short term store to improve performance.

Self-scaling systems and data migration: There has been much work on dynamic reconfiguration of storage to adapt to workloads, e.g. switching between RAID-5 and mirroring [24] or re-encoding of data in cluster storage [1]. Many storage systems [2, 5, 7, 10, 11, 14, 18] scale incrementally by redistributing existing data across newly added nodes; the same mechanisms can also be used to dynamically reconfigure individual workloads when load changes. For example, Amazon's Dynamo [7] can redistribute keys across nodes to balance load; however this is a heavyweight operation not designed for use when one node becomes a temporary I/O hotspot.

In general data migration allows a storage system to adapt to changes in workload behavior over the long term. It is not suitable for short, upredictable bursts of load where the system cannot predict ahead of time when, how, and what data to migrate. Migrating or re-encoding data during the load burst itself will add additional load to the system. For short unpredictable bursts a better approach is to opportunistically and temporarily balance the load with minimal migration of data, as is done in Everest. Data is not permanently migrated or re-encoded in Everest, but temporarily off-loaded and then reclaimed to the original volume. It is thus complementary to long-term reconfiguration through migration or re-encoding of data. Everest is also transparent to applications and file systems, and hence can be incrementally deployed with minimal changes to server software.

Automatic provisioning: Provisioning tools such as HP's Disk Array Designer [4] generate storage system configurations from workload characteristics and service-level objectives. Such tools can optimize for different goals, such as cost, performance, or power [21]. However workload characteristics must be explicitly specified, perhaps using a language such as Rome [23].

Workloads with high peak-to-mean ratios are a problem for these approaches. Even if the peak levels are known, the only option to get good performance at peak load is to massively over-provision the storage subsystem. This problem is compounded by the rapid increase in disk capacity compared to throughput over time: increasingly throughput is driving the number of disks required in data volumes. Rather than provisioning each data volume for its peak load, Everest exploits the fact that peaks may not be correlated among all the workloads in a data center, and statistically multiplexes storage across workloads for short periods of time. This allows volumes to be provisioned for common-case loads or 99th percentiles rather than worst-case peaks.

Write off-loading: In previous work [15] we showed that write off-loading can save energy by increasing the length of idle periods and allowing disks to spin down. In the current work we re-use much of the same infrastructure — with the addition of N-way replication for fault-tolerance — to address performance degradation due to unbalanced peak loads. The two applications of write off-loading are complementary, and exploit different workload characteristics to achieve different ends.

Log-structured stores: Everest stores are log-structured, in order to be able to efficiently handle a write dominated workload. They are different in design and use from traditional log-structured file systems [17, 19] as well as file system journals, DBMS transaction logs, or block-level write-ahead logs [6]. Unlike a log-structured file system, data is only stored in Everest stores for short periods of time, and in the common case stores do not serve application reads. Free space on the store is created by clients reclaiming and deleting data, rather than by using a log cleaner. Write-ahead logs and journals are used to improve write response times; however they share the same disk resources as the underlying DBMS or file system. By contrast, Everest clients opportunistically use idle bandwidth on other volumes to off-load writes.

Idle disk bandwidth: Everest utilizes idle disk bandwidth for off-load and reclaim. Modern storage subsystems might also use idle disk bandwidth for a variety of background maintenance tasks [12, 22].

Solid-state storage: There have been many proposals for using flash-based solid state memory for storage [3, 9, 16, 25]. This might be in the form of solid-state storage devices (SSDs), or the flash might be added inside the disks, in the RAID controllers, or on the motherboards. The Everest store design could be used with any of these. The circular log design results in writes being sequential and evenly spread over the flash memory, and hence is optimal both for write performance and for wear-leveling [9].

6 Conclusion

Server I/O workloads are bursty, and under peak load performance can degrade significantly. Everest addresses this by pooling idle disk bandwidth into a virtual short-term persistent store to which overloaded volumes can off-load write requests. These volumes can then dedicate their I/O bandwidth to read requests, thereby improving performance. Everest can be interposed transparently at the block I/O level, and unmodified applications can benefit from its use. We have demonstrated the effectiveness of the approach using traces from a production Exchange server, as well as benchmarks.

Acknowledgements: We thank Bruce Worthington, Swaroop Kavalanekar and Chris Mitchell for the production Exchange server traces used in this paper. We also thank our shepherd Kim Keeton and the anonymous reviewers for their feedback.

References

[1] M. Abd-El-Malek, W. V. Courtright II, C. Cranor, G. R. Ganger, J. Hendricks, A. J. Klosterman, M. Mesnier, M. Prasad, B. Salmon, R. R. Sambasivan, S. Sinnamohideen, J. D. Strunk, E. Thereska, M. Wachs, and J. J. Wylie. Ursa Minor: versatile cluster-based storage. In *Proc. USENIX Conference on File and Storage Technologies (FAST)*, San Francisco, CA, Dec. 2005.

