

An Efficient Data Sharing Scheme for iSCSI-Based File Systems

Dingshan He David H.C. Du

February 16, 2004

Abstract

iSCSI is an emerging transportation protocol for transmitting storage I/O commands and data blocks over TCP/IP networks. It combines two popular technologies, SCSI and TCP/IP. iSCSI takes advantage of ubiquitous TCP/IP infrastructures, such that it is possible for application servers to access remote storage devices sitting anywhere in the Internet. This allows iSCSI to provide supports for many mission-critical applications, such as remote data mirroring and remote data backup. Moreover, breaking of direct binding between application servers and storage devices enables device and data sharing among multiple application servers, since these servers are able to concurrently access the same storage devices. Although it has been widely accepted that iSCSI is a promising storage-over-IP technology, some issues for environments based on this technology have not been addressed yet. A fundamental one is how to enable consistent and efficient data sharing, as data sharing is a major advantage of iSCSI-based storage subsystems. With multiple application servers performing read/write operations on shared data concurrently, a concurrency control scheme is necessary to maintain data consistency. Furthermore, application servers may cache a portion of shared data to improve performance, which becomes almost indispensable as iSCSI-based storage subsystems will be attached to WANs directly. Therefore, a cache consistency control mechanism is required to make cache functioning correctly. In this paper, we propose a data sharing scheme for iSCSI-based file systems, and use ext2 as an example for implementation. This proposed scheme utilizes physical block caching at application servers to achieve acceptable performance over long distance networks. A cache consistency control mechanism based on callback is used to keep the content of caches up to date. To insure shared data on iSCSI-based storage subsystems to be consistent, our proposed scheme employs locking mechanisms to prevent shared data from getting corrupted. When designing the locking mechanisms, we treat metadata and normal file data differently, based on the observation of different access patterns for them. For normal file data, we propose a hierarchical locking scheme, which is capable of balancing between the level of access concurrency and the costs for locks, including both the network communication cost and the memory space needed to maintain locks. In addition, we incorporate transaction file sharing semantics into our scheme. This could assure a sequence of operations to be executed consistently, similar to database transactions. Finally, we use simulation tools to verify the correctness of our design, and study its performance.

1 Introduction

iSCSI [1, 2, 3] combines two successful protocols in the data storage area and the network communication area - SCSI and TCP/IP. Small Computer System Interface (SCSI) enables host computer systems to perform I/O operations of data blocks with peripheral devices. iSCSI extends the connections in SCSI from traditional parallel cables, which could only stretch to several meters, to ubiquitous TCP/IP networks. iSCSI handles the unreliable environment of IP networks, since SCSI requires stable and responsive communication channels. iSCSI encapsulates and reliably delivers SCSI Protocol Data Units (PDUs) over TCP/IP networks.

iSCSI is expected to expand developments in the System Area Network (SAN) market. Currently, Fiber Channel (FC) [4] is playing a major role in this field. FC has dedicated fiber connections between storage subsystems and processors, and uses Fiber Channel Protocol (FCP) to carry SCSI PDUs over fiber networks. Thus, FC offers low-latency and high-bandwidth data channels. Unfortunately, FC still has certain limitations. First, FC introduces another technology, which is completely different from prevail IP-based technologies, into corporate network infrastructures. The cost of this technology is still prohibitively high due to the relatively small volume of production, as well as expenses on separate management for FC, in addition to TCP/IP. Secondly, FC is still limited by connect distance up to 10 kilometers, due to the fact

that FCP is a non-routable protocol. Although we have seen efforts, such as iFC and FCIP, to connect isolated FC islands through TCP/IP networks, the complexity seems to be a major obstacle, since they involve mappings from SCSI to FC, and from FC to TCP/IP. Therefore, FC is not suitable for storage systems which need to span over long distances.

iSCSI could complement FC, since it is running on ubiquitous and mature TCP/IP networks. There is no limit on the distance of physical connections, as ip networks could be connected by routers to extend to anywhere. The costs for hardware, softwares, and managements are low due to the large scale of existing deployments of TCP/IP infrastructures.

One major feature of SAN is to provide a shared pool of storage resources for multiple accessing hosts. Therefore, storage sharing is natively supported by SANs, including iSCSI-based ones. However, data sharing is going beyond storage sharing. Storage sharing allows storage devices to be physically shared by multiple clients. In this case, it is not necessary to support sharing of data stored on shared storage devices. Such capability is natively supported by the SAN architecture, since storage devices are not directly attached to any hosts. The typical scenario is that the physical space of shared storage devices are partitioned, and each partition is used by one client exclusively. However, a native SAN system has not fully achieved data sharing mentioned above. With data sharing, a single piece of data could actually be shared by multiple clients, and the integrity of data content has to be enforced. Therefore, concurrency control mechanism is necessary when multiple hosts could read/write a single piece of data simultaneously. The IBM Distributed Lock Manager (DLM) and the DLM in the Lustre project are examples to address this issue in cluster environments. Both of them are after the DLM in VAX clusters.

When using iSCSI-based storage subsystems, the systems are not restricted to cluster environments or LANs. Application servers and their shared storage subsystems could be connected by TCP/IP networks, over long distances. Applications, such as file systems, on the application servers are not aware of the fact that iSCSI-based storage subsystems are accesses through iSCSI, and these storage subsystems are potentially shared with other application servers. Traditionally, data blocks are organized into files managed by file systems. In environments such as Network Attached Storage (NAS) and Network File System (NFS) [5], data are accessed based on files. File systems on NAS devices or NFS servers will monitor file accesses, since all data access requests have to go through their file systems. Therefore, the data sharing functionality is naturally carried out by file systems sitting on NAS devices or NFS servers. However, in iSCSI-based architectures, there is no file system module running on iSCSI targets. File systems on application servers are inadequate, since they are only aware of requests going through themselves. Application servers need to be coordinated when accessing shared data concurrently. In this paper, we are going to integrate several new modules into the existing iSCSI-base file systems, and these new modules will finally achieve the real data sharing.

Since iSCSI-based architectures could be deployed over WANs, latency introduced by large physical distance will be a severe restriction on performance. To improve performance, caching at application servers is necessary. In addition to hiding physical latency, caching at application servers can also reduce load at iSCSI targets, which are busy serving requests from many iSCSI initiators. However, caching mechanisms introduces the problem of cache consistency, which is the consistency between cached data on clients (application servers) and data on servers (iSCSI-based storage subsystems). In our design, a callback-based cache consistency control mechanism is used.

Metadata are also concurrently accessed, and cached by multiple clients. However, a general solution for both metadata and normal file data would be inefficient, since metadata and normal file data are difference in terms of access pattern. Our work takes this into consideration and designs different mechanisms for them separately. We use a semi-preemptible-shared (SPS) locking mechanism for metadata, and a two-tier hierarchical (TTH) locking mechanism for normal file data.

Finally, most existing SAN file systems, such as Lustre [6], only provide UNIX file sharing semantics. Under such file sharing semantics, writes are mutually atomic, and the result of a write is immediately visible to all readers. However, the UNIX file sharing semantics is not suitable for transactional execution of a sequence of operations. In the transaction file sharing semantics, there should be a consistent view of any involved data throughout the execution of a transaction. Deadlocks are possible, so detection and resolution of deadlocks are considered. In addition, rollback capability is required in case a transaction is unable to complete due to deadlocks. One focus of our design is to provide above supports for the transaction file sharing semantics. In order to reduce occurrences of deadlocks, we design a file-level admission control

scheme to limit the number of concurrent accesses on files. This file-level admission control scheme is based on the TTH locking scheme. Details of this scheme is presented in section 4.4.

Our contributions are followings. First, we have designed a data sharing scheme, including concurrency control and caching consistency mechanisms, for iSCSI-based file systems. Secondly, we have identified and compared the distinction between metadata and normal data. Based on these, we have designed different locking mechanisms for metadata and normal data to fully exploit the concurrency and efficiency. Thirdly, we have designed our system to be able to support transaction file sharing semantics. This semantics supports applications that require to perform a sequence of operations in a transaction way. Finally, we have proposed a file-level admission control mechanism based on the TTH locking mechanism. Our simulation shows that this mechanism is effective in reducing the occurrence of deadlocks, such that it reduces the waste on rollback and replay of transactions.

The rest of this paper is organized as following. In section 2, we will introduce the iSCSI protocol and the unix file system as the background of our design. In section 3, we will discuss data sharing in iSCSI-based file systems, and show why current implementations are insufficient. In section 4, we are going to give an overview of our design. The concurrency control schemes for metadata and normal data will be discussed in detail. In addition, our cache consistency control mechanism is also presented. Our detailed design are discussed in section 5. We will discuss those new modules that we have inserted to iSCSI initiators and iSCSI targets. In section 6, we present the simulation results over the ns-2 to evaluation the performance of our design. Finally, in section 7, we summarize the related works.

2 Background

In this section, we are going to introduce the iSCSI protocol and the unix file system as backgrounds. Although readers can find discussions about unix file systems in any modern operating system textbook, we still describe it here for completeness.

2.1 iSCSI protocol

iSCSI integrates Small Computer System Interface (SCSI) over TCP/IP. SCSI enables host computer systems to perform input/output (I/O) operations of data blocks with peripheral devices [3]. The architecture of SCSI is based on a client/server model. Clients typically are host systems that issue requests to read or write data blocks. Servers could be any peripheral devices that respond to client requests. In the SCSI terminology, a client is called an *initiator* that actively issues SCSI commands, and a server is called a *target* that passively executes commands and responses with results. A target has one or more than one processing units called *logical units* (LUs). Each logical unit is assigned a *logical unit number* (LUN) that uniquely identifies it within a target. An initiator issues a command by constructing a data structure called *command descriptor blocks* (CDB), which is sent to a target for a specific LU, which is denoted as target/LUN. The CDB could contain request for either reading or writing a number of data blocks. Upon receiving the CDB, the specified LU within the target will start either sending or receiving data. At the end, a response message will always be sent from the target to the initiator to indicate the status of the request.

iSCSI extends the connections between initiators and targets from traditional parallel cables to TCP/IP networks. Traditional SCSI parallel cabling has limitations on the distance and the number of devices supported. Such limits motivate the development of new connections based on network technologies, such as Fiber Channel and Gigabit Ethernet. iSCSI deals with the unreliable environment of IP networks, since the SCSI protocol requires stable and responsive communication. As shown in the figure 1, iSCSI encapsulates and reliably delivers CDBs and data blocks over TCP/IP networks. An iSCSI session is set up between an initiator and a target through login process. During the login phase, variable parameters are negotiated between an iSCSI initiator and an iSCSI target. In addition, a security routine may be invoked to perform authentication for the connection request, since iSCSI is trying to accommodate un-trusted IP environment. Once the login phase completes successfully, the iSCSI session leaves logic phase and enters full feature phase for normal SCSI transactions. Multiple TCP connections could be set up within a session for concurrent unrelated SCSI transactions. However, individual command/response pair must flow over the same TCP connection, which is known as *connection allegiance*.

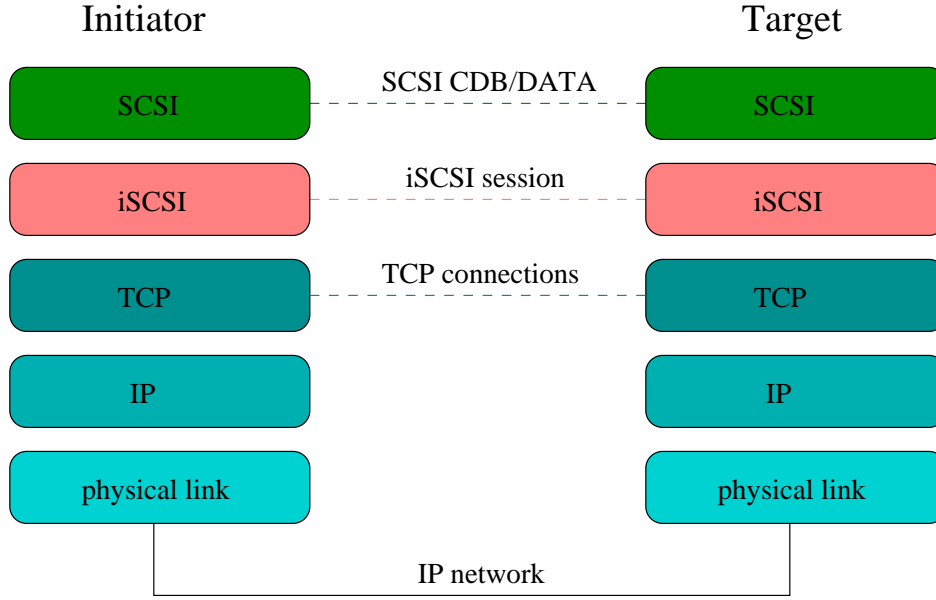


Figure 1: iSCSI protocol stack

As we have mentioned, SCSI assumes an error-free inter-connection. Such assumption is not guaranteed in IP networks. To handle errors and recover from errors, both iSCSI initiators and iSCSI targets buffer commands and responses, until they are acknowledged. Each PDU has a sequence number within its session. Any missing of PDU will be recovered through sequence number acknowledgement, which requests the initiators to retransmit missing PDUs. Corrupted PDU will be detected by iSCSI targets, and triggers a Reject iSCSI PDU in response, which in turn triggers the recovery for the corrupted PDU.

As shown in figure 2, a typical scenario includes several file servers or database servers as iSCSI initiators, and disk arrays or tape subsystems as iSCSI targets. The iSCSI initiators and the iSCSI targets may or may not be sitting in the same Gigabit Ethernet. Multiple application servers could share the same data on some iSCSI target/LUN. Every initiator and every target are equipped with network interface and iSCSI protocol stacks, and all data are serviced through iSCSI over IP networks. A detailed discussion of iSCSI-based file systems will be given in section 3.

2.2 Unix file system

Our design is based on Unix-like file system, but it is easy to extend our design to any file systems. We consider each LU as an independent storage device, and each LU has its own file system starting from a single root directory. A LU is partitioned into fix-sized physical blocks. Block numbers are assigned to these physical blocks, starting from 0. According to the convention of the Unix file system, block 0 is the boot block, which contains booting code. Block 1 is the super-block, which contains basic information of the file system on this device, such as the size of a physical block, the total number of physical blocks, etc. Following the super-block, there are several blocks called inode-bitmap blocks. Their functionality is to manage free inodes. Each bit represents an inode. When an inode is in use, the corresponding bit in the indoe-bitmap is set to 1; otherwise, the bit should be 0. Following the inode-bitmap blocks, there are several data-block-bitmap blocks. They are similar to the inode-bitmap blocks, except that their functionality is to manage free data blocks. The left physical blocks are divided into two sections. The first section is the inode-block section, and the other one is the data-block section. Figure 3 shows such layout on the physical device.

In Unix files systems, directories and devices are all considered as files. Each file is identified by an inode, which has a unique inode number within its file system. The first inode in the first inode block is the inode 1, which is usually the root directory of its file system. The other inodes are numbered accordingly. In the iSCSI environment, our major interest is data accessing, so we ignore the devices files. Then, we have two types of files – directory files and normal files. An inode has a field to indicate the type of the file identified

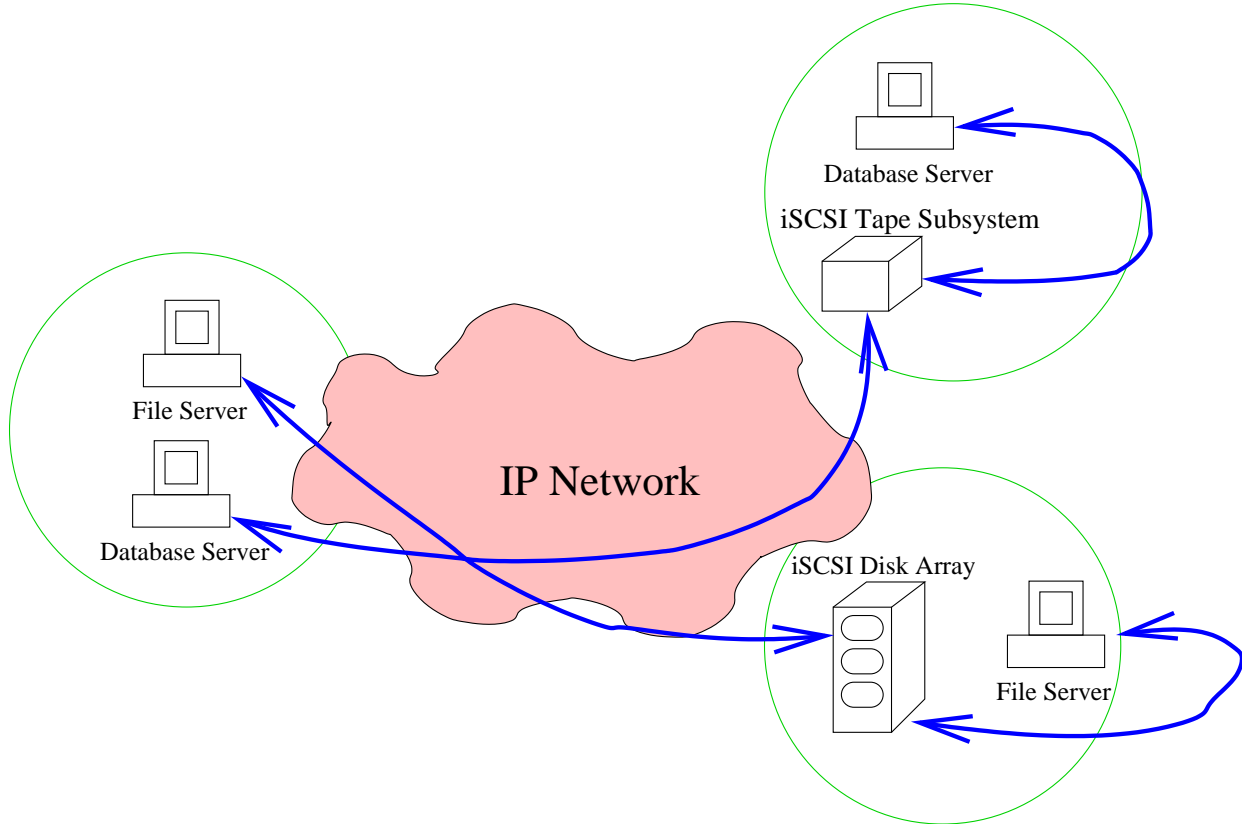


Figure 2: iSCSI network architecture scenario

by it. An inode only carries information for management purpose, such as the type of the file, the size of the file, the time information of the file, etc. The data of a file are stored on several data blocks, whose block numbers are stored in the inode of the file. For a directory file, its data blocks are called directory data blocks. A directory data block contains several directory entries. Each directory entry has two fields – the name and the inode number. In figure 3, inode 1 is the root directory, and the first data block is a directory block. In the example, this directory block contains directory entries, such as foo1, foo2, foo3, and so on. For a normal file, its data blocks are called normal data blocks, which contain the really data of the file. In the figure 3, directory entry foo2 is referring to inode 3. This is a normal file, so the inode 3 is pointing to several data blocks, which contain the real data of file foo2. Since the inode has fixed size, indirect, double indirect, or even triple indirect blocks are used to support large-sized files.

3 Overview of Data Sharing in iSCSI-based File Systems

As we have shown in figure 2, file systems are potential system applications of iSCSI-based storage subsystems. In this section, we are going to discuss the environment of iSCSI-based file systems in detail. Then we will show why current implementations of iSCSI-based file systems are insufficient to achieve consistent data sharing. Although we only focus our discussion on iSCSI-based file systems, the same issues will be encounter in other iSCSI-based applications, such as iSCSI-based database systems. In the next section, we will present our design based on current implementations to address the data sharing issue in iSCSI-based file systems.

In current implementations, an iSCSI driver or iSCSI network interface card is installed in the IP hosts, which function as iSCSI initiators. IP hosts contact a iSCSI target to perform iSCSI login via a defined iSCSI target address and port. If an iSCSI login is successful, the operation enters full feature phase and data transport is allowed. Within an IP host, after a successful login phase, the available iSCSI targets/LUNs

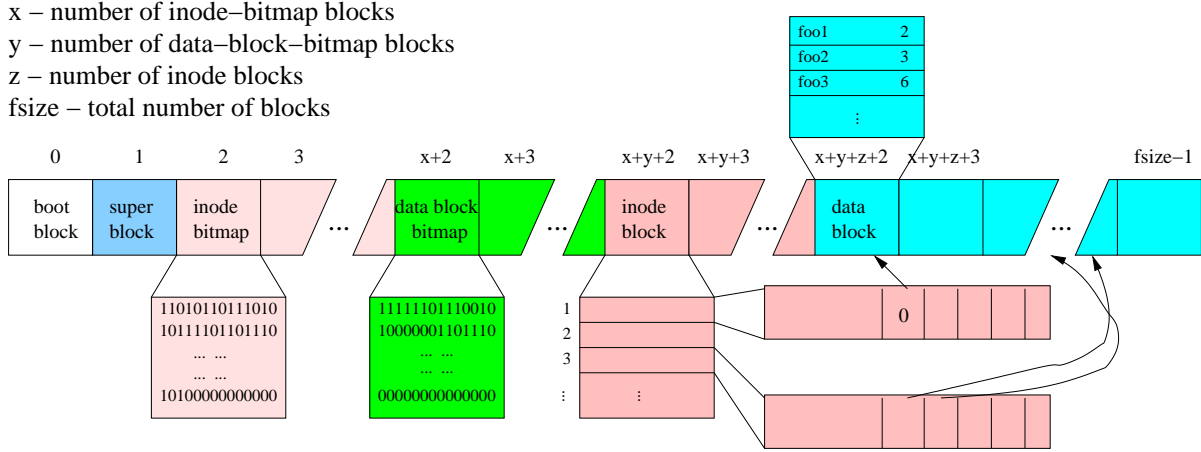


Figure 3: The unix file system layout

are mapped as local SCSI devices by the iSCSI driver. For example, under linux, these iSCSI targets/LUNs could be mapped to local SCSI devices `/dev/sda`, `/dev/sda1`, and so on. In order to access the file system on one iSCSI target/LUN, the SCSI device corresponding to this iSCSI target/LUN is mounted onto a mounting point within the IP host's local file system. The file system on a iSCSI target/LUN could be any file systems, as long as the kernel of the mounting IP host has corresponding file system modules, in case of linux kernel, to handle them. These file systems are not modified for iSCSI at all. In this kind of implementation, IP hosts are independent of each other, in that a specific IP host is unaware of the existence of any other IP hosts.

Although the mounted file system modules in the IP hosts are able to use existing file system locks locally, multiple IP hosts still can not guarantee consistent data sharing. Local file system locks can assure that all processes running on a specific IP host have mutual exclusive accesses to the mounted file system on some iSCSI target/LUN. However, if an IP host does not cooperate with other IP hosts, who are accessing the same file systems on certain shared iSCSI stargets/LUNs, the shared data could get corrupted. Therefore, the essential idea behind our design is to coordinate the accesses of multiple IP hosts functioning as iSCSI initiators.

Another potential candidate to coordinate the concurrent accesses is Device Lock mechanisms from GFS, which has been included in SCSI 3 specification as Dlock command. Device locks are advisory locks implemented on SCSI devices to allow mutual exclusive accesses of shared resources. The Device Lock mechanism requests the appropriate device lock before accessing (either reading or writing) any data, and releases the lock when finished. However, it is up to the user applications to issue locking request to lock the shared resources, such that the concurrency on the shared resources will suffer by inappropriate locking request. Moreover, devices supporting the locks have no awareness of the nature of the resource that is locked. This ends up with inefficient lock management on the devices.

In order to improve system performance, file systems typically use various in-memory caches to hide the latency caused by mechanical devices. As iSCSI extends the connections from LANs to WANs, the latency introduced by large physical distance makes caching on IP hosts indispensable. Caching mechanisms have been proven to be an effective way to improve the performance in NFS [5]. The environment of iSCSI-based file systems is similar to NFS, in that data potentially need to be transferred over WANs. While caching mechanisms improve the performance dramatically, it always introduces the problem of caching consistency, which is the consistency between the cached data on the client side and the data on the server side. In both iSCSI-based file systems and NFS, there could be multiple clients, which could cache the same data at certain point of time. Without mechanisms to ensure cache consistency, modification of any copy of the data will lead to cache inconsistency. NFS does not really solve this problem. It only alleviates such effects by setting expiration time on cached data. Depending on the type of the data, the length of the expiration period would be different. However, it is still possible that stale data would be accessed. In many applications, such accessing of stale data is not acceptable. In our design, we will enforce strong consistency on cached data.

In addition, the expiration is not accurate enough, in that a cached item may have expired even if it is still consistent with the server side. More accurate scheme should be used to avoid any unnecessary expiration, since the distance latency is dominant in such environments. In our design, we use a callback cache similar to the Coda file system [7]. An iSCSI target keeps track of cached physical blocks on all connected iSCSI initiators. The iSCSI target forces the iSCSI initiators to discard their stale copy, when it is going to modify a physical block.

4 Design Overview

Our proposed scheme for data sharing in iSCSI-based file systems consists of two major parts. The first part is a concurrency control mechanism to coordinate multiple concurrent accesses for data on shared iSCSI target/LUNs. In our design, we add locking management components into iSCSI target devices to manage locks for data in one of their LUNs. Individual iSCSI target/LUN is independent, in that there is no single operation involving data from more than one iSCSI target/LUN. In real implementation, depending on the capability of iSCSI target devices, these added components could sit either on the same devices, or on a separate server functioning as a lock manager.

Based on observations that the access patterns of metadata and normal data are different, we designed different locking schemes for metadata and normal data, respectively. From the study done by Roselli et al. [8] on read-write access pattern of metadata objects (super block, bitmap blocks, and inodes), it has been found that the percentage of metadata reads is much larger than metadata writes. Hence to take advantage of this fact, our design allows the iSCSI initiators to cache the shared locks (referred to as semi-preemptible locks [9]) on metadata object so that we can reduce the read traffic from the iSCSI initiators to the iSCSI target. Once an iSCSI initiator has acquired a shared lock on a metadata object it can keep reading that metadata object without contacting the iSCSI target unless the iSCSI target asks it to discard that lock. In that case it would need to acquire the lock again for the next read on that metadata object. Nevertheless, the fact that a large percentage of metadata accesses are reads greatly improves throughput by reducing read latency.

On the other hand, normal data are organized into files. The access patterns for files are application dependent. Therefore, we are trying to design a general locking scheme with certain goals, in addition to integrity of shared data, which is the very basic requirement. Our first goal is to achieve high concurrency when the shared files are accessed by multiple iSCSI initiators simultaneously. It is actually about the granularity of the locks. With very fine granularity, the concurrency could be maximized. The second goal is to reduce the costs related to locks. One part of the costs is the network communication. Since the locking management components are not within iSCSI initiators, any locking requests have to go through IP network. In iSCSI-based environment, such delay could be significantly high, so we want to reduce the number of requests. Another part of the costs related to locks is the memory space used by the locking management components. It is always desirable to use less amount of memory, so that more locks any iSCSI target/LUNs could be supported. However, these two goals are conflicting with each other. To maximize concurrency, fine granularity of locks are preferred, meanwhile coarse granularity of locks reduces number of lock requests and memory space used to maintain locks. Our design is try to balance between these two conflicting goals. For normal data, we employ hierarchical locking scheme. The detail of this scheme will be discussed later in section 4.2.

The second part of our proposed scheme is a cache consistency control mechanism. The physical blocks fetched over networks are cached in iSCSI initiators' buffer caches. Buffer caches will be checked first when a physical block is requested. In order to avoid revalidating consistency of cached data blocks when they are used, we employ a mechanism based on callback. A callback record will be set up on iSCSI target side, when a physical block is read out. When an iSCSI initiator is going to write a physical block, it first sends a SCSI CDB with write request. The iSCSI initiator will wait for a R2T response from the iSCSI target before starting transmit data. When a iSCSI target receives a SCSI CDB with write request, it will check the callback records for the requested physical blocks. If there are standing callback records, callback requests will be sent to those iSCSI initiators to ask them to purge the requested physical blocks out of their buffer caches. The iSCSI target will not send R2T response until it receives confirmations for all callback requests it sent out.

	M_S	M_X
M_S		+
M_X	+	+

Table 1: Compatibility of metadata locks

In the rest of this section, we are going to discuss our locking schemes for metadata and normal data in detail. We will also discuss the transaction file sharing semantics, which supports transaction execution of a group of operations.

4.1 Locking scheme for metadata

The locks for metadata are applied on metadata objects. We define following 5 kinds of metadata objects: 1) directory files, 2) normal files, 3) super block, 4) inode bitmap blocks, and 5) data-block bitmap blocks. For a directory file object, it includes the inode for that directory file as well as the directory blocks, which contains directory entries. For a normal file object, it only has the inode and indirect blocks for that file. The reason we treat the directory files and normal data files differently is because that the directory entries are still a kind of metadata. In addition, operations on directories are more deterministic, and often include search through the entire directory blocks. Currently, we do not support special files, since we don't see the necessity at this moment.

For each metadata object, there are two possible kinds of locks: M_S and M_X . M_S is the metadata share lock, which gives shared access to the requested metadata object. M_X is the metadata exclusive lock, which gives exclusive access to the requested object. The compatibility of the locks is shown in table 1, where '+' indicates incompatible of the two locks.

The M_S lock is semi-preemptible in that the requestor is allowed to hold a M_S lock until the lock manager asks the requestor to release that M_S lock. Therefore, the M_S is actually cached at the initiator, and the callback mechanism is used to force the holder to release the lock, when some initiator requests M_X on the same metadata object. However, the holder of a M_S lock should not allow the lock to be called back at any moment. If the holder is in the middle of an operation, which includes accesses to the metadata object referenced by the M_S lock, the holder should reply to the callback request only after this operation is finished. In our implementation, there is a counter associated with each cached M_S lock. The value of the counter indicates how many active instances of the M_S lock exist. When one operation involving an instance of the M_S lock is finished, the counter is reduced by 1. Therefore, when a callback request for a M_S lock comes, if the counter associated with this M_S lock is greater than 0, the reply for the callback request has to wait until the counter decreases to 0. New operations involving shared locking on the metadata object referenced by the M_S lock could keep coming, such that the counter associated with the M_S lock is always greater than 0. To avoid the happening of such situation, a cached M_S lock can no longer serve lock requests from operations, once there is a waiting callback request on that M_S lock.

A M_X lock for a metadata object is requested by an initiator when it is going to modify the metadata object. The M_X locks will not be cached at the initiators, so initiators have to contact targets every time. When the iSCSI target receives a M_X lock request, it will check its locking table to find out the current locking state of the requested metadata object. If there is any waiting request in the waiting queue, the new request has to enter the end of the waiting queue. If the waiting queue is empty, but there is another M_X lock in the locking queue, the new request still needs to enter the waiting queue. If there are one or more than one M_S locks in the locking queue, the target needs to send callback requests to each initiators holding these M_S locks to ask them to release their cached M_S locks. As we described before, these initiators will send back reply immediately or after finishing current operations, depending on whether there are operations using the cached M_S lock or not. A M_X lock is always released after the involved operations have finished.

Another possible operation on a M_S lock is to upgrade it to a M_X lock. This happens when an metadata object is first read, and hence a M_S lock is cached locally. Later, a write request for the same metadata object arrives at the same initiator. A new M_X would conflict even with the cached M_S , so an upgrade request for the cached M_S lock to a M_X lock is sent. When the metadata lock manager sitting on the target receives such upgrade requests, it will only allow such upgrading when no other M_S locks were approved for the same metadata object. Otherwise, callbacks have to be sent to all other initiators

	D_S	D_X	D_IS	D_IX
D_S		+		+
D_X	+	+	+	+
D_IS		+		
D_IX	+	+		

Table 2: Compatibility of hierarchical locks

holding these M_S . The approval for the upgrade request can't be sent, until all callback requests are replied. Whenever there are more than one upgrade requests for a metadata object, a deadlock happens, because each of them can't be approved until one of them release its M_S lock.

When a M_X lock is released by an iSCSI initiator, locking requests in the waiting queue of the metadata object could be arouse. Since any locking request always enters the waiting queue at the end, the sequence of the requests in the waiting queue is corresponding to their arrival sequence at the iSCSI target. In case of metadata locking, since the M_X is exclusive, there could not be another active lock in the locking queue when there is a M_X lock. The process of arising is to find the prefix of the sequence of requests in the waiting queue, such that this prefix sequence is compatible. Such compatible sequence could be a sequence all of M_S requests, or a sequence all of M_S requests followed by a M_X request. In the second case, the last M_X request will actually make the target send callback break to initiators of the preceding M_S requests.

4.2 Locking scheme for normal data

Normal data are organized into files. As we have discussed in previous sections, maintaining data integrity and maximizing concurrency are two conflicting goals. We choose the compromising scheme - hierarchical locking scheme in our design.

Before we show how data bytes of a file are organized into a hierarchical tree, let's take a look at the 4 possible modes of locks applicable to the nodes of the tree.

- D_IS : Gives intention share access to the requested node and allows the requestor to explicitly lock descendant nodes in D_S or D_IS mode. It does no implicit locking to the sub-tree rooted from the requested node.
- D_IX : Gives intention exclusive access to the requested node and allows the requestor to explicitly lock descendent nodes in D_X , D_S , D_IX , and D_IS mode. It does no implicit locking to the sub-tree rooted from the requested node.
- D_S : Gives share access to the requested node and to all descendants of the requested node implicitly.
- D_X : Gives exclusive access to the requested node and to all descendants of the requested node implicitly.

Intention mode is used to indicate the fact that locking is being done at finer level and thereby prevents implicit or explicit exclusive or share locks on the ancestors. The table 2 gives the compatibility of the requested lock modes, where '+' means conflict.

The implicit locking of nodes will not work if lock requests are allowed to leap into the middle of the hierarchy tree. To avoid this, there are several rules that must be obeyed:

Rule 1: Locks are requested from root to leaf. Before requesting D_S or D_IS lock on a node, all ancestor nodes of the requested node must be held in D_IX or D_IS mode by the requestor.

Rule 2: Locks should be released either at the end of the transaction (in any order) or in leaf to root order.

Rule 3: The leaf nodes are never requested in intention modes since they have no descendants.

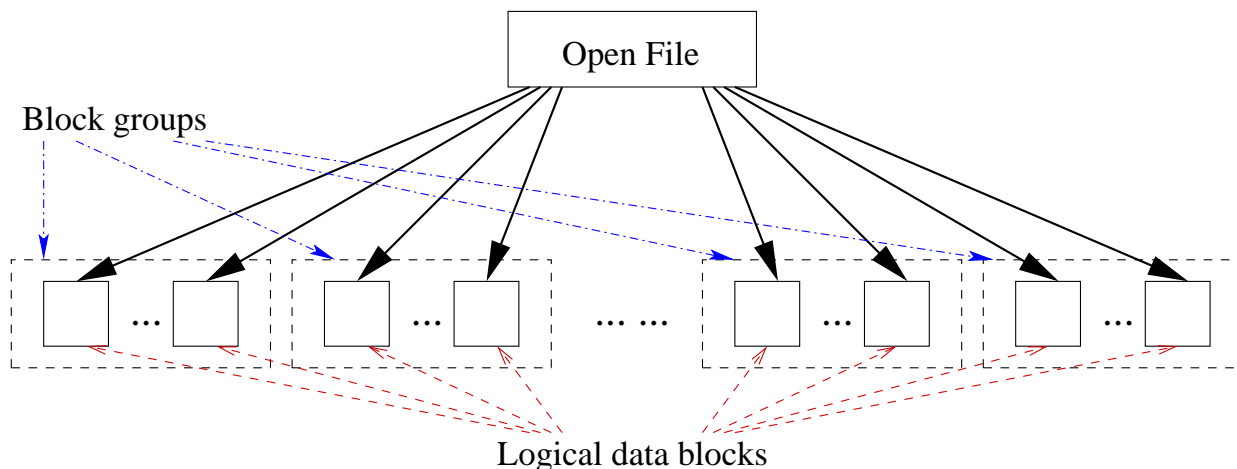


Figure 4: Two-tier file data organization

With implicit locking, hierarchical locking allows large transactions which access many blocks of the disk file to set large locks without high lock overhead, and small transactions which access few blocks of the disk file to set small locks to achieve a high concurrency.

The bytes of a file are organized into a hierarchy tree. The top level, i.e. the root of the hierarchy tree, is the whole file of all bytes. At the second level, the whole file could be divided into any partition of the upper level, and the third level is to further divide each parts of the second level in a recursive way. Each node of the hierarchy tree can be locked. The goal of the hierarchy-locking scheme is to reduce the lock request messages by implicitly locking an entire sub-tree. Therefore, deep hierarchy has more flexibility in locking, and allows higher concurrency. In theory, we could apply arbitrary level of hierarchies to a file anyway down to a single byte at extreme. However, deeper hierarchy needs more memory, and takes more CPU time to perform management tasks.

To balance between high concurrency and high resource consumption, we design a two level hierarchy, which has low resource consumption, and supports good concurrency at the same time. As shown in the figure 4, the top level is the file, and the next level, which is the leaf level as well, is fix-sized group of blocks. The basic unit of iSCSI accesses is the physical block, so the size of block group is at least 1. Since there are only two levels in our design, the size of the block group determines the potential degree of concurrency. If the size is large, there are less number of groups for a file, and the maximum achievable concurrency is low. If the size is small, there are more groups, and the maximum achievable concurrency is higher. The extreme condition for large group size is the whole file as a group. Then, only one exclusive lock is allowed for the whole file. The other extreme for small group size is a single physical block per group. Under this extreme, each block could have an exclusive lock without conflicting with others, which maximizes the concurrency. However, it is obvious that such fine granularity causes heavy overhead in locking messages. Our simulation inspects various group sizes, and evaluates the impact of group size on overhead, and performance.

4.3 Transaction file sharing semantics

Many systems only supports unix file sharing semantics. Under unix file sharing semantics, writes are mutually atomic, and the result of a write is immediately visible to all readers. However, the Unix file sharing semantics is not suitable for transactional execution of a sequence of operations. In the transaction file sharing semantics, there should be a consistent view of any involved data throughout the execution of a transaction.

In our design, file-accessing operations are grouped into transactions. At the beginning of a transaction, an iSCSI initiator needs to send a start transaction requesting to an iSCSI target, such that the target can create corresponding data structure to monitor this transaction. The transaction will be assigned a unique transaction id within the session between the iSCSI initiator and the iSCSI target. The detailed data structure for supporting transaction will be discussed in section 5.

In addition, before any real read/write operation, all files involved in the new transaction should be opened with proper access mode. Currently, our design only considers *READ_ONLY* and *READ_WRITE* modes. It is also possible to add other flag such as *TRUNCATE* or *APPEND* presenting in most Unix systems. If a file open request is approved, a file level hierarchical lock will be granted. Otherwise, the requested open mode should be conflicting with some existing ones. The following file access operations need to request block group level hierarchical locks on appropriate block groups.

At the end of a transaction, all locks held by the transaction will be released. A transaction can hold metadata locks as well as normal data locks. For share metadata locks *M_S*, since our scheme caches it at the initiator, only a counter associated with the cached copy is decremented by 1 to indicate that there is one less active instance of this cached share metadata lock. For normal data locks and exclusive metadata locks *M_X*, released requests are sent back to the target. However, to improve the performance, only one close transaction request is sent back to the target, and the target will release all locks held by this transaction. In section 5, we will show that the target actually have data structures to record all such necessary information.

Deadlocks are going to happen, since we are supporting transactions. Due to the nature of random access of the file data, it is difficult to prevent the happening of deadlocks. Therefore, in our design, we use deadlock detection mechanism to detect deadlock. When detecting a deadlock, a victim transaction will be selected and rolled back. The selection of such a victim transaction could be based on various criteria. One way is to evaluate the progress of the transactions involved in the deadlock, and select the one with the least progress as the victim to break the deadlock. This will require each transaction to monitor their own progress, in terms of how many steps of operations it has taken. Intuitively, the removal of the least-progressed transaction will reduce the waste of processing. In addition to minimizing waste of processing, the fairness problem is another concern. Some transaction will be prevented from finishing, since it is always selected as the victim to break the deadlocks. Additional monitoring information could be associated with each transaction to indicate how many rollbacks have been applied to it. Then, the final decision would be based on a cost function involving both the progress of each transaction and number of times that the transaction has been rolled back. However, a detailed discussion of such cost function and other selection criteria are beyond the scope of this paper. Our current simple implementation is to choose the owner of the latest lock request, which cause a deadlock.

The mechanism to detect deadlocks is to find a loop among transactions and locks. Each transaction has table to record what locks it is holding. Meanwhile, each lock has queues to indicate which transactions are holding this lock and which transactions are waiting on this lock. Following these tables and queues, a loop can be detected, which indicates the happening of a deadlock. In the implementation section, we will show such tables and queues associated with transactions and locks.

4.4 File-level admission control

Deadlocks will decrease the system performance. In order to reduce the occurrence of deadlocks, we propose a file-level admission control mechanism. We assume that each transaction knows an estimation of the percentage of data for each file that the transaction is going to access. Transactions always open files with such estimated percentage values as parameters. A transaction could open a file in *READ_ONLY* mode, or *READ_WRITE* mode. For *READ_ONLY* mode, since only shared locks will be requested for reading, the estimated percentage is just ignored. The reason is that we hope a large number of *READ_ONLY* transactions to co-exist. However, for *READ_WRITE* mode, the exclusive locks could prevent other requests. The estimated percentages of *READ_WRITE* openings are always accumulated. The system enforces a threshold on this accumulated value. Once it has exceeded the threshold, the new open requests with *READ_WRITE* mode will be put into waiting queues.

When a transaction closes a file at the end, the estimated percentage will be subtracted from the accumulated value. If the accumulated value is now below the threshold, the waiting transaction will not be waken up until the running transactions all release their lock on the file (either finished or rollback). The reason is to avoid high possibility of deadlocks of allowing new admission immediately, as the existing transactions already have a high accumulated percentage.

5 Detailed design and data structures

The figure 5 shows the architecture overview of our implementations. We insert new modules in to both iSCSI initiators and iSCSI targets. In iSCSI initiators, vfs is used between the upper level system call layer and the lower level iSCSI layer. we have inserted following two modules into the kernel of iSCSI initiators.

- **iSCSI client module** is actually a modified ext2 file system module. It manages transactions and various metadata and normal data locks.
- **Initiator cache manager module** manages a dedicated buffer cache for the iSCSI client module. It supports callback mechanism to assure cache consistency.

iSCSI targets are responsible for maintaining active transactions, maintaining opened files, supporting callbacks for cached physical blocks, and so on. In iSCSI targets, we have inserted following three modules into iSCSI targets' kernel.

- **Metadata lock manager module** manages lock requests for metadata objects mentioned in section 4.1.
- **File lock manager module** manages transaction requests, file open/close requests, and block group lock requests. It is also responsible for deadlock detection and resolution.
- **Target cache manager module** maintains callback records for physical blocks cached at iSCSI initiators. When there is a SCSI CDB with write command triggering callbacks, this module is also responsible for suspending the write command until all conformations for callback requests are received.

In the following subsections, we are going to discuss each of above modules in detail.

5.1 iSCSI client

A iSCSI client manages transactions as well as the metadata locks and normal data locks belonging to the transactions. The figure 6 shows the components of the iSCSI client.

One transaction has a data structure to store associated information. This includes transaction number, transaction type, and type-specific transaction parameters. In addition, a transaction usually opens several files to perform operations, so a transaction has its own open file structures to store management information. In particular, an open file structure contains following fields: a mark to indicate whether this structure is in use or not; the name of the file; the mode of opening, which could be *READ_ONLY* or *READ_WRITE*; the estimated percentage of access; a pointer to the rnode of the file; a file descriptor assigned by the iSCSI target; metadata lock type and id of the r-node of the file; a hash table to store block-group locks; a link list to store dirty physical blocks. The hash table for block-group locks performs the hash function on the group number. Actually, there is no restriction of the hash function to be used. Even more, other structure could be used instead of hash table, such as binary search tree. The function of the link list is to store dirty physical blocks. As we mentioned before, in order to support roll back function of transactions, dirty blocks are temporarily stored within the space of the transaction until the commitment of it. Therefore, our design is to store the dirty blocks together with the open file structure.

Another important function of the iSCSI client is to manage the rnodes. The rnodes are corresponding to the vnodes within the vfs layer. However, rnodes could contain more information than the vnodes. Even for the same information, the format could be different. Our rnodes contain all common information in Unix disk inodes, such as the mode, the number of links, the user id, the group id, the size, the timestamps, the physical block mapping. Additionally, rnode contains some fields not presenting in disk inodes, such as referencing counter and dirty bit. The referencing counter is used to record the number for active usage of the rnode. When the value of counter decrease to 0, the space is release for other use. The dirty bit is used to indicate whether the fields corresponding to disk inode's fields are modified, and hence whether writing back is necessary or not.

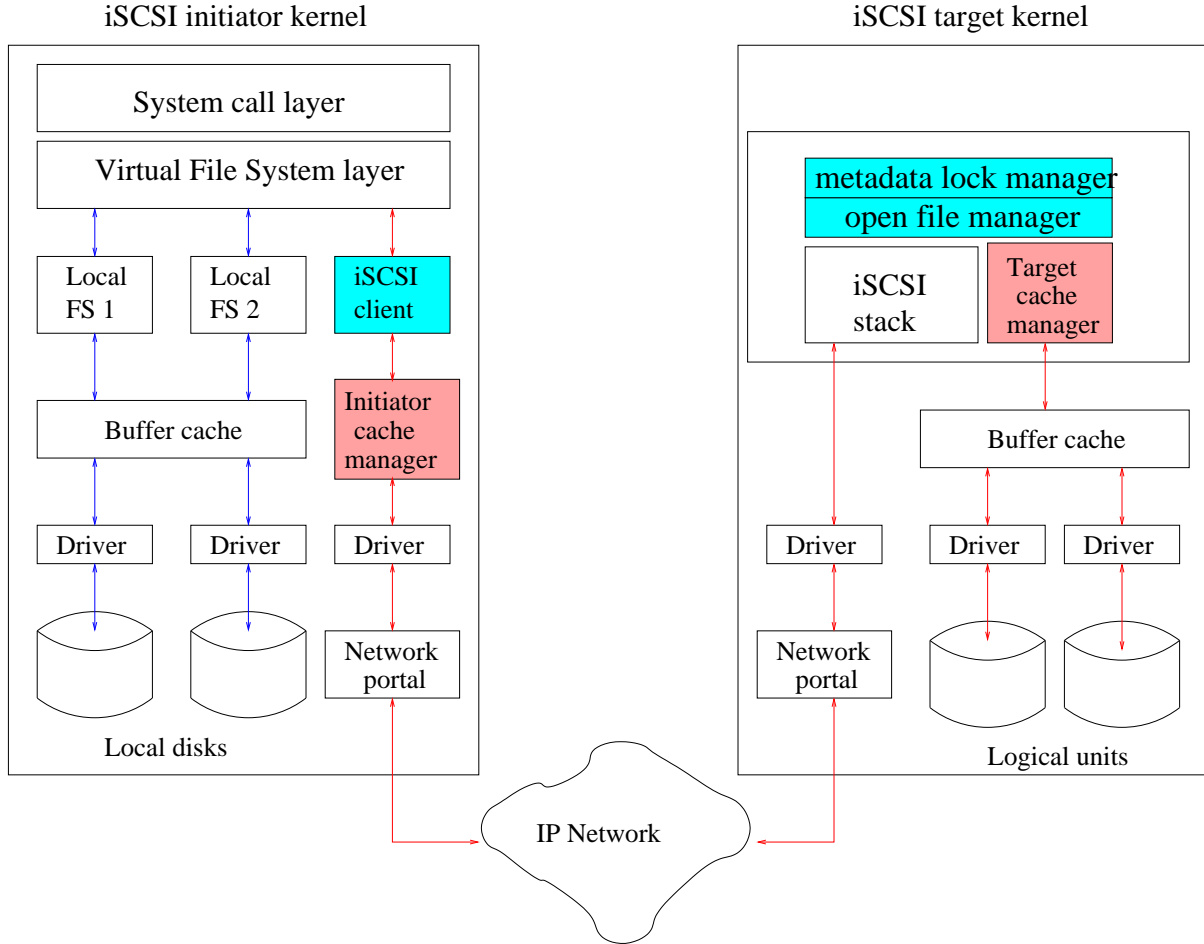


Figure 5: Overview of the architecture

5.2 Initiator cache manager

In our design, the ICM, which is the dedicated buffer cache for iSCSI, is separate from the buffer cache for other file systems. The main reason is that the ICM is using a cache coherence protocol based on callbacks.

In addition, the ICM works in a unique way from traditional buffer cache in writing operation. In traditional buffer cache, writing operation could use either write-through policy or write-back policy. However, from performance point of view, most systems use write-back policy, which only write-back dirty blocks when the cached blocks are going to be purged out from the buffer cache. In contrast, write-through policy sacrifices performance to achieve higher reliability, since any modification would be written back to non-volatile storage device with minimum delay. In some sense, the ICM is more like a write-through cache, in that it initiates a iSCSI write command when a dirty block is put back. In fact, the ICM works together with the link list of dirty physical blocks of the transaction structure to form a cache hierarchy. When the operation of a transaction needs a specific physical block, it will use a procedure call *get_block()* with the block number as parameter. The *get_block()* procedure always check the transaction's link list of dirty physical blocks to see whether this transaction has already accessed and modified the requested block before. If the requested physical block is not in the link list, the *get_block()* procedure will request it from the ICM. The ICM will always return the requested physical block either directly, when it caches such a physical block, or through iSCSI protocol, when it does not have such a copy. In the later case, the ICM will cache the physical blocks grabbed through iSCSI.

In above, we haven't mentioned the relation between the locking mechanism and the caching mechanism. A transaction always uses *get_block()* procedure to get desired physical blocks. However, before doing that,

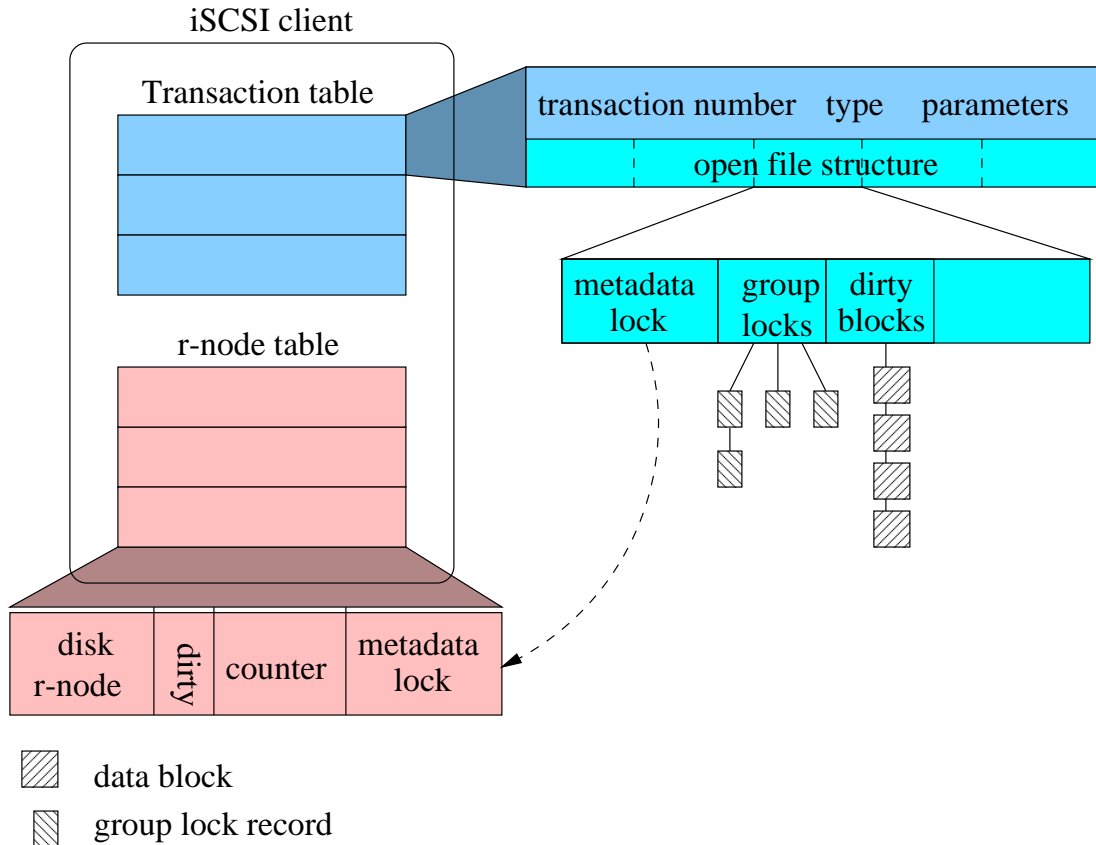


Figure 6: Components and data structures within a iSCSI client

the transaction has to go through locking mechanism to get proper locks, such that its access will not be conflicting with anyone else. For metadata objects as described in section 4.1, a metadata lock is requested with proper mode, depending on what kind of operation is going to be performed. Only when such locking process has been passed successfully, which returns a lock id, the transaction can call *get_block()* procedure with the lock id. Since a physical block is the minimum unit for iSCSI protocol, for metadata object such as an inode, the entire physical block, which contains the requested inode and several other inodes, is retrieved through *get_block()*. The normal data blocks are always accessed when the file containing these normal data blocks is opening. As described in section 4.2, there is a file level lock when the file is opened. The second level of locking in the hierarchy is the block group. When a file is opened with a *F_IS* or *F_IX* lock and the transaction is going to access the blocks, the transaction has to get a proper lock, either *F_S* or *F_X* on the block groups.

5.3 File lock manager

The file lock manager is responsible for managing transactions, files opened by these transactions, and block group locks for data of these opened files. It is also doing deadlock detection and resolution. As show in figure 7, this module manipulates two data structures - a transaction table and a open file table.

The transaction table has fixed number of entries. Each entry of the transaction table records the information of one transaction. One entry has fields like iSCSI initiator number and Logic Unit Number (LUN) to identify which iSCSI initiator and which LU the transaction belongs. Transactions are identified by their transaction numbers, which is just the index of the corresponding entries in the transaction table. There are five locking queues in one entry of the transaction table. Two of them are file-level locking queues, and the other three are block-group-level locking queues. The two file-level locking queues are holding-file-lock queue, and waiting-file-lock queue. The holding-file-lock queue records the granted file-level locks of

the transaction. The waiting-file-lock queue records the file-level lock for which the transaction is waiting because of presenting of conflicting locks. The three block-group-level locking queues are the holding-grp-lock queue, the waiting-grp-lock queue, and the upgrade-grp-lock queue. The first two of them are similar to the corresponding file-level queues, except they are at the level of block groups. The upgrade-grp-lock is for the case that a transaction first requests and is granted a *F_S* shared lock on a block group to perform read-only operation, and later it would like to perform a write operation on the same block group, which requires a *F_X* exclusive lock. This will cause an upgrade of lock level, since *F_X* locks are stronger than *F_S* locks. When the upgrade of a *F_X* lock will result in confliction, the lock is moved from the holding-grp-lock queue to the upgrade-grp-lock queue. The transaction is still holding it, and the deadlock detection has to take such locks into consideration.

The open file table maintain all opened files in a specific iSCSI target/LUN. The number of entries of this table is fixed as well. A file is referenced by an integer number called file descriptor after it has been opened. The file descriptor is indeed the index of the entry of the file in the open file table. One entry of the open file table has fields like the LU Number (LUN) of the file, the inode number of the file on its LU, and an accumulative percentage field to records the sum of the estimations of those opening instances that have already successfully open the file. The last one is used to perform the file level admission control as discussed in section ???. An open-file-table entry has two file lock queues. One is the holding-file-lock queue, and the other is the waiting-file-lock queue. In addition, one open-file-table entry has a hash table for block groups. This hash table is a dynamic data structure in that the entries are created a inserted dynamically. On entry of this hash table represents one block group, and has three block-group-level lock queues. They are actually the holding-grp-lock queue, the waiting-grp-lock queue, and the upgrade-grp-lock queue that are corresponding to the three block-group-level queues of the transaction structure.

The transaction table and the open file table are related two each other by sharing lock records between corresponding lock queues. As shown is figure 7, each records, either a file-level one or a block-group-level one, has two pointer fields of its own type. Then, two link lists are constructed respectively. Following one of these two link lists, one can find all records belonging to the same transaction. On the other hand, following the other link list, one can find all records belonging to the same open file. This sophisticated data structure is used to detect deadlocks. Starting from a suspicious transaction, if there is a loop of pointers to come back to the same transaction, a deadlock happens.

5.4 Metadata lock manager

The metadata lock manager module manages locks for super block, free space bitmaps, free inode bitmaps, and inodes. For the first three kinds of metadata objects, there are one possible lock record for each iSCSI target/LUN. For locks for inodes, a hash table is used to facilitate management, similar to the hash table used for block group locks in open file tables. If there are lock requests for an inode, one lock record for that inode will present in the hash table. One such lock record has three queues - a lock_queue, a wait_queue, and a convert_queue. In addition, it has a callback element to indicate whether a callback request has been previously sent for this inode. If the callback element is not empty, it contains a counter, which is the expected number of confirmations for that callback request.

5.5 Target cache manager

As show in figure 7, iSCSI target cache manager modules use a simple and efficient mechanism to maintain the callback for cached physical blocks. The data structure is a bitmap for each iSCSI target mounted on each LU. In other words, when an iSCSI initiator is mounting a LU on the iSCSI target, the iSCSI target allocates a bitmap for this. In the bitmap, each bit represents a physical block, just like the free space management in typical Unix file systems. Initially, the entire bitmap has 0 on every bit. When an iSCSI initiator uses iSCSI READ command to retrieve a physical block, the corresponding bit in the proper bitmap is set to 1. In fact, the bitmaps need not necessarily to be dynamically allocated at mounting. They could be allocated statically, since even a large storage device will need small space for bitmap.

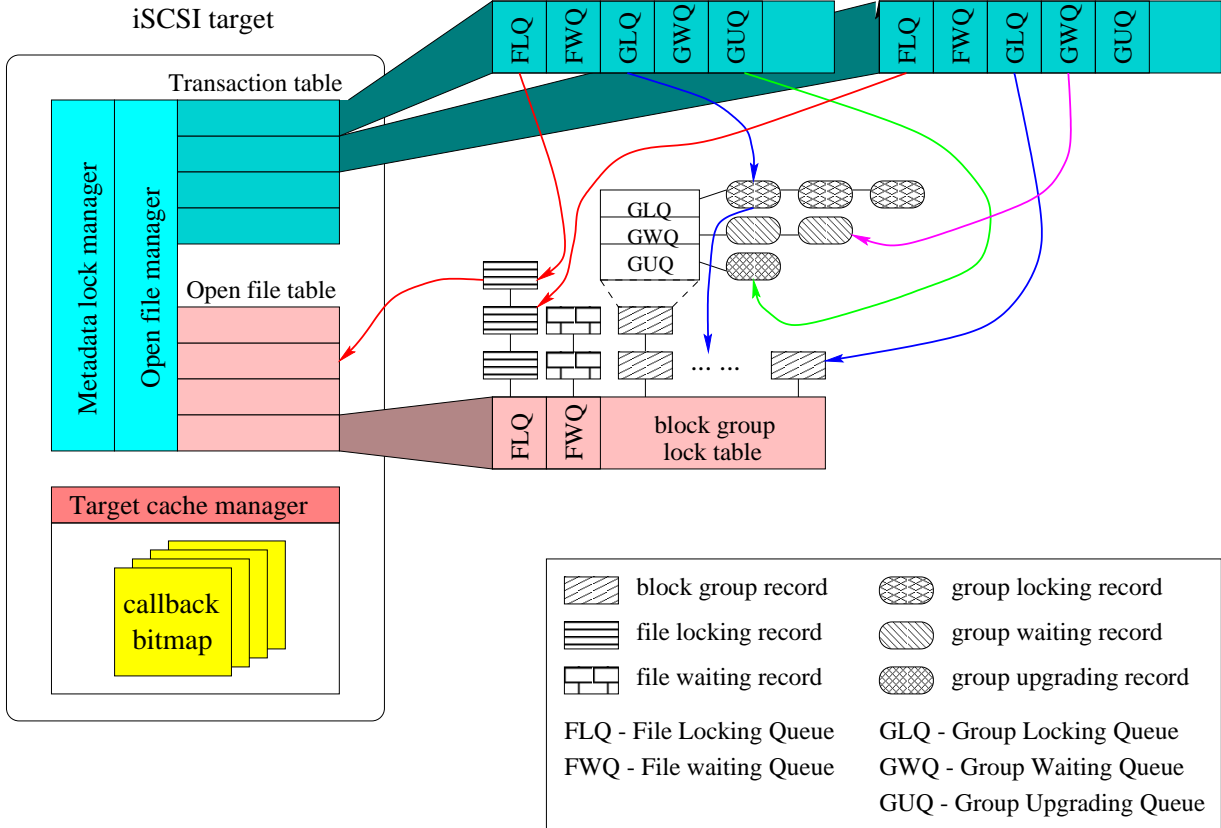


Figure 7: Components and data structures within a iSCSI target

6 Simulation result

In this section, we present simulation-based performance results of our proposed iSCSI data sharing scheme. This ns-2 [10] network simulator was used to implement application-level iSCSI initiators and iSCSI targets. An iSCSI initiator contains components, such as an iSCSI client and an iSCSI initiator cache manager. An iSCSI target application could manage multiple LUs, by manipulating data structures associated with each LU. Such data structures include transaction tables, open file tables, and so on. At this stage, we still assume a transaction only access data within a single LU, such that the management of LUs are independent from each other. Our simulation skips some iSCSI management functions, such as session setup, since we are focusing on the behaviors of data block transmission. Furthermore, our protocol is designed to extend the current iSCSI protocol (we use reserved command of the iSCSI protocol, so the message format fits into the protocol completely), so we don't need to worry about the functions that are supported by the current iSCSI protocols, include these skipped functions. In addition to simulation, we are continuing working on implementation of our scheme on Linux platform. We are extending the ext2 file system over cisco iSCSI driver implementation.

There are three parts in the rest of this section. In the first part, we simulate sequential and non-concurrent reading/writing of a large file. This is trying to show pure overheads of our locking scheme, compared with real reading/writing of physical blocks. The 'pure' means that no waiting time for locking requests is included, since an iSCSI initiator accesses a file exclusively in this set of simulations. In the second part, we are trying to show the performance of our scheme under OnLine Transaction Processing (OLTP) scenarios. We use trace data generated according to the TPC-C benchmark of the Transaction Processing Performance Council to drive our simulations. Each transaction could involve reading or writing of multiple files. In these simulations, locking requests could be put into waiting queues. Even worse, deadlocks could happen, so that some transactions have to rollback and restart. With this set of close-to-real simulations, we

will show the amount of saving of locking requests by doing caching of both metadata locks and normal data locks. In the third part of this section, we are going to show the behavior of the system when a single file is randomly accessed by multiple initiators concurrently. This part is different from the second part, in that each transaction is only accessing a single file in this part. We are going to show the performance impact of variable system parameters, such as physical block size, block group size, and so on. This study will give a guide on the system design.

Before going into the simulation results, we first list various system parameters we have used. In all of our simulations, the disk modules use the parameters from Seagate's Cheetah 15K.3 family disk drives. The *Average Latency* is 2.0 msec. The *Average Seek Time* for Read/Write is 3.6/3.9 msec. The *Internal Transfer Rate* we used is the average of its *Sustained Transfer Rate* (49 to 75 Mbytes/sec, which is 62 Mbytes/sec. However, in our simulation, no cache of the disk module is assumed. Therefore, the delay for access one block of data, once the command leaves the waiting queue and gets executed, is computed as $Delay_{Read/Write} = AverageLatency + AverageSeekTime_{Read/Write} + BlockSize/InternalTransferRate$. We also assume that a target can only process locking requests sequentially, so the response for each request is delay for certain amount of time from the previous response in the target. The delayed time is the time the target takes to look through its locking records to make a decision on whether to grant the request or not. For the network latency and bandwidth, we designed three combinations to represent three typical scenarios. The first is local area networks (LAN), which has latency of 1ms, and bandwidth of 100Mbps. The second scenario is medium-wide area networks (MAN), which has latency of 50ms, and bandwidth of 1Mbps. The last scenario is large-wide area networks (WAN), which has latency of 100ms, and bandwidth of 1Mbps. The table ?? is a summary of these parameters.

6.1 Sequential Reading/Writing

To investigate the overhead of the concurrency control and cache consistency scheme, we run the simulations for sequential writing of a single file. The file size we used in our simulations is 100MB. For writing of each physical block, we assume it is only a partial overwritten, such that the physical block should be read first, before written back. When doing reading and writing of physical blocks, we let the driver to do them individually for each physical block.

- For each of LAN, MAN, and WAN scenario, I am going to show the total transaction time used for different physical block size (1kB, 2kB, 4kB, and 8kB) and different block group size (1, 2, 4, 8) correspondingly. This will show that larger physical block size will cost less time, and larger group size will save on locking requests for block groups.
- I will use a bar graph to show the composition of the total transaction time. The components are time on reading/writing physical blocks, time on locking metadata (inodes, freebitmap), time on locking file data (file-level locks and block group locks), time on open/close transactions, and others. In this set of simulations, since there is no concurrency, the time for locking requests will be the pure cost on transmitting of requests. This will show that the pure overhead of locking requests is small compared with time spent on reading/writing of physical blocks.

6.2 Trace Simulation

The TPC-C benchmark is an OLTP benchmark for database systems. Since our current system application is a file system, we adapt the benchmark to our needs. The TPC-C benchmark involves a warehouse management database with 9 relation tables. We view each relation table as a file, storing fix-sized records consecutively in the logical blocks. The TPC-C benchmark has 5 different transactions in SQL. Each transaction includes reading, writing, or both for records in multiple relation tables. To fit into our requirements, we only trace the location of real reads and writes of records in the table files. We totally ignore the additional metadata such as index for keys in the real database systems. The only metadata we are currently concern are the inodes of the table files, and the bitmaps for free space management. However, as we go forward to implement the database application, all other metadata access could be involved.

When generating the warehouse management database, we set up 2 warehouses. Each warehouse covers 10 distinct districts. There are a number of customers registered to a certain district. We set up 6 client

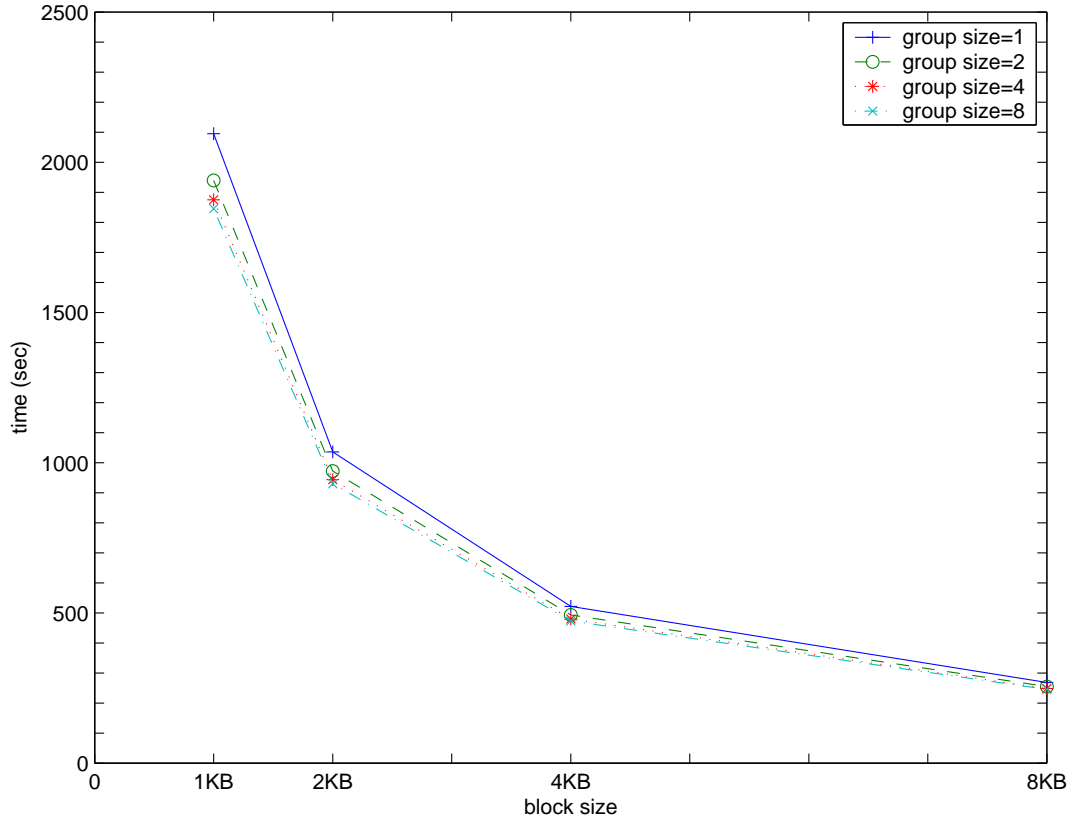


Figure 8: Total transaction time for sequential access in LAN

terminals to generate transactions concurrently. Each client terminal generates 200 transactions during the simulation. Among these transactions, 45 percent of them are new-order transactions, 43 percent of them are payment transactions, 4 percent of them are order-status transactions, another 4 percent of them are delivery transactions, and the rest 4 percent of them are stock-level transactions. There are 3 client terminals associated with each of the 2 warehouses. When a client terminal is bound to a warehouse, all its transactions will be operations on records belonging to that particular warehouse. Therefore, when multiple client terminals are bound to the same warehouse at the same time, they will have a high probability to sharing the same data pieces. Client terminals of different warehouses will share metadata of the files concurrently. The trace data generated by a client terminal is used to drive a iSCSI initiator in our trace simulation accordingly.

- I am going to show the total transaction time of each of the 6 initiators under different combinations of physical block size and block group size. This will show that larger physical block size and block group size are not always good in transaction processing, since they will cause more locking conflicting, so that locking request is going to wait longer, and even run into deadlocks.
- I am going to show the composition of the total transaction time. This will show that the time spent on locking block groups becomes the major part of the total transaction time. By varying physical block size and block group size, I will show that larger physical block size will cause longer read/writing time, since OLTP only access files randomly, instead of sequentially. Meanwhile, larger physical block size also cause long time spent on locking file data, since it cause more locking conflicts.

We compare our proposed iSCSI data sharing scheme with the simple UNIX locking scheme. In order to support UNIX file sharing semantics, which requires that the result of every write operation is immediately visible to subsequential read operations, UNIX locking scheme requests locks on each file read or write system

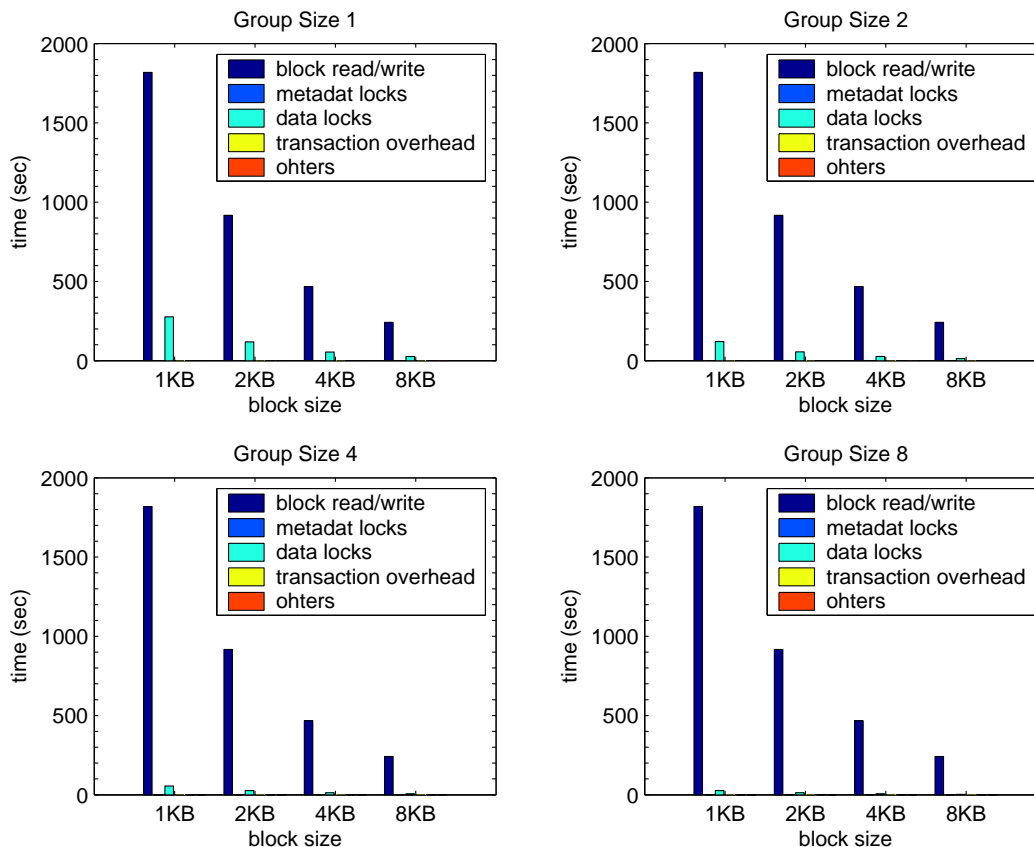


Figure 9: Composition of total transaction time for sequential access in LAN

	iSCSI locking scheme	UNIX locking scheme
inode locks	387 + 191	3039454
block locks	40045 + 9829	3017706

Table 3: Comparison to UNIX locking scheme

call. In addition, these locks should be released immediately upon return of the system calls to allow the written result to be available to others, since there is no concept of transaction. Similarly, the access to the metadata, including the inodes of files, runs into the same problem. Systems calls, such as *read_map()*, which reads the inode and possibly the indirect blocks of a inode to map a logical address to a physical block, needs to request a lock on the inode every time. There is no caching of locks on inode, either. Every locking request will incur message communication between the initiator and the target.

The table 3 shows the comparison between our proposed iSCSI data sharing scheme and the UNIX locking scheme, based on the trace simulation. In this simulation, we set the physical block size to 8KB, and the block group to 8 blocks per group. There are total 2400 transactions from the 12 client terminals, 200 transactions for each one. During the entire simulation, there are 3039454 accesses to inodes, each of which needs a lock request in UNIX locking scheme. Using our proposed locking scheme over iSCSI, we only observe 387 lock requests for inodes, plus 191 request to upgrade inode locks from share ones to exclusive ones. Similarly, we have observed high saving of lock request for data block accesses. Totally, 3017706 read/write requests for data blocks present during the simulation. The proposed locking scheme only issues 40045 lock requests for block groups, and 9820 requests for lock conversion.

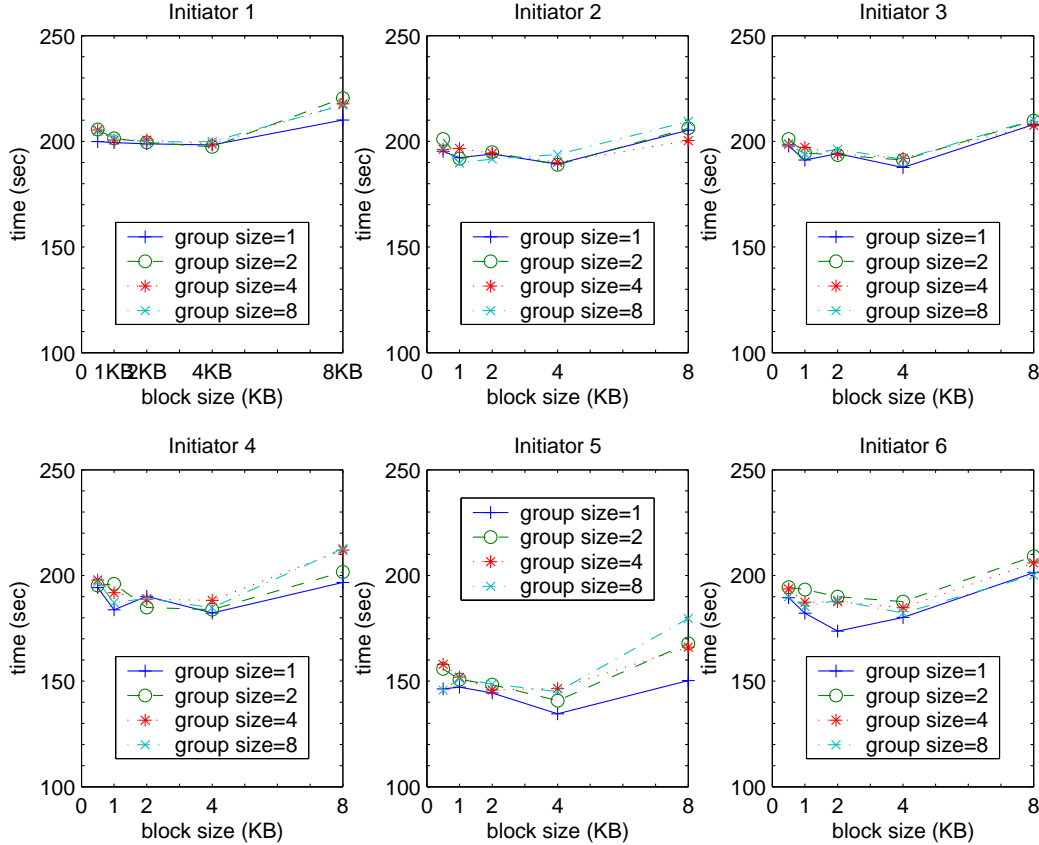


Figure 10: Total transaction time for tpcc in LAN

6.3 Performance Study

Our next experiment is trying to understand the impact of group size on the throughput and deadlocks. As we have discussed in section 4.2, locking on groups of blocks can reduce amount of request message for locks, and avoid wasting storage space due to large block size. The proposed scheme is designed to support transaction file sharing semantics, so the throughput and the deadlocks are concerned. With large block group size, it is more possible that accessing of two different blocks will result in requesting locks for the same block group. For example, suppose the block 1 and block 5 are going to be accessed. If the block group size is 4 blocks, there will be requests for group 1 (including block 1 to 4), and group 2. However, if the group size is increased to 8, it will result in the lock requests for the same block group. Therefore, as the size of block group increase, there is a higher probability that deadlocks will happen, which lead to roll back of half executed transactions, and decrease the throughput of the entire system. In figure 12 and figure 13, we show the results of our experiments. In the experiments, we set the physical block size to 8KB. There are 8 iscsi initiators, and 1 iscsi gateway in this scenario. All 8 iscsi initiators are connected to the iscsi gateway through links with bandwidth 100Mbps and latency 10ms. Since we want to investigate the effect of group size on throughput and deadlocks, we turned of the file-level access control function in the iscsi gateway. By this, the iscsi gateway could allow any iscsi initiator to start the transaction immediately. All these 8 iscsi initiators are accessing the same file, which is of size 1GB, through out the experiments. An iscsi initiator will repeat running transactions sequentially, throughout the 18000 seconds simulation duration. A transaction will access certain percentage of the entire file. Each time, we randomly generate the accessed blocks with uniform distribution. We ran our experiments under two different percentages for the transactions. Figure 12 shows the change of the throughput as the group size increases. The x axial is the base 2 logarithm value of the group size in terms of number of physical blocks. For instance, if the group size is 4 physical blocks, the base 2 logarithm value will be 2. Since we have set the physical block

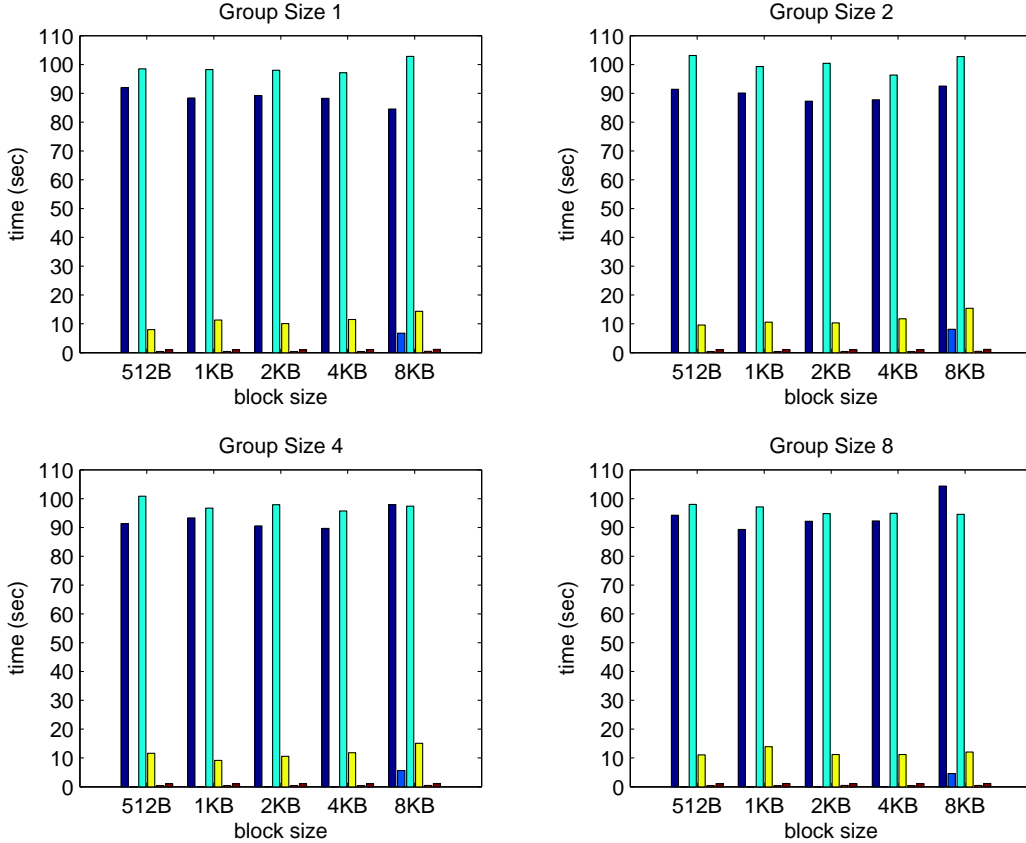


Figure 11: Composition of total transaction time for tpcc in LAN at initiator 1

size to 8KB, the group size in terms of number of bytes would be 32KB. The solid line represents the results when each transaction accesses 2% of the entire 1GB file, which means 20MB or 2560 physical blocks. At the beginning, the throughput increase as we increase the group size. This is due to the decrease in the number of lock requests. According to the property of access locality, if the group size is larger, one lock request could be used by more nearby block accesses, since the proposed scheme caches the locks. However, as the group size increase, the probability of happening of deadlocks increases as well, which is illustrated in figure 13. Therefore, when the group size reaches certain value, the throughput starts decreasing due to too many deadlocks. In our experiment, the peak throughput appears when the group size is 2 blocks/16KB. In both figure 12 and figure 13, the dashed line shows the experimental result when each transaction accesses 4% of the entire file. Compared with the scenarios, where each transaction accesses 2% of the entire file, the variance in throughput is much less. The reason is that deadlock dominates when a larger percentage of the file is accessed.

From above studying, we get the following design guidance in choosing the group size. The block group size is a per file parameter, in that two different files in the same system could be using different group sizes. If the transactions will only access a small part of the entire file, the lock requests will dominate the overhead. In this case, it is better to increase the group size until the deadlocks start to overwhelm. On the other hand, if transactions will access a larger part of the entire file, the deadlock is always dominant. Under such condition, it is wise to set the group size to just 1 physical block, which has the least probability in causing deadlocks. In real implementation, the group size parameter could be either statically assigned, based on the access pattern of the files, or dynamically adjusted by monitoring the happening of deadlocks.

In section 4.4, we mentioned that one advantage of hierarchical locking for data files is that it allows file-level access control. The purpose of file-level access control is to control the happening of deadlocks to alleviate their negative impact on the throughput. We performed our experiments again with the file-level

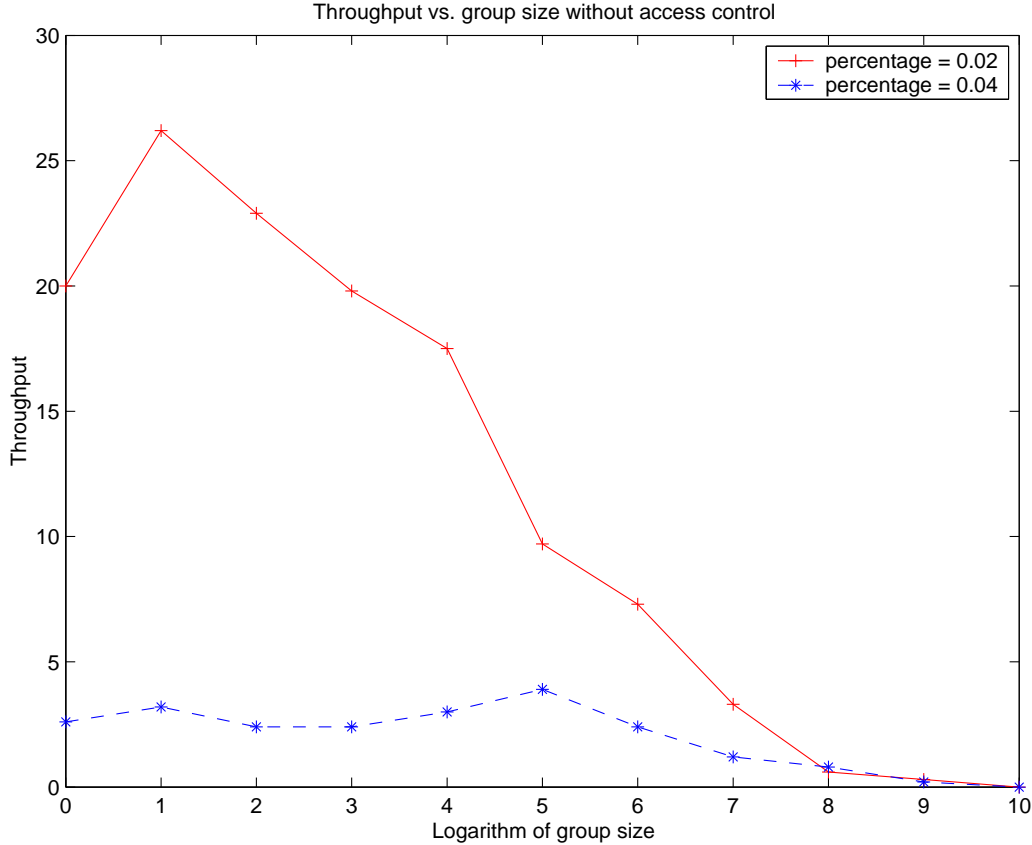


Figure 12: Throughput vs. group size without file-level access control

access control enabled on the iscsi gateway. In the implementation, the iscsi gateway maintains a threshold value, and each transaction from the iscsi initiators provides its estimated access percentage over the entire file. The estimated access percentages of all active transactions, which have WRITE mode on the file, will be added up. The iscsi gateway will not admit a new transaction if that will lead the aggregated value to exceed the threshold. In our experiment setting, each transaction will access the 1GB file with WRITE mode and estimated access percentage of 2%. All other settings are the same as the previous experiments. We varied the threshold from 0.03 to 0.17, increasing 0.02 at each step. This is equivalent to allow 1 to 8 active transactions to be running simultaneously. We present the results in figure 14 and figure 15. When the threshold is 0.03, only one transaction is running at any moment. Of course, there will be no deadlock at all. However, due to overhead in sending commands and responses, the iscsi target is not always busy. By increasing the thresholds to allow more concurrent active transactions, we see the improvement in throughput, when the block group sizes are under 64KB. When the group size is smaller, such improvement is more evident. This is because concurrent transactions could make the iscsi device in high utilization. However, with concurrent transactions, deadlocks start to happen. Figure 15 shows that deadlocks always increase as the threshold increases. With larger group size, more deadlocks happened during the experiments. In figure 14, when the group size is more than 128KB, there is no improvements in throughput at all, as the thresholds increase. This illustrates that large group size will easily lead to deadlocks, which compensates the benefits from concurrent transactions.

7 Related works

Various sharing file systems have been in use for more than fifteen years and various cache consistency schemes have been employed by these file systems. Most cache consistency schemes can be divided into two

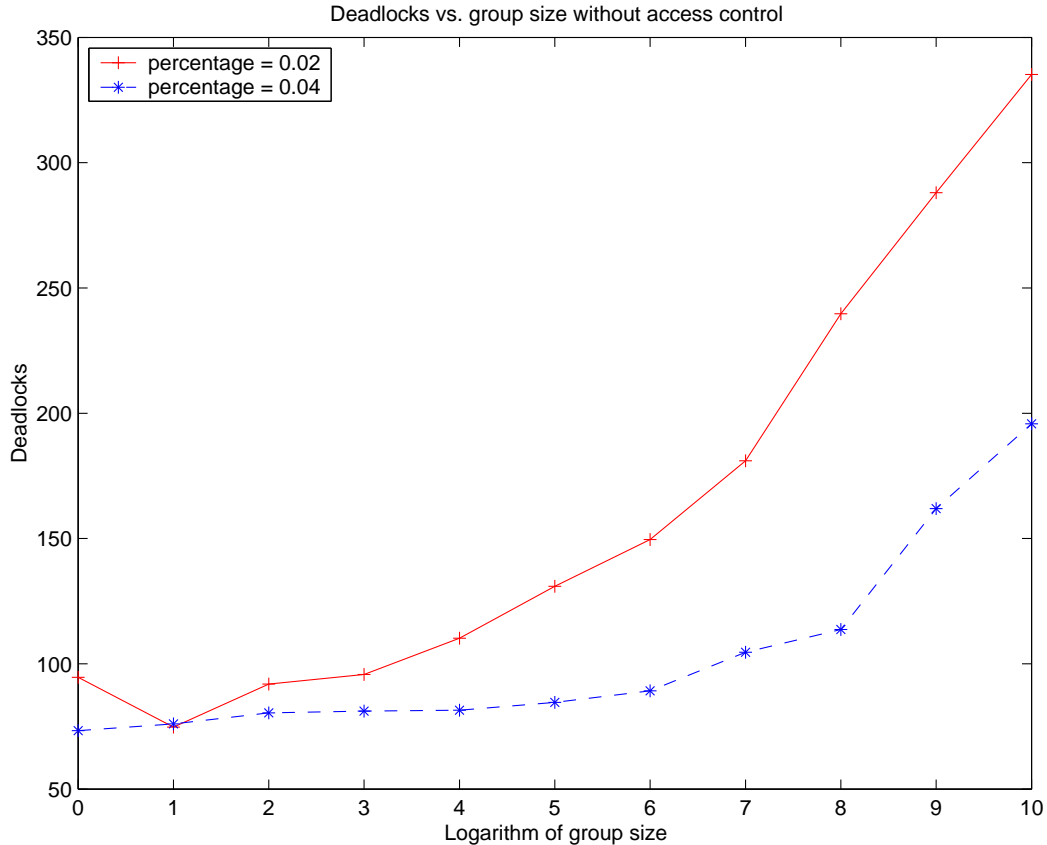


Figure 13: Deadlocks vs. group size without file-level access control

categories:

- Client Initiated approach (as in NFS)
- Server Initiated approach (as in AFS)

IBMs Virtual Shared Disk [11] allows multiple workstations to access disks as if they were connected locally; multiple (possibly distributed) physical disks may be grouped together to form a single virtual disk. However, VSD does not provide caching or locking. Distributed VSD applications have to synchronize via some external mechanism.

Token-based locking and cache consistency mechanism is used in the shared logical disk scheme [12] and also in distributed file systems like Sprite [13], AFS [14], Decorum [15], Frangipani [16] and Echo [17]. While these systems vary in the details of the coherence protocol, particularly in cases of concurrent write-sharing, their basic ideas are the same: clients acquire read and write tokens before accessing file data, and the servers revoke these tokens when other clients request conflicting tokens. This idea of token revocation has been adopted in our design, but the important difference between these systems and iSCSI SAN is the granularity of consistency, e.g. iSCSI SAN is block based.

The idea of leases [18] is adopted from the V operating system of Stanford but it is used only for write locks. Leases have been proposed in the V operating system as a time-based mechanism that provide efficient consistent access to cached data in distributed systems.

xFS [19] seeks to use a central, trusted core of machines as a scalable file server for a larger, less trusted group of file system clients. xFS is also severless like iSCSI SAN but they use a distributed approach, which is more complex and requires cooperating workstations.

Finally a concept similar to semi-preemptible [9] lock that allows clients to cache privileges is used in our design. This idea is incorporated only for read locks in this paper. It results in saving a lot of

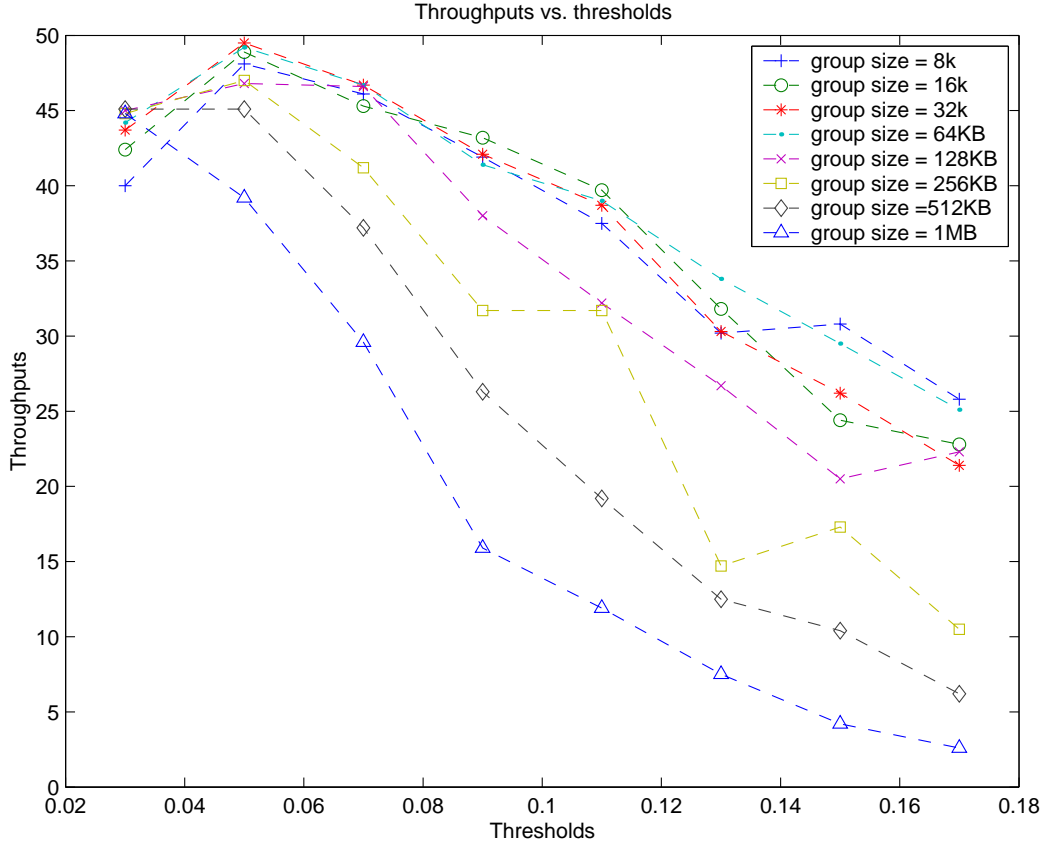


Figure 14: Throughput vs. access control thresholds

unnecessary network traffic, especially for metadata because the percentage of metadata reads is much higher than metadata writes. This observation has been made in a paper dealing with comparison of file system workloads [8].

Also the need to maintain metadata integrity in the event of a system failure (client failure or iSCSI storage gateway failure) has been done in the past and file systems like Log-structured File System (LFS) [20] and Journaling File System (JFS) have been proposed. Also the concept of soft updates [21] deals with the same issue of providing stronger integrity and security guarantees. This concept is not dealt with in detail in this paper, but it should be incorporated in the design in the future.

NFS [5], which is typically on UNIX systems, and CIFS [22], which is on Windows NT systems, are widely used methods to share files among multiple computers. They provide a logically shared view of file systems, allowing remote files to appear as if they were local files and allowing multiple computers to share the files. However, their client initiated cache consistency scheme would not be efficient for metadata sharing in iSCSI SAN, since for each metadata read/write the clients need to contact the server resulting in unnecessary network traffic. In addition, the cache consistency guarantee provided will not be very strong if the checks are delayed.

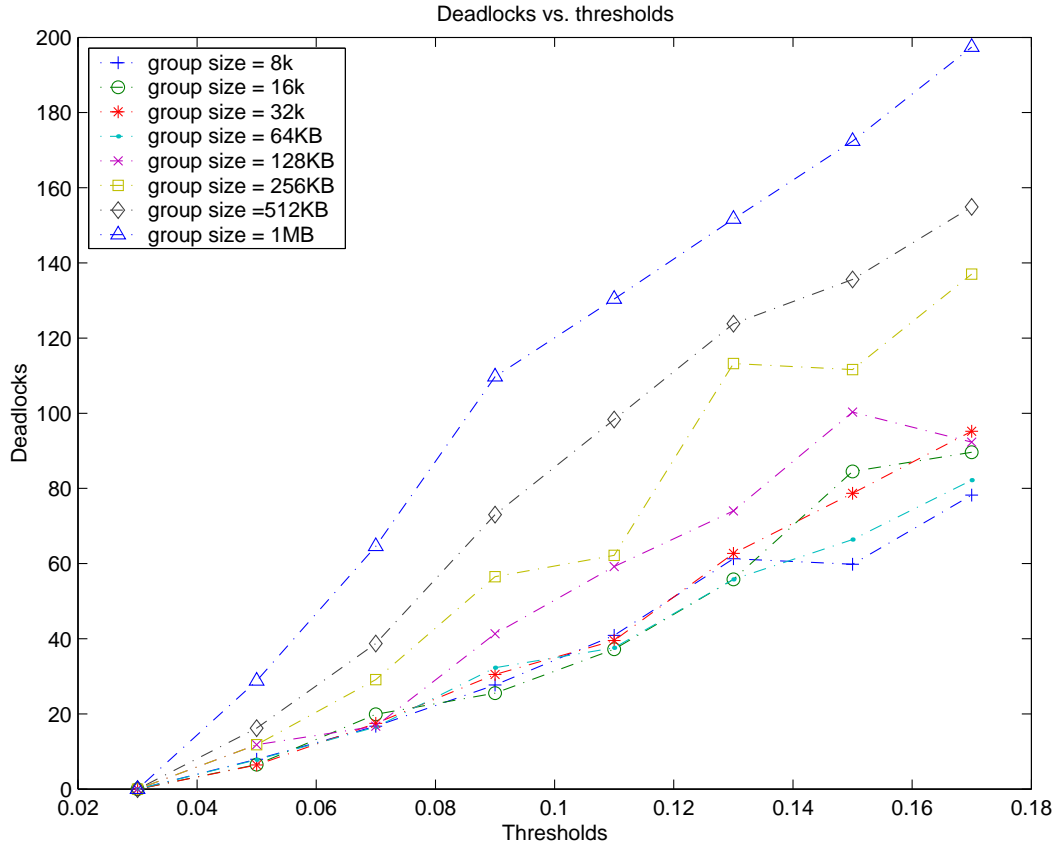


Figure 15: Deadlocks vs. access control thresholds

References

- [1] J. Satran, K. Meth, C. Sapuntzakis, and E. Zeidner, *iSCSI*. IP Storage Working Group, January 2003.
- [2] Cisco Systems, *iSCSI Protocol Concepts and Implementation*.
- [3] Storage Networking Industry Association and SNIA IP Storage Forum, *iSCSI Technical White Paper*.
- [4] A. F. Benner, *Fiber Channel: Gigabit Communications and I/O for Computer Networks*. McGraw-Hill, 1995.
- [5] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon, "Design and implementation of the sun network filesystem," in *In Proceedings of the Summer 1985 USENIX Conference*, (Portland, OR), pp. 119–130, June 1985.
- [6] P. J. Braam and M. J. Callanhan, *Lustre: A SAN File System for Linux*. Stelias Computing Inc.
- [7] M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel, and D. C. Steere, "Coda: A highly available file system for a distributed workstation environment," *IEEE Transactions on Computers*, vol. 39, no. 4, pp. 447–459, 1990.
- [8] D. Roselli, J. Lorch, and T. E. Anderson, "A comparison of file system workloads," in *In Proceedings of USENIX Technical Conference*, (San Diego, California), pp. 41–54, June 2000.
- [9] R. C. Burns, R. M. Rees, and D. D. Long, "Semi-preemptible locks for a distributed file system," in *In proceedings of the 2000 International Performance Computing and Communication Conference (IPCCC)*, (Phoenix, AZ), February 2000.

- [10] S. McCanne and S. Floyd, “The lbnl network simulator,” 1997.
- [11] C. Attanasio and M. Butrico and C. Polyzois and S. Smith and J. Peterson, “Design and implementation of a recoverable virtual shared disk,” IBM Research Report RC 19843, IBM T. J. Watson Research Center, Yorktown Heights, NY, 1994.
- [12] Robert A. Shillner, Edward W. Felteni, “Simplifying distributed file systems using a shared logical disk,” Tech. Rep. TR-524-96, Princeton University CS Department, 1996.
- [13] Michael N. Nelson, Brent B. Welch, John K. Ousterhout, “Caching in the Sprite network file system,” *ACM Transactions on Computer Systems*, vol. 6, no. 1, pp. 134–154, 1988.
- [14] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West, “Scale and performance in a distributed file system,” *ACM Transactions on Computer Systems*, vol. 6, February 1988.
- [15] M. Kazar, B. Leverett, O. Anderson, A. Vasilis, B. Bottos, S. Chutani, C. Everhart, A. Mason, S. Tu, and E. Zayas, “Decorum file system architectural overview,” in *In Proceedings of the Summer USENIX Conference*, (Anaheim, CA), pp. 151–163, June 1990.
- [16] C. A. Thekkath, T. Mann, and E. K. Lee, “Frangipani: A scalable distributed file system,” in *Symposium on Operating Systems Principles