Efficient Logging and Replication Techniques for Comprehensive Data Protection

Maohua Lu Shibiao Lin Tzi-cker Chiueh

Computer Science Department
Stony Brook University
{mlu,slin,chiueh}@cs.sunysb.edu

Abstract

Mariner is an iSCSI-based storage system that is designed to provide comprehensive data protection on commodity ATA disk and Gigabit Ethernet technologies while offering the same performance as those without any such protection. In particular, Mariner supports continuous data protection (CDP) that allows every disk update within a time window to be undoable, and local/remote mirroring to guard data against machine/site failures. To minimize the performance overhead associated with CDP, Mariner employs a modified track-based logging technique that unifies the long-term logging required for CDP and short-term logging for low-latency disk writes. This new logging technique strikes an optimal balance among log space utilization, disk write latency, and ease of historical data access. To reduce the performance penalty of physical data replication used in local/remote mirroring, Mariner features a modified two-phase commit protocol that in turn is built on top of a novel transparent reliable multicast (TRM) mechanism specifically designed for Ethernet-based storage area networks. Without flooding the network, TRM is able to keep the network traffic load of reliable N-way replication roughly at the same level as the no-replication case, regardless of the value of N. Empirical performance measurements on the first Mariner prototype, which is built from Gigabit Ethernet and ATA disks, shows that the average end-to-end latency for a 4KByte iSCSI write is under 1.2msec when data logging and replication are both turned on.

1. Introduction

As modern enterprises increasingly rely on digital data for continuous and effective operation, data integrity and availability become the critical requirements for enterprise storage systems. Replication is a standard technique to improve data integrity and availability, but typically incurs a performance overhead that is often unacceptable in practice. This paper describes the design, implementation and evaluation of an iSCSI-based storage system called *Mariner*, which aims to support comprehensive data protection while reducing the associated performance overhead to a minimum. More specifically, *Mariner* uses *local mirroring* to protect data from disk and server failures, and *remote replication* to protect data from site failures. In addition, *Mariner* keeps the before image of every disk update to protect data from software failures, human errors or malicious attacks.

A *Mariner* client, for example a file or a DBMS server, interacts with three iSCSI storage servers: a master storage server, a local mirror storage server and a logging server. When a *Mariner* client writes a data block, the write request is sent to these three servers. The data block is synchronously committed on the logging server, and then asynchronously committed on the master server and the local mirror server. In addition, the logging server is responsible for remote replication, which is also done asynchronously. When a *Mariner* client reads a data block, the read request is only sent to the master server, which services the request on its own.

Mariner supports block-level continuous data protection (CDP), which creates a new version for every disk write request and thus allows roll-back to any point in time. As more and more data corruption is caused by software bugs, human errors and malicious attacks, CDP provides a powerful primitive for system administrators to correct the corrupted data. *Mariner*'s logging server is responsible for archiving the before image of every disk write within a period of time (called the *protection window*) so that it can undo the side effects of any disk write in the protection window.

24th IEEE Conference on Mass Storage Systems and Technologies (MSST 2007)
0-7695-3025-7/07 $25.00 © 2007 IEEE

To reduce the performance penalty associated with CDP and data replication, *Mariner* modifies the track-based logging (*Trail*) technique [1]. *Trail* was originally designed to reduce the write latency of locally attached disks and adapts the idea to the network storage environment where *Mariner* operates. The original *Trail* requires a log disk in addition to a normal disk, which hosts a write-optimized file system. By ensuring that the log disk's head is always on a free track, *Trail* could write the payload of a disk write request to wherever on the track the disk head happens to be. Once this write is completed, *Trail* returns a completion signal so that the high level software can proceed. Therefore, the latency of a synchronous disk write is reduced to only the sum of the *controller processing time* and the *data transfer delay*. However, the original *Trail* design is inadequate for *Mariner* for two reasons. First, the disk utilization efficiency of the design is too low to meet CDP's demanding log space requirement. Second, the design switches a disk's head to the next free track after servicing a request and thus incurs substantial disk switching costs. To address these problems, *Mariner* makes four modifications to *Trail*. First, after the payload of a logical disk write (W1) is written to a log disk, the payload is kept for a sufficiently long period of time so that the following write (W2) against the same data block can be undone T days after W2 is performed. Here T is the length of the protection window, and the payload of W1 is the before image of W2. Second, *Mariner* batches multiple logical disk write requests that arrive within an interval as much as possible into a physical disk write in order to amortize the fixed overhead associated with each physical disk write, and allows multiple physical disk writes to be written to a track until the track's utilization efficiency exceeds a certain threshold. Third, *Mariner* exploits an array of log disks to amplify both the log capacity and the effective logging throughput in terms of physical I/O rates. Finally, *Mariner* uses a modified version of two-phase commit protocol to propagate the effect of each logical disk write consistently to all replicas while minimizing the write latency visible to the software.

In addition to involving more servers, N-way data replication also introduces N times as much load on the storage client's CPU, memory and network interface. It is possible to move this load to the storage area network using special hardware such as SANTap [2], which replicates a disk write request coming into a SAN switch across multiple predefined ports in a way transparent to the storage client. However, this approach requires special and proprietary hardware support. *Mariner* uses a software-only approach called *Transparent Reliable link-layer Multicast (TRM)* to approximate the hardware-based in-network replication supported by SANTap. More specifically, TRM achieves in-network replication by exploiting link-layer tree-based multicast available in modern commodity Ethernet switches.

Modified *Trail* and TRM together enables *Mariner* to provide comprehensive data protection at a performance cost that is almost negligible when compared with vanilla iSCSI storage servers without any protection. In Section 2, we review the previous work on related logging and replication technologies in the context of network storage systems. Section 3 gives an overview of *Mariner*'s system architecture. In Section 4, 5 and 6, we describe the design and implementation of track-based logging, modified two-phase commit and TRM, respectively. In Section 7, we present the evaluation methodology together with experiment result and analysis. Finally, we summarize this paper with the main research contributions in Section 8.

2. Related Work

Continuous Data Protection (CDP) backups the data on-the-fly as it is written to the disk. Therefore, every update is undoable. Traditional data protection system relies on file system backup, which is performed in a much coarser granularity than CDP. Using a CDP based solution in network storage could result in an increase in network and individual server resource load since all the data written to master storage node have to be backed up to the local mirror storage nodes at the same time.

Parallax [3] also adopts block-level *Copy On Write(COW)* semantics to minimize data copies between different *Virtual Disk Image(VDI)*s in large-scale distributed environments. It focuses on the management of a large number of Virtual Machine(VM)s and does not consider preserving data over a long term. It employs a radix tree to provide ready access to one historical image and a group of radix trees are organized in a separate data structure. In contrast, Mariner uses an External B-Tree to preserve the full mapping of the logical address plus a timestamp to the physical address.

The idea of the disk head prediction used in Trail is not new. One example is the *Free block scheduling* [4, 5]. The objective of Free block scheduling is to piggy-back background media transfer with normal workload activity with little-to-no overhead by utilizing the rotational and seek latency of the requests belonging to the normal workload. A freeblock scheduler predicts the amount of the rotational latency before the next foreground media transfer and inserts background media transfer within the anticipated latency to minimize the impact on the foreground media transfer. The key point for the feasibility of free block scheduling is that the ordering of requests for background disk activities is not mandatory for background activities.

In the original Trail architecture [1], there is a normal disk, which holds the user data, and a log disk, which

provides a fast staging buffer for disk writes. Given a disk write request, Trail first writes its payload to the log disk, and then completes the write to the normal disk asynchronously. Because Trail ensures that the disk head is always on an empty track and could accurately estimate the disk heads position in real time, it can write a piece of data to the log disk where the disk head happens to be at that instant. Consequently, each write operation incurs very little rotational latency and zero seek delay.

Fibre Channel (FC) is the predominant storage area networking technology. It evolved as an alternative data transfer technology to the low performance 10 Mbps Ethernet technology. With the advent of Gigabit Ethernet, the initial concerns of bandwidth and latency requirements of SANs no longer remain a core issue. The ultimate enabler of Ethernet-based SAN is the iSCSI [6] protocol, which defines semantics for block level SCSI I/O over any IP network. iSCSI is the FCP counterpart on Ethernet networks that maps the SCSI command set to the TCP/IP stack. The increase in the momentum of iSCSI is evident from its availability through several major operating system vendors and the availability of iSCSI enabled HBAs from major hardware vendors. Although it is debatable whether iSCSI/Ethernet will completely replace Fibre Channel SAN or not, the increasing momentum of Ethernet based SANs is undeniable.

The memory to memory approach has been widely applied to avoid data copy [7] [8] [9]. Remote Direct Memory Access(RDMA) is a zero-copy networking technique, permitting data to be transferred directly from application memory of one machine to that of another machine without involvement of host CPU processing, caches and context switches of host Operating Systems. These features are especially important in highly parallel networking systems such as clustered computing [10]. Although technically superior to other alternatives, RDMA is not widely advocated in network storage systems because its most common underlying infrastructure is InfiniBand [11], a point-to-point switch fabric interconnect technology not widely used. In contrast, iSCSI operates seamlessly on Ethernet networks, which is the most widespread LAN technology in use till now.

Payload caching [12] and Network-centric buffer caching [13] share similarity with RDMA in the sense that they all aim to minimize the data copying. Payload caching caches payload in Network Interface Card (NIC) and reduces data traffic through host I/O bus. Network-centric buffer caching keeps a network friendly format in page/buffer cache to avoid data content copying and transformation overhead. As *TRM* aims to minimize traffic load on Ethernet network, these techniques are orthogonal with *TRM*. Compared with these techniques, *TRM* is much more favorable for applications that requires data replications.

Cisco SANTap [14] is a protocol that sits between the MDS switch and a storage application appliance. The SANTap service registers as both an initiator (host) and a target device (storage array) in the Fibre Channel name server. It allows the storage appliance to get a copy of the I/O exchange between the server and storage without compromising the primary I/O. The appliance is no longer in the data path. In addition, SANTap also needs to provide error recovery services to permit recovery in the event of appliance or port failure. However, special hardware support is required to use the SANTap approach. Our *TRM* approach achieves similar functionality on Ethernet switches and leverages TCP to make sure of reliable data replication.

The Viking project [15] revisits architecture features of Ethernet technology when Ethernet is applied to network storage and large scale Metropolitan Area Network(MAN). Viking overcomes one efficiency weakness of Ethernet technology by extending a single spanning tree to multiple spanning trees in forwarding routed data packets by leveraging on standard Virtual LAN technology. Viking improves the efficiency of underlying Ethernet-based network by making a better use of underlying data link capacities and reducing the down time of link failures . In contrast, *TRM* targets at minimizing network traffic passing through NICs and therefore gains better network efficiency.

Link-layer multicast could be implemented by exploiting IGMP snooping [16]. Traditionally, Ethernet switches treated link-layer multicast packets as broadcast packets. Network performance suffers as unnecessary packets are forwarded through the network. Fortunately, most modern Ethernet switches, particularly Gigabit switches, support a feature called IGMP snooping, which was designed to support IP multicast without using link-layer broadcasting. *TRM* makes a novel use of IGMP snooping to implement the link-layer multicast.

3. System Architecture

As shown in Figure 1, a *Mariner* storage system consists of six types of storage nodes. A *client* node, which could be a file or database server, accesses data in a virtual storage device through the iSCSI protocol. The current data of a virtual storage device is stored on a master *storage* node, and replicated on a local mirror *storage* node. The virtual storage device's historical versions are maintained on a *logging* node (called Trail node from this point on), which also serves as a control gateway for remote replication. Data writes are first committed to *remote logging* nodes and then propagated to *remote storage* nodes. *Manager* node is used for system configuration, administration, monitoring and failure recovery. A typical *Mariner* system contains multiple client nodes, storage nodes, Trail nodes, remote logging nodes and remote storage nodes, but only

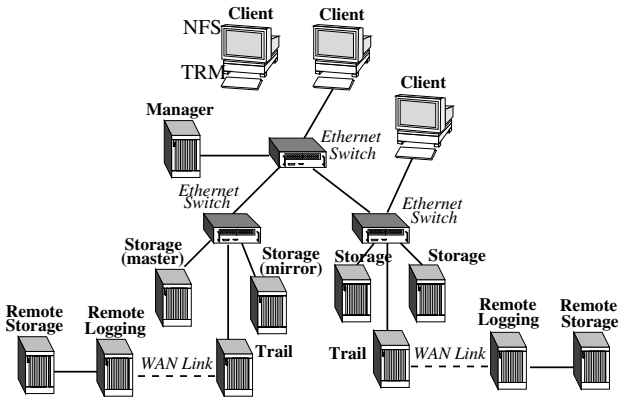

Figure 1. *A Mariner storage system consists of six types of nodes:* client *nodes that issue data access requests,* manager *nodes for system configuration and administration,* storage *nodes that hold local replicas of current data,* Trail *or* logging *nodes that maintain historical data and serve as a gateway for remote replication, and remote logging/storage nodes that keep a remote copy of current data.*

one manager node. A Trail node can be shared by multiple master and mirror nodes.

With CDP, *Mariner* allows users to roll back a virtual storage device to any point within the protection window. Users can only read and write the current or read any historical snapshot of a virtual storage device. To maintain the file system consistency for a particular point-in-time storage snapshot, *Mariner* may need to perform a fsck-like recovery procedure on the snapshot to return a storage view with consistent file system metadata. This recovery procedure needs to modify a historical storage snapshot, but the associated disk writes are held in a temporary buffer and are thrown away when the snapshot is no longer needed.

Read requests for the current data on a virtual storage device are serviced by its associated storage nodes. Write requests for the current data on a virtual storage device are serviced by its associated Trail node and storage nodes. More specifically, a logical disk write request is first sent to the corresponding Trail node, which logs it to disk and returns an OK reply to the requesting client. Then the client writes it to one or multiple storage nodes, depending on the degree of local mirroring supported. As far as a *Mariner* client is concerned, a disk write is completed when it receives an OK reply from the Trail node. Because of track-based logging, *Mariner* clients experience very low disk write latency. To reduce the performance penalty associated with sending a disk write's payload to multiple nodes, *Mariner* uses TRM to duplicate the payload packet in the network.

The Trail node of a virtual storage device services all read and write requests for that device's historical data, and batches multiple disk writes to replicate them to a remote site more efficiently. Because of space constraints, the details of remote replication are omitted in this paper.

4. Low-Latency Disk Array Logging

The original Trail design [1] moves the log disk's head to the next track after each write operation to ensure that the disk head is always on an empty track. Therefore, the log records are contiguous on a track-by-track rather than byte-by-byte basis, hence the name track-based logging. This per-write disk head movement incurs a track-to-track seek delay for every write operation, and results in low disk space utilization. The modified Trail design allows multiple physical writes per track and uses an array of log disks to further mask track-to-track seek delays.

Mariner maintains a disk request queue for each log disk. At any point in time, one of the log disks serves as the active disk. In the beginning, *Mariner* randomly chooses one of the log disks as the active disk. Once a log disk becomes the active disk, it remains as the active disk until the waiting time of the oldest pending request exceeds a threshold, T_{wait}. Whenever a new logical disk write request arrives at a Trail node, *Mariner* inserts the request to the active disk's queue as long as the waiting time of its oldest pending request is smaller than T_{wait} and there is enough free space in the current track to accommodate the new request; otherwise *Mariner* dispatches the request batch in the active disk's queue, and chooses another log disk as the active disk and inserts the request to its queue.

To choose a new active disk for an incoming write request, *Mariner* computes the time at which the write request could be written to each log disk, and selects the one that can write the request to disk at the earliest. When computing an incoming disk write request's write time on a log disk, *Mariner* takes into account the current position of the log disk's head and the possibility of batching the request with others already in the disk's queue. For those log disks that are currently idle, *Mariner* only needs to consider the delay due to batching.

A key design decision in *Mariner* is to encourage batching of multiple logical disk writes into one physical disk by dispatching a new write request to the active disk, rather than to the disk with the earliest write time for that request. As we will show in Section 7, this design choice significantly increases *Mariner*'s batching efficiency and thus effective throughput.

For every logical disk write, *Mariner* creates a log record that contains the write's Logical Block Address (LBA), timestamp and payload, and writes it to the log disk chosen for the request. To facilitate accesses to historical data, *Mariner* maintains an index structure to map a disk block's logical block number and a timestamp to the physical block number of the corresponding historical ver-

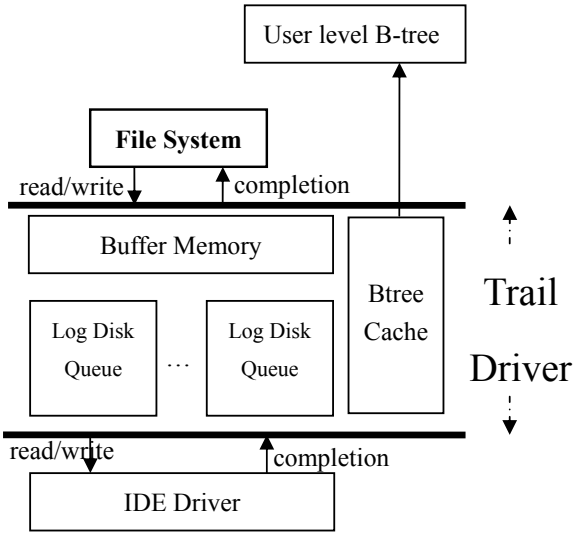

Figure 2. *The software architecture of* Mariner*'s Trail node. The Trail module, which sits between the file system and the physical disk driver, manages a disk block buffer cache, a B-tree cache, and a set of disk request queues, one for each log disk. The user-level B-tree daemon maintains the index tree for mapping a disk block's LBA and timestamp to its corresponding physical block.*

sion. This index data structure is maintained by a user-level daemon and organized as a B-tree residing on a different disk, and contains only the log records of those logical disk writes in the protection window. Because the log record of each logical disk write is self-contained, *Mariner* can reconstruct the index tree by scanning the log disks. Therefore, *Mariner* can afford to batch updates to the index tree due to disk writes and perform them asynchronously.

Trail is currently implemented under the Linux 2.6 kernel as a virtual device driver between the file system and the physical disk driver, as shown in Figure 2. It dispatches logical disk write requests to the per-log-disk request queues, maintains a disk block buffer cache to facilitate the service of current data accesses, and a B-tree cache to facilitate the look-up of historical versions of disk blocks. To implement track-based logging, *Mariner* statically extracts the physical disk geometry information from every log disk, and then uses a disk head position estimation algorithm to predict each log disk's disk head position at run time. More concretely, after a physical disk write is completed, *Mariner* records the LBA of its last sector, LBA_0, and its completion timestamp T_0. Assuming the disk head stays in the same track, when the next write arrives at T_1, *Mariner* estimates the disk head's current position *CurrentLBA* using the following formula:

$$CurrentLBA = SPT \cdot \frac{(T_1 - T_0)\ mod\ RoTime}{RoTime} + LBA_0 \quad (1)$$

where *SPT* is the number of sectors in the current track, *RoTime* is the disk's full rotation time. The final predicted position, *DestinationLBA*, is $CurrentLBA + Lookahead$ to account for such delay as the controller delay. *Lookahead* is an empirical value chosen to avoid a full rotation. For the IBM Deskstar DTLA-307030 disk, this value is set to be 22 sectors. The accuracy of the above disk head position estimation algorithm decreases with the value of $T_1 - T_0$. To ensure the algorithm's accuracy is always adequate, *Mariner* issues additional dummy disk reads to guarantee that $T_1 - T_0$ is always below a threshold, T_{idle}, even when the input load is low.

To satisfy CDP's log space requirement, *Mariner* allows multiple physical writes to go to the same track in order to use the log disks' space more efficiently. However, higher log disk space utilization efficiency means longer rotation latency because it is less likely that when a new write request arrives at a log disk's queue, the disk's head happens to be on a sufficiently large free region that can hold it. To determine when to switch a log disk's head to the next free track, *Mariner* uses the following metric to gauge the degree of fragmentation of the current track:

$$F = \frac{ServicedReqNum}{10 \cdot (1 - Utilization)} \quad (2)$$

where *ServiceReqNum* is the number of write requests already written to the current track and $Utilization$ is the percentage of the current track that is already occupied. The larger the values of *ServiceReqNum* and $Utilization$, the more fragmented the current track. As *Mariner* can self-describe its data logging, *ServiceReqNum* can be extracted from the hard drive in case of crashes. After a log disk services a physical write request, *Mariner* computes its current track's fragmentation metric. If the metric's value exceeds a pre-defined threshold, T_{switch}, and its request queue is empty, *Mariner* issues a seek command to move the disk's head to the next track. To minimize the delay of the track-to-track seek, the destination LBA of the seek command (which is a write command for IDE drives because IDE drives only support two operations, read and write) is set to $CurrentLBA + CurrentSPT$, where *CurrentSPT* is *SPT* of the current track.

5. Trail-based Asynchronous Replication

Mariner leverages Trail's low-latency disk write capability and a modified two-phase commit protocol to replicate data asynchronously without compromising data integrity. Figure 3 shows the message sequence used in this modified two-phase commit protocol. The *Mariner* client issuing a logical disk write request serves as the coordinator, and the Trail, master and local mirror nodes are the participants. The client first sends the write request to the

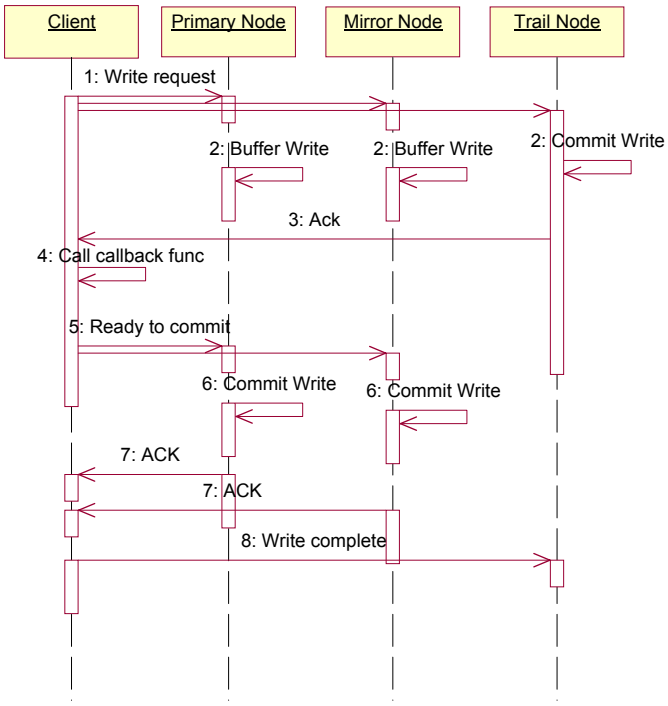

Figure 3. *The message sequence used in the modified two-phase commit protocol when there is no device failure.*

Trail, master and local mirror nodes of its virtual storage device. Upon receiving this request, the Trail node immediately commits the request to its log disk and sends an ACK back to the client after it is done, but the master and local mirror nodes simply buffer this request, waiting for further instruction. When the client receives the Trail node's ACK, it notifies the master and local mirror nodes to commit the buffered write request, and resumes the thread that issues the write request by invoking the associated callback function. The master (local mirror) node sends back an ACK after completing the write request to disk. Finally, the client asynchronously informs the Trail node about each write request's completion status on the master and local mirror nodes, so that the Trail node can keep track of their progress. Whenever possible, the messages of this modified two-phase commit protocol are piggy-backed with normal iSCSI command packets. In addition, the protocol has built in extensive retry mechanisms to deal with such failures as packet loss, message corruption, TCP connection time-out and iSCSI connection time-out.

This modified two-phase commit protocol is different from the standard two-phase commit protocol because its goal is to commit a write request on as many participant nodes as possible, rather than to achieve all-or-nothing consistency among participants. Therefore, the coordinator does not need to collect ACKs from all participants before committing a write request. Instead, it keeps a record of who has committed which requests so that after a failed node recovers, the system knows how to replay which write requests to bring it to synchronization with others. As the client informs the Trail node about the other two nodes' write progress, the other two nodes snoop the network and also each keep a local write progress log about others. When the Trail node is alive, it is the Trail node's write progress log that serves as the ground truth. When the Trail node is dead, it is master node's write progress log that serves as the ground truth.

When the master node dies, the local mirror node becomes the master node, and each write request is sent to the new master node and the Trail node; after the old master node recovers, it contacts the Trail node, which keeps track of each node's write progress, to replay missing write requests, and becomes the local mirror node. When the local mirror node dies, each write request is sent to the master node and the Trail node; after the local mirror node recovers, it contacts the Trail node to replay missing write requests, and continues to be the local mirror node. When the Trail node dies, CDP and remote replication cease to function, and each write request is sent to the master and local mirror nodes; after the Trail node recovers, it contacts the master node for synchronization and continues to act as a logging disk.

6. Transparent Reliable Multicast (TRM)

When a *Mariner* client sends a write request to the Trail, master and local mirror nodes in the modified two-phase commit protocol, it uses TRM to reliably multicast the write request and achieve almost the same network efficiency as the no-replication case.

6.1. Multicast Transmission of Common Payloads

Logically, TRM is a software layer residing below the TCP/IP stack that constantly monitors the contents of outgoing TCP connections to look for common bytes. When packets from a set of TCP connections share common bytes, TRM sends only one of them as an Ethernet multicast packet to the destination nodes associated with these connections. The software architecture of iSCSI-based TRM is shown in Figure 4. There are two key components in TRM: (a) the client side component monitoring TCP connections for common data payload and constructing multicast packets that carry these common payload, and (b) the server side component reconstructing the original TCP streams based on the payloads and headers received.

The client-side TRM component of the current *Mariner* prototype includes an iSCSI parser that tracks iSCSI commands in iSCSI-carrying TCP connections. Once detecting common write payloads among the three iSCSI con-

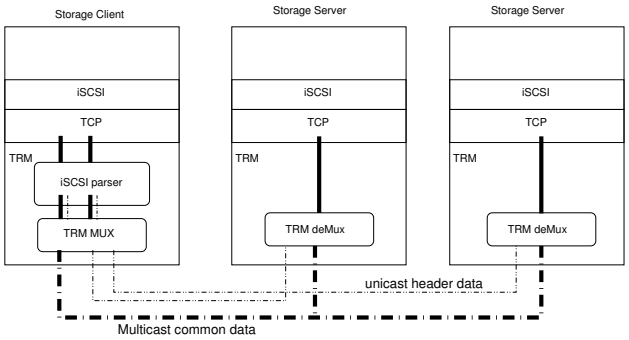

Figure 4. *Data flow of an iSCSI-based TRM system supporting 2-way replication. An iSCSI protocol parser keeps track of contents in TCP connections corresponding to the two iSCSI sessions involved in data replication. The first iSCSI copy associated with each SCSI write request is sent as multicast packets, whereas the headers of the second copy are merged and sent as a unicast packet. The TRM layer at the receivers reconstructs each individual TCP stream based on the received unicast and multicast data.*

nections to a virtual storage device's Trail, master and local mirror nodes, it asynchronously merges the three write requests sharing the same payload by sending the earliest-arriving copy of each iSCSI write request using multicast and the headers of the other two copies as unicast packets to their corresponding nodes. Asynchronous merging does not require the three TCP connections involved in data replication to be strictly synchronized.

The server-side TRM component reconstructs the individual TCP streams by taking the common payloads, which are received as multicast packets, and headers, which are received as unicast packets, putting them together into original unicast TCP packets, and passing them up to the TCP/IP stack for further processing.

When there is packet lost, TRM relies on TCP to retransmit the lost packets and therefore does not require any additional machinery to support reliable transmission. Retransmitted packets are always transmitted as unicast packets. As a result, packet retransmission may cause the TCP connections being merged to become de-synchronized.

6.2. Common Payload Detection

A *Mariner* client sends each iSCSI read request only to the master node, but sends each iSCSI write request to the Trail, master and local mirror nodes. Therefore, the TCP connection associated with the master storage node contains more iSCSI commands than the two TCP connections associated with the local mirror and Trail node. Because TCP is a stream protocol and does not preserve application-level packet boundaries, packet-by-packet comparison may not be able to reliably detect all common payloads among

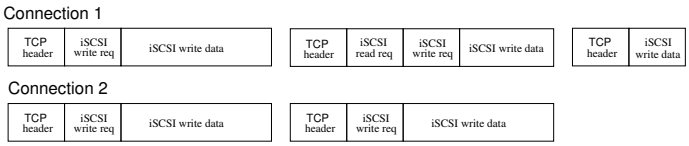

Figure 5. *Because Connection 1 contains both READ and WRITE commands and Connection 2 contains only WRITE commands, packet-by-packet comparison between these two connections cannot detect the payloads of WRITE commands.*

connections, as shown in Figure 5, which calls for a more expensive byte-by-byte comparison approach to detect common payloads.

To reduce the performance overhead associated with common payload detection, *Mariner* exploits protocol-specific knowledge. More concretely, the current *Mariner* prototype parses the iSCSI commands in each of the three TCP connections and is able to pinpoint the precise location of the payload portion of each iSCSI write request. From these locations, the *TRM* layer can easily detect common payloads without resorting to expensive byte-by-byte comparison.

7. Performance Results and Analysis

7.1. Evaluation Methodology

We first evaluate each component of *Mariner*, including Trail, modified two-phase commit, and TRM, and then the entire system as a whole. We use synthetic workloads to stress-test each *Mariner* component and real traces to evaluate *Mariner*'s end-to-end performance. The four traces used in this study include both file system and database workload:

1. IO Trace

    - Lair62b The original Lair62b is an NFS RPC trace collected on an NFS server by the SOS project of Harvard University [22]. This trace is converted into a block-level disk access trace through an FFS-like file system simulator, which models the I-node and data blocks and ignores other meta-data [23]. The block size is 4KB and the trace is a one-day long trace with 12631475 requests, 2816401 of which are writes.
    - OLTP(On-Line Transaction Processing) OLTP trace is a database buffer cache access trace collected on an IBM DB2 database running IBM's TPCC benchmark of 1,000 warehouses [23]. The trace is featured by a large amount of random access. The block size is 4KB.

- DSS(Decision Support System) DSS trace is another database buffer cache access trace collected on an IBM DB2 database running IBM's TPCH benchmark [23]. The trace contains several large sequential scan of a big table. The block size is 4KB.
- Cello99 Cello99 is a low-level disk I/O trace collected from a HP UNIX platform. Since the trace is filtered by the file system cache, the spatial locality is quite poor. The block size is 8KB.
- MS-SQL-Large I/O trace MS-SQL-Large trace is a disk I/O trace collected from a Microsoft SQL database server running the standard TPC-C benchmark for two hours. The TPC-C database consists of 256 warehouses and occupies around 100 GBytes of storage excluding log disks. The trace is filtered by a 1 GByte SQL server cache. The block size is 4KB and the trace has 5390743 requests, 866029 of which are write ones.
- MS-SQL-Small I/O trace This trace is collected with the same setup as the previous trace except that the server cache is 64 MB.

2. File-level Trace

   - Postmark Postmark [24] is a file system benchmark emulating very heavy small file workload. The benchmark creates a specified number of files, performs various file system operations and finally deletes those files. For all runs, we run Postmark with 10,000 files, 1000 subdirectories and 50,000 transactions.
   - Lair Trace played by TBBT [25] trace player. NFS server is the Trail client side. It is the same trace as Lair62b, the only difference is it is played at NFS level. It is a one-day trace on Oct 21, 2001 worth of 2GB data in total.

The testbed used in this study consists of one client node, a Trail node, a master node and a local mirror node, all of which are connected by a Netgear GS508T 8-port Gigabit Ethernet switch. The Trail node is a Dell PowerEdge 600SC machine with an Intel 2.4 GHz CPU, 768 MB memory, a 400 MHz front-side bus, an embedded Gigabit Ethernet Card, and up to five ATA/IDE hard disks, each of which is a 80-GB IBM Deskstar DTLA-307030 disk. The master node, the local mirror node and the client node are PowerEdge SC1425 machines with an Intel 3.8 GHz CPU, 1 GB memory, a 800 MHz front-side bus and four embedded Gigabit Ethernet Cards. We use UNH iSCSI implementation (version 1.6.0) [26] on the iSCSI initiator side and Linux's iSCSI Enterprise Target (*IET*) implementation on the iSCSI target side. Note that in the `fileio` mode of the *IET* implementation, each write request is synchronous as a sync-like function is called after each write operation. In terms of performance metrics, we measure the average write latency and the I/O rate of each test run.

In this study, we first evaluate the basic track-based logging technique as this is the first time this technique is implemented on a commodity IDE/ATA drive. Then we examine the write latency of a Trail node that uses an array of log disks and supports multiple writes per track, and impacts of various configuration parameters. Next, we evaluate the effectiveness of TRM in terms of its savings in network load. Finally, we measure the end-to-end write latency of a logical disk write request under *Mariner*, which includes the effects of Trail, two-phase commit and TRM.

7.2. Array of Logging Disks

Mariner's Trail node uses an array of log disks, rather than a single log disk. This subsection evaluates the effectiveness of *Mariner*'s disk request dispatching algorithm in exploiting request batching to improve the I/O rate without compromising the write latency. In this experiment, there are five log disks and each log disk is a commodity IDE/ATA hard drive connected via an independent ATA/IDE channel from the Promise Ultra100 TX2 IDE controller. We issue additional disk access requests to guarantee that the maximal time interval between consecutive accesses to each disk is at most 50msec. In addition, the maximum waiting time in the disk request queue is 0.3msec, T_{switch} is set to 6 to achieve reasonable disk utilization efficiency. Under this setup, a stand-alone Trail device can deliver 1.8msec write latency and achieve 70% disk space utilization under an input workload of 12500 writes/sec and 4KB per write request.

Batching of multiple logical writes into a physical write improves *Mariner*'s physical write efficiency and thus its effective throughput. Batching is especially useful in the face of a burst of write requests. However, batching increases the write latency because it forces those requests that arrive early to wait even when the disk is idle. To resolve this issue, *Mariner* sets a limit on a request's wait time (T_{wait}) when it batches logical write requests. Table 1 shows the impact of T_{wait} on the log disk array's write latency. The workload used in this experiment is a synthetic workload that consists of 4KB write requests with a fixed inter-request interval, 60usec. When T_{wait} is set to zero, there is not much room for request batching, and the batch size, i.e., the average number of logical writes per physical write, is small, around 3 or 12KB. Smaller batch size leads to lower I/O rate for the log disk array, and eventually causes subsequent write requests to queue up and experience higher latency. On the other hand, when T_{wait} is set

Wait Time Limit (msec)	Batch Size (KB)	Write Latency (msec)
0	12	6.7
0.12	12.7	5.8
0.24	15	3.1
0.36	20	1.9
0.48	22	2.1
0.60	24	2.3
0.72	28	2.4
1	32	3.0

Table 1. *Impact of wait time limit on the batching efficiency and average write latency, where the request size is 4KB and $T_{switch} = 6$.*

to 0.36msec, the resulting batch size is larger, the log disk array's I/O rate improves, and the average write latency actually decreases. This result demonstrates that it is better to force requests to wait a little bit longer out front in order to improve the batching efficiency and eventually decrease the write latency for everybody. However, as T_{wait} is increased beyond 0.36msec, the write latency starts to increase again, because each request is likely to wait longer and each physical write also takes longer to complete.

7.3. Sensitivity Study

In this subsection, we study the performance impact of each configuration parameter in *Mariner*'s track-based logging design. There are 4 configuration parameters: (1) the threshold of the fragmentation metric T_{switch}, (2) the disk head recalibration interval (T_{idle}), (3) the wait time limit for batching (T_{wait}) and the number of log disks in the array. Unless specified otherwise, the following parameter settings are used by default: $T_{idle} = 50msec, T_{switch} = 12, T_{wait} = 0.2msec$, and the number of disks is 5.

We use a synthetic workload to feed into *Mariner*'s Trail node by varying the inter-request interval until reaching the maximum throughput of the log disk array. The synthetic workload contains $20,000$ write requests of 4 KBytes and there is no read request, and the write latency from the device driver is measured. We use six different inter-request interval values to generate six different input request rates: 0.08msec, 0.1msec, 0.12msec, 0.14msec, 0.16msec and 0.18msec.

T_{switch} determines when to switch a disk's head to the next track and thus plays an important role in the tradeoff between disk write latency and disk space utilization efficiency. Every curve in Figure 6(a) has up to six measurements, which from left to right correspond to the six inter-request intervals in decreasing order. We stop decreasing the inter-request interval as soon as the measured latency exceeds 2msec. For a given T_{switch}, as the input request rate increases (or inter-request interval decreases), each physical write batches more logical writes, and the disk utilization efficiency improves because Equation (2) is based on the number of physical writes and the same number of physical writes can pack more bytes when batching is more effective. However, improved disk utilization efficiency worsens the average write latency, because each physical write is larger and takes longer to complete. For a given input request rate, as T_{switch} increases, the disk utilization efficiency improves significantly without degrading the average write latency too much. This result empirically justifies one of the key design decisions in *Mariner*: allowing multiple physical disk writes per track.

Each curve in Figure 6(b) shows the latency and throughput of a given number of log disks when the inter-request interval decreases from 0.18msec to 0.08msec from left to right. For a fixed number of log disks, increase in the input request rate increases both their throughput and latency because batching is more effective and the size of each physical write is bigger. For a given input request rate, as the number of log disks increases, the throughput increases linearly and the latency remains largely unchanged. For example, when the inter-request interval is 0.1msec, the 1-disk configuration can achieve a throughput of 1000 disk writes operations per second (OPS) with an average write latency of 1.4msec, and the 2-disk configuration can can achieve a throughput of 2000 disk writes operations per second (OPS) with the same average write latency. This linear improvement comes from the fact that multiple disks can mask the disk head switch delays of individual disks as well as provide higher aggregate raw transfer bandwidth.

Again each curve in Figure 6(c) has up to six measurements, which from left to right correspond to the six inter-request intervals in decreasing order, and we stop decreasing the inter-request interval as soon as the measured latency exceeds 2msec. For a given T_{wait}, as the input request rate increases, the throughput of the log disks increases because batching is more effective, and the average write latency grows because each physical write is larger and takes longer to complete. For a given input request rate, increase in T_{wait} improves the batching efficiency, which in turn increases the throughput and the average write latency of the log disks. From the results, 0.2msec seems to be a good choice for T_{wait} to achieve a reasonable tradeoff between disk write latency and incurring reasonable write latency overhead.

Figure 6(d) shows the performance impact of the recalibration frequency on the average write latency. Increase in the recalibration frequency improves the accuracy of disk head position prediction and thus reduces the rotational latency of disk writes. However, increase in the recalibration frequency also introduces additional load to the log disks

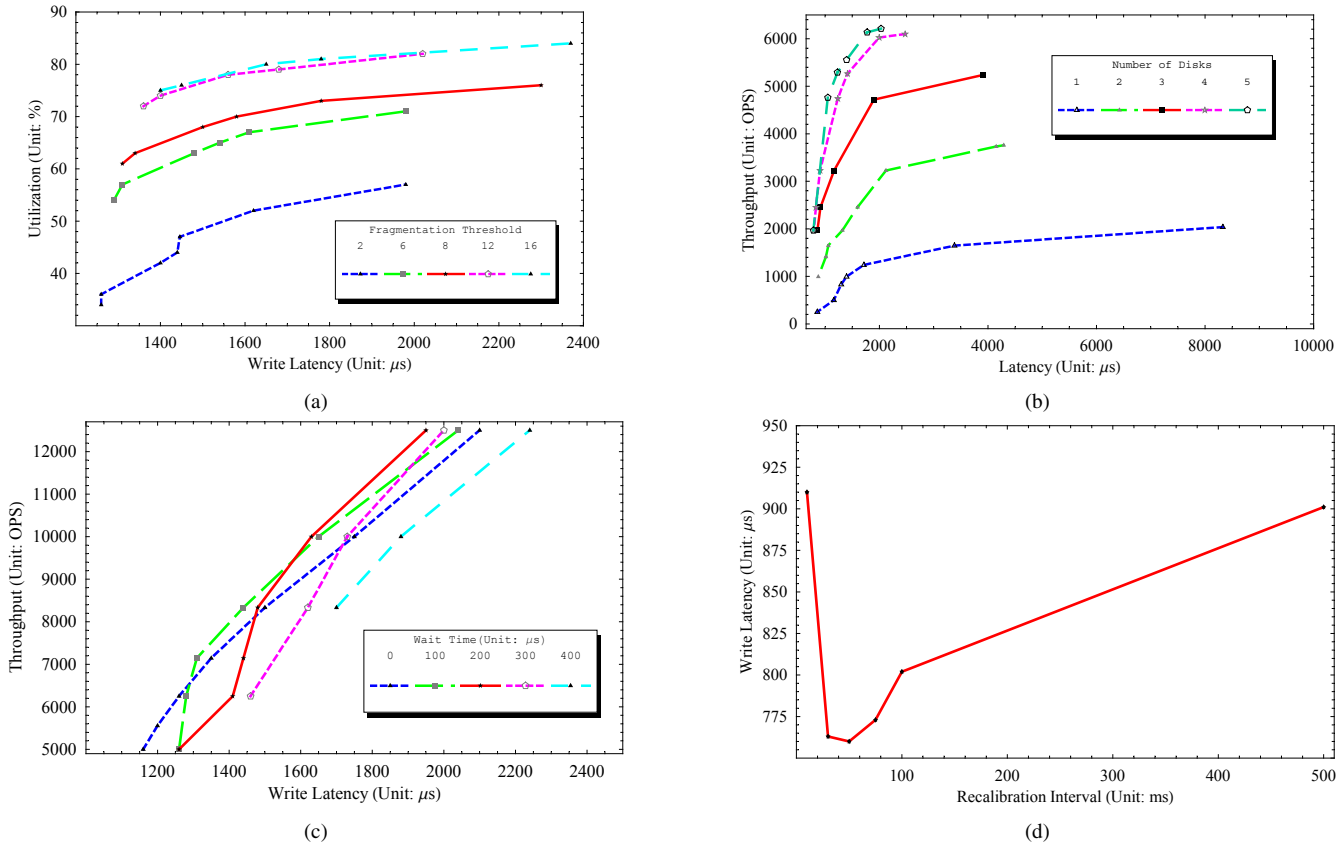

Figure 6. *(a): Performance impact of the choice of T_{switch} on the log disks' disk space utilization efficiency and write latency. (b): Performance impact of the number of log disks on the log disks' throughput and write latency.(c): Performance impact of wait time limt (T_{wait}) on the log disks' throughput and write latency. (d): Impact of the choice of the recalibration frequency or interval on the log disks' write latency.*

and may actually delay the disk writes requests from users. Therefore, for a given workload, there is an optimal recalibration frequency that balances these two performance factors, as shown in Figure 6(d). For a workload consisting of 4 KB large requests, a recalibration interval of 50msec is the optimal value.

7.4. Impact of iSCSI Processing

This subsection evaluates the write throughput and latency of *Mariner*'s Trail node as seen from an iSCSI client. The test environment contains one iSCSI connection from an iSCSI client to an iSCSI target that uses *Mariner*'s Trail node as the underlying storage device. We use a synthetic workload that keeps sending write requests of size 4KB at a fixed inter-request interval and for each run, we measure the average write latency. The queue length of both the iSCSI initiator and the iSCSI target is set to 2048 to accommodate large bursts.

We modify the *IET* iSCSI implementation in the following ways to improve its write latency. The first modification is to avoid going through the file-system-related APIs as used by the `fileio` mode of the *IET* implementation. Instead, we call *generic_make_request* directly, a standard interface between the block device and other components of the kernel. The second modification is to simplify the software architecture. The original *IET* iSCSI implementation has two categories of threads: a network thread and a pool of worker threads to issue requests to the underlying storage entities. These two threads relay data through an iSCSI command queue. Our implementation eliminates the iSCSI command queue and directly places write requests into the per-log-disk request queue.

Figure 7 shows how the iSCSI-level write latency varies with the inter-request interval. The iSCSI-level write latency increases dramatically when the inter-request interval falls below 0.18msec, because the input load corresponding to the inter-request interval of 0.18msec hits the capacity of the log disks. The IET iSCSI target implementation could process an iSCSI command every 0.08msec. The average batch size is around 8KB, which takes a fixed processing overhead of 0.1msec. The Trail implementa-

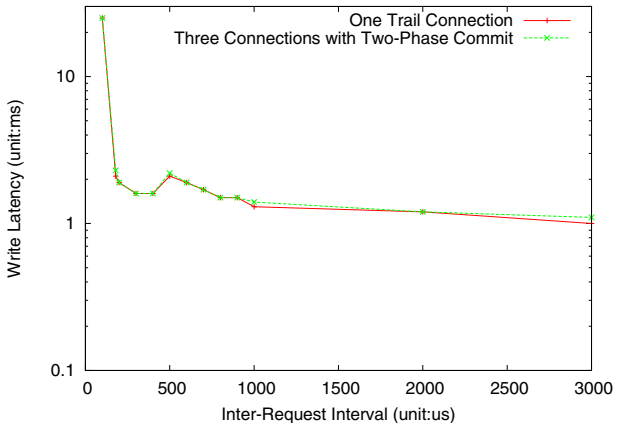

Figure 7. *The impact of the inter-request interval of the input write request sequence on the average write latency. Both the iSCSI initiator and target set their queue length to 2048. T_{wait} is set to 0.36 msec and T_{switch} is set to 2. One vanilla iSCSI connection with only Trail node is compared with two-phase commit implementation.*

tion in the *Mariner* prototype introduces a small overhead (around 0.07msec), which comes from decision logic that determines which log disk to use, and post-processing after each physical I/O completion.

Figure 7 also shows that the iSCSI-level write latency increases as the inter-request interval increases from 0.3msec to 0.5msec. This is because a request is delivered to the disk controller under two scenarios: either the request's wait time exceeds T_{wait} when the next request comes in or there is no queued request and a free log disk is ready to be used. These two conditions conflict with each other: a wait time exceeding T_{wait} indicates there have been queued requests and future requests will get queued. Therefore, a time point exists to reach the worst case: the time just falls within T_{wait} and forces subsequent requests to believe there have been queued requests and experience the queuing delay in the same way. For T_{wait} of 0.36 msec and 5 log disks, this time point happens to be 0.5 msec. After reaching a peak value at 0.5 msec, the write latency drops down as the inter-request interval increases because input request rate is far below *Mariner*'s capacity and no request needs to be queued.

7.5. Impact of Modified Two-Phase Commit Protocol

In this section, we study the performance impact of the modified two-phase commit protocol on the write latency. An iSCSI client is connected to a Trail node, a master node and a local mirror node. The queue length of both the iSCSI initiator and the iSCSI target is set to 2048 to accommodate large bursts. We use both synthetic workload and real disk access traces in this experiment. The synthetic workload consists of multiple request bursts with a sufficiently long time interval between two consecutive bursts to allow the master and local mirror node to finish the previous burst. The target block address of each disk write request in the synthetic trace is borrowed from the Lair62b trace. Both the master node and local mirror node could complete a burst size of 1000 requests within 100msec when the on-disk cache is turned on. The burst size in the synthetic workload is set to 1000 and the interval between two consecutive bursts is 200msec.

Figure 8(a) shows the write latency versus the inter-request interval within a write burst. Compared the one connection case with that of modified two-phase commit, it is clear that the modified two-phase commit protocol does not introduce any noticeable penalty on the write latency. There are two reasons. First, because of TRM, the additional payload transfer due to data replication does not incur additional networking overhead. Second, the latency of a modified two-phase commit transaction ends when the write to the Trail node is completed, which is exactly the same as the iSCSI-level write latency reported in the previous subsection. Figure 8(b) shows the measured write latency under disk access traces played back at different speedup factors. Because the MS-SQL-Large trace is collected on a server with a large buffer cache, it contains larger bursts and the average inter-request interval is small.

When the speedup factor is smaller than 1, the write latency increases because the inter-request interval increases and the additional batching delay of the current *Mariner* prototype kicks in and when the requests are sparse, the additional batching delay disappears. When the speedup factor is large, the write latency also increases because of the larger input load. For the MS-SQL-Large trace, increase in the speedup factor eventually exceeds the throughput capacity of the log disks and result in very long write latency, most of which is queuing delays at the *Mariner* client and Trail node.

To illustrate the performance improvement, we setup a vanilla storage server where writes are only propagated to one storage node consisting of a vanilla hard drive. The vanilla hard drive has their on-disk cache turned off. Storage client and server are attached locally. We use MS-SQL-Large and MS-SQL-Small trace to drive the comparison. Figure 8(b) shows the performance improvement.For all speedup factors, our scheme beats the vanilla configuration by at least a factor of 500.

Figure 9(a) illustrates that both the write and read throughput are improved by a factor of 5. This is because the throughput of Postmark is sensitive to per-request latency especially when writes are synchronous. This is because Postmark is a single-threaded benchmark and each

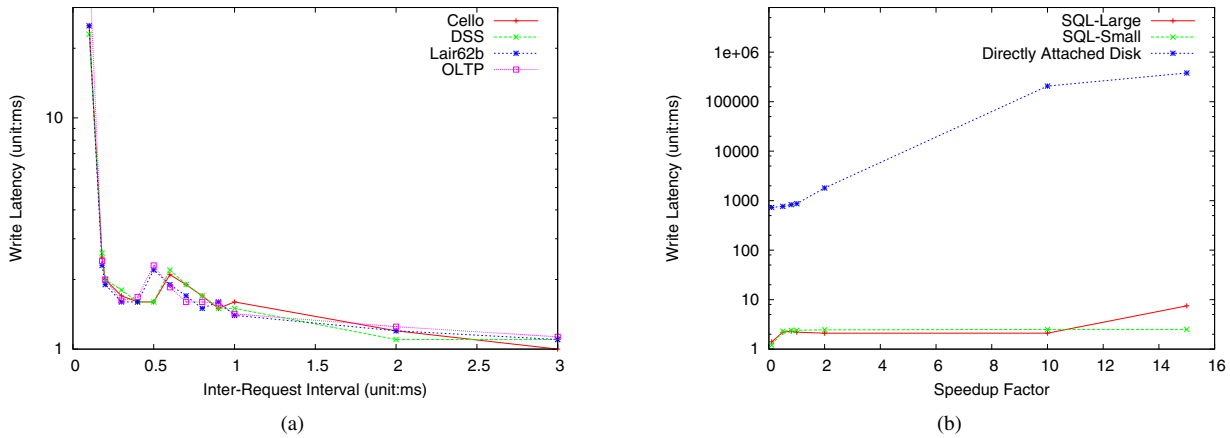

Figure 8. *(a): The measured end-to-end latency from a* Mariner *client under four different types of workload. The burst size is set to 1000 and inter-burst interval is set to 1 second. (b): The measured end-to-end latency from a* Mariner *client under two real traces at different speedup factors. Y axis is in log scale. Both iSCSI initiator and target set their queue length to 2048.* T_{wait} *is set to 0.36msec and* T_{switch} *is 2.*

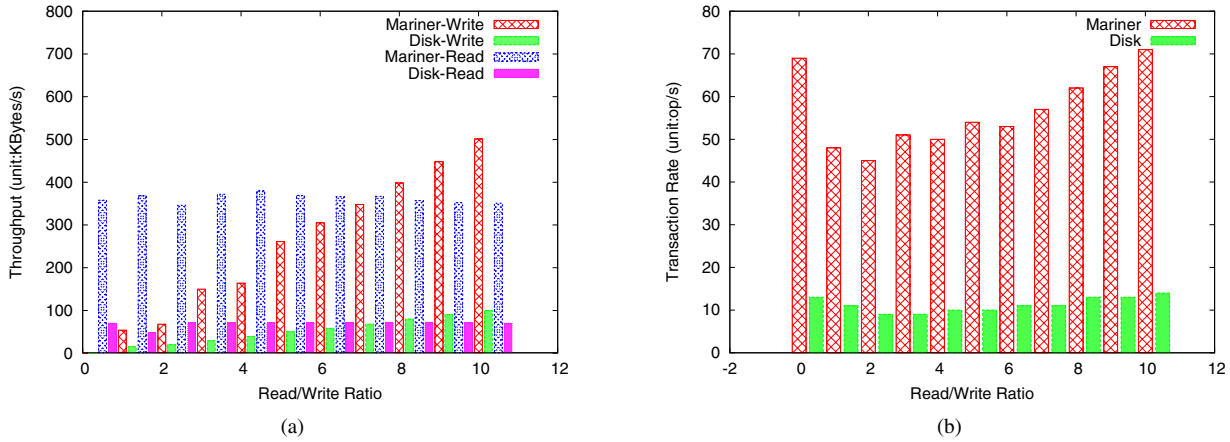

Figure 9. *(a): The measured Postmark throughput for directly-attached disk and* Mariner*'s storage architecture, respectively with different read/write ratio. We create 100,000 files and run 500,000 transactions. Both the iSCSI device and the locally-attached disk are synchronously mounted to the working directory of Postmark. (b): The measured Postmark transaction rates for directly-attached disk and* Mariner*'s storage architecture with different read/write ratio. For* Mariner *, both iSCSI initiator and target set their queue length to 2048.* T_{wait} *is set to 0.36msec and* T_{switch} *is 2. For directly-attached hard drive, we turn off their on-disk write cache.*

synchronous operation will prevent future requests from being sent out. As a result, a reduction in the per-request elapsed time by N will lead to a N times increase in the throughput. The average per-write elapsed time is 3 msec for *Mariner* client above the file system and 20 msec for directly-attached disks.

Figure 9(b) shows the transaction rate is also improved by a factor of 5. As all writes are synchronous, a throughput improvement of 5 indicates a per-request latency improvement of 5. This is backed by the average per-write latencies of both directly-attached disks and *Mariner*'s storage system. The per-request latency on directly-attached disk is around 20 msec and the per-request latency on *Mariner*'s storage system is 3.2 msec when read/write ratio is zero. However, under this workload, the advantage of request batching can not be shown very clearly as at any point in time, there is only one outstanding write request and there is no batching.

Table 2 shows the per-request latency improvement for different NFS operations by playing the Lair trace with TBBT trace player. TBBT trace player is at the NFS client side. Playing one-hour trace of the whole day trace(2:00am on Oct 21st, 2001) in full speed takes only 3 seconds for *Mariner* and 21 seconds for direct-attached disk. Per-write

NFS OP	Avg Elapsed Time for Disk (msec)	Avg Elapsed Time for *Mariner* (msec)
setattr	33.3	4.8
write	121	12
create	82.7	9
remove	64.5	7
rename	57.3	9.0
link	83.2	8.9
mkdir	161	13.4

Table 2. *Elapsed time improvement for different write-related NFS operations, where T_{switch} = 2 and T_{wait}= 0.36 msec. Both direct-attached disk and* Mariner*'s iSCSI device are mounted synchronously on the NFS server directory.*

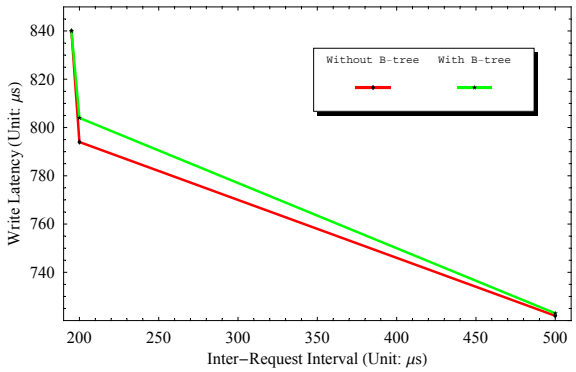

Figure 10. *The overhead associated with metadata insertion and update*

NFS operation latency improves by a factor of between 6 and 10. This is because there are multiple outstanding requests for batching. The latency at the iSCSI layer is 2 msec and .

7.6. Metadata Update Overhead

For each logical disk write, *Mariner* creates a log record, writes it to a log disk together with the payload and updates the B-tree index structure asynchronously. More specifically, the user-level B-tree daemon periodically polls the kernel for updates to the B-tree, and makes the corresponding modifications in one batch. The performance overhead of index structure updates mainly comes from context switching and additional disk I/O to force the updates to disk. To measure the performance impact of updates to the index structure, we compare the average write latencies when the index structure update is turned on and off.

In this experiment, we set up an iSCSI initiator and an iSCSI target, which uses a Trail device as the underlying storage device, and issue a total of 200000 4KB synthetic write requests from the iSCSI initiator. Because a Trail device writes every incoming write request where the disk head happens to be at that instant, the locality characteristic of the input write requests is immaterial. Therefore, it is perfectly reasonable to use a synthetic workload in this study. We measure a Trail device's throughput by gradually decreasing the inter-request interval of the input workload until the measured average write latency starts to increase dramatically, at which time the input load exceeds the system's capacity and the queuing delay starts to dominate.

The difference between the two curves in Figure 10 represents the performance overhead due to index structure updates. This overhead is less than 1% when the user-level B-tree daemon polls the new B-tree entries in the kernel every 200msec, which is translated to five context switches per second. The resulting context switching overhead is negligible. Because the user-level B-tree daemon writes updates to disk in batches, the disk I/O cost is also insignificant.

8. Conclusion

Modern enterprise storage systems are increasingly geared towards the notion of *comprehensive data protection*, which aims to protect data from hardware/software failures, human errors, malicious attacks and environmental disasters. To achieve comprehensive data protection, existing storage systems/products tend to glue together a variety of data protection mechanisms that were developed separately in an ad hoc way. The result is that comprehensive data protection comes with excessive performance overhead. The goal of the *Mariner* project is to develop efficient implementation techniques that could reduce the performance penalty associated with comprehensive data protection to a negligible level. Along the way, we recognize that replication and logging are the two fundamental building blocks for comprehensive data protection, and organize *Mariner*'s system architecture around these two primitives to minimize the associated performance overhead. More specifically, the *Mariner* project makes the following research contributions:

- A modified track-based logging technique that can simultaneously achieve low write latency, high write throughput and high disk space utilization efficiency using only commodity IDE/ATA drives,
- A modified two-phase commit protocol that exploits low-latency disk logging to hide the latency of local mirroring and remote replication without compromising the data integrity, and
- A novel transparent reliable multicast mechanism that exploits Ethernet switches' VLAN support to implement tree-based link-layer multicast and drastically

reduces the storage-area networking overhead associated with data replication.

In the end, for a five-disk configuration, *Mariner* is able to deliver 1.8msec write latency and achieve 70% log disk space utilization under an input workload of 12500 writes/sec and 4KB per write request. With this performance result, we believe we have proved our thesis that it is possible to support comprehensive data protection without incurring significant performance overhead.

Although the run-time performance overhead of the *Mariner* architecture is similar to that of vanilla iSCSI storage systems without any protection, it requires much more sophisticated recovery processing for various failure modes. How to produce a provably correct implementation of these data recovery schemes is our current focus. In addition, how to transform *Mariner*'s block-level CDP functionality into something similar to those provided by standard versioning file systems is another research direction that we are currently pursuing.

References

[1] Tzi cker Chiueh and Lan Huang, "Track-Based Disk Logging," in *DSN '02: Proceedings of the 2002 International Conference on Dependable Systems and Networks*, Washington, DC, USA, 2002, pp. 429–438, IEEE Computer Society.

[2] Cisco Corp., "A Closer Look at SANTap An Open Protocol for Appliance Based Storage Applications," http://www.snseurope.com/cisco_supplement/5.pdf.

[3] Andrew Warfield, Russ Ross, Keir Fraser, Christian Limpach, and H. Steven, "Parallax: Managing Storage for a Million Machines," in *HotOS X, Tenth Workshop on Hot Topics in Operating Systems*, pp. 1–11.

[4] Eno Thereska, Jiri Schindler, Christopher R. Lumb, John Bucy, Brandon Salmon, and Gregory R. Ganger, "Design and Implementation of a Freeblock Subsystem," Carnegie Mellon University Parallel Data Lab Technical Report CMU-PDL-03-107, December, 2003.

[5] Eno Thereska, Jiri Schindler, John Bucy, Brandon Salmon, Christopher R. Lumb, and Gregory R. Ganger, "Awarded Best Student Paper! – A Framework for Building Unobtrusive Disk Maintenance Applications," in *FAST '04: Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, Berkeley, CA, USA, 2004, pp. 213–226, USENIX Association.

[6] J. Saran et al., "Internet Small Computer Systems Interface (iSCSI)," Tech. Rep. RFC3720, Internet Engineering Task Force (IETF), April 2003.

[7] Thorsten von Eicken, Anindya Basu, Vineet Buch, and Werner Vogels, "U-Net: a user-level network interface for parallel and distributed computing," in *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP)*, 1995, vol. 29, pp. 303–316.

[8] Gregory D. Buzzard, David Jacobson, Milon Mackey, Scott Marovich, and John Wilkes, "An Implementation of the Hamlyn Sender-Managed Interface Architecture," in *Operating Systems Design and Implementation*, 1996, pp. 245–259.

[9] M. A. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. W. Felten, and J. Sandberg, "Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer," in *Proc. of the 21th Annual Int'l Symp. on Computer Architecture (ISCA'94)*, 1994, pp. 142–153.

[10] Jiuxing Liu, Jiesheng Wu, and Dhabaleswar K. Panda, "High Performance RDMA-based MPI Implementation over InfiniBand," *Int. J. Parallel Program.*, vol. 32, no. 3, pp. 167–198, 2004.

[11] *InfiniBand Architecture Specification, Volumes 1 and 2, Release 1.0.a.* , http://www.infinibandta.org/.

[12] Ken Yocum and Jeffrey S. Chase, "Payload Caching: High-Speed Data Forwarding for Network Intermediaries," in *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*, Berkeley, CA, USA, 2001, pp. 305–317, USENIX Association.

[13] Gang Peng, Srikant Sharma, and Tzi cker Chiueh, "A Case for Network-Centric Buffer Cache Organization," in *Hot Interconnect 2003*, Aug 2003.

[14] Cisco Corp., "Cisco MDS 9000 Family SANTap Service-enabling Intelligent Fabric Applications," http://www.cisco.com/.

[15] S. Sharma, K. Gopalan, S. Nanda, and T. Chiueh, "Viking: A Multi-SpanningTree Ethernet Architecture for Metropolitan Area and Cluster Networks," in *INFOCOM 2004*, 2004.

[16] W. Fenner et al., "Internet Group Management Protocol, Version 2," Tech. Rep. RFC2236, IETF, Nov 1997.

[17] IEEE, "IEEE Standard for Local and Metropolitan Area Networks: Virtual Bridged Local Area Networks," Institute of Electrical and Electronics Engineers, 1998.

[18] IEEE, "IEEE Standard for Local and Metropolitan Area Networks: Supplement to Media Access Control (MAC) Bridges: Traffic Class Expediting and Multicast Filtering," Institute of Electrical and Electronics Engineers, 1998.

[19] IEEE, "IEEE Standard for Local and Metropolitan Area Networks: Multiple Spanning Trees," Institute of Electrical and Electronics Engineers, 2002.

[20] IEEE, "IEEE Standard for Local and Metropolitan Area Networks: Rapid Configuration of Spanning Tree," Institute of Electrical and Electronics Engineers, 2000.

[21] IEEE, "IEEE Standard for Local and Metropolitan Area Networks: Media Access Control (MAC) Bridges," Institute of Electrical and Electronics Engineers, 1990.

[22] Daniel Ellard, Jonathan Ledlie, Pia Malkani, and Margo Seltzer, "Passive NFS Tracing of Email and Research Workloads," in *FAST '03: Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, Berkeley, CA, USA, 2003, pp. 203–216, USENIX Association.

[23] Zhifeng Chen, Yan Zhang, Yuanyuan Zhou, Heidi Scott, and Berni Schiefer, "Empirical evaluation of multi-level buffer cache collaboration for storage systems," in *SIGMETRICS '05: Proceedings of the 2005 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, New York, NY, USA, 2005, pp. 145–156, ACM Press.

[24] J. Katcher, "PostMark: A new file system benchmark," in *Tech. rep. TR-3022*, Sunnyvale, CA, 1997, Network Appliance Inc.

[25] Ningning Zhu, Jiawu Chen, Tzi cker Chiueh, and Daniel Ellard, "TBBT: scalable and accurate trace replay for file server evaluation," *SIGMETRICS Perform. Eval. Rev.*, vol. 33, no. 1, pp. 392–393, 2005.

[26] University of New hampshire InterOperability Laboratory, "iSCSI Initiator and Target Reference Implementations for Linux," http://sourceforge.net/projects/unh-iscsi.

[27] Yamini Shastry, Steve Klotz, and Robert D. Russell, "Evaluating the Effect of iSCSI Protocol Parameters on Performance," in *Parallel and Distributed Computing and Networks*, 2005, pp. 159–166.

0-7695-3025-7/07 $25.00 © 2007 IEEE