[2] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer. Farsite: Federated, available, and reliable storage for an incompletely trusted environment. In *Proc. Symposium on Operating Systems Design and Implementation (OSDI)*, Boston, MA, Dec. 2002.

[3] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy. Design tradeoffs for SSD performance. In *Proc. USENIX Annual Technical Conference*, Boston, MA, June 2008.

[4] E. Anderson, S. Spence, R. Swaminathan, M. Kallahalla, and Q. Wang. Quickly finding near-optimal storage designs. *ACM Transactions on Computer Systems*, 23(4):337–374, November 2005.

[5] T. E. Anderson, M. D. Dahlin, J. M. Neefe, D. A. Patterson, D. S. Roselli, and R. Y. Wang. Serverless network file systems. In *Proc. ACM Symposium on Operating Systems Principles (SOSP)*, Copper Mountain, CO, Dec. 1995.

[6] W. J. Bolosky. Improving disk write performance by logging. Private Communication.

[7] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Proc. ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2007.

[8] J. R. Douceur and W. J. Bolosky. Progress-based regulation of low-importance processes. In *Proc. ACM Symposium on Operating Systems Principles (SOSP)*, Kiawah Island, SC, Dec. 1999.

[9] E. Gal and S. Toledo. Algorithms and data structures for flash memories. *CSURV: Computing Surveys*, 37, 2005.

[10] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *ACM Symposium on Operating System Principles*, Lake George, NY, Oct. 2003.

[11] G. A. Gibson, D. F. Nagle, K. Amiri, J. Butler, F. W. Chang, H. Gobioff, C. Hardin, E. Riedel, D. Rochberg, and J. Zelenka. A cost-effective, high-bandwidth storage architecture. In *Proc. International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, San Jose, CA, Oct. 1998.

[12] R. Golding, P. Bosch, C. Staelin, T. Sullivan, and J. Wilkes. Idleness is not sloth. In *Proc. USENIX Annual Technical Conference*, New Orleans, LA, Jan. 1995.

[13] G. Hunt and D. Brubacher. Detours: Binary interception of Win32 functions. In *Proc. USENIX Windows NT Symposium*, Seattle, WA, July 1999.

[14] E. K. Lee and C. A. Thekkath. Petal: distributed virtual disks. In *Proc. International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Cambridge, MA, Oct. 1996.

[15] D. Narayanan, A. Donnelly, and A. Rowstron. Write offloading: Practical power management for enterprise storage. In *Proc. USENIX Conference on File and Storage Technologies (FAST)*, San Jose, CA, Feb. 2008.

[16] V. Prabhakaran, T. L. Rodeheffer, and L. Zhou. Transactional flash. In *Proc. Symposium on Operating Systems Design and Implementation (OSDI)*, San Diego, CA, Dec. 2008.

[17] M. Rosenblum and J. Ousterhout. The design and implementation of a log-structured file system. In *Proc. ACM Symposium on Operating Systems Principles (SOSP)*, Pacific Grove, CA, Oct. 1991.

[18] Y. Saito, S. Frølund, A. Veitch, A. Merchant, and S. Spence. FAB: building distributed enterprise disk arrays from commodity components. In *Proc. International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Boston, MA, Oct. 2004.

[19] M. Seltzer, K. Bostic, M. K. McKusick, and C. Staelin. An implementation of a log-structured file system for UNIX. In *Proc. USENIX Winter Conference*, San Diego, CA, Jan. 1993.

[20] D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3):652–686, 1985.

[21] J. D. Strunk, E. Thereska, C. Faloutsos, and G. R. Ganger. Using utility to provision storage systems. In *Proc. USENIX Conference on File and Storage Technologies (FAST)*, San Jose, CA, Feb. 2008.

[22] E. Thereska, J. Schindler, J. Bucy, B. Salmon, C. R. Lumb, and G. R. Ganger. A framework for building unobtrusive disk maintenance applications. In *Proc. USENIX Conference on File and Storage Technologies (FAST)*, San Francisco, CA, Mar. 2004.

[23] J. Wilkes. Traveling to Rome: QoS specifications for automated storage system management. In *Proc. International Workshop on Quality of Service (IWQoS)*, Karlsruhe, Germany, June 2001.

[24] J. Wilkes, R. Golding, C. Staelin, and T. Sullivan. The HP AutoRAID hierarchical storage system. *ACM Transactions on Computer Systems*, 14(1):108–136, February 1996.

[25] D. Woodhouse. JFFS: The journalling flash file system. In *Proc. The Linux Symposium*, Ottawa, Canada, July 2001.

[26] B. Worthington and S. Kavalanekar. Characterization of storage workload traces from production Windows servers. In *Proc. IEEE International Symposium on Workload Characterization (IISWC)*, Seattle, WA, Sept. 2008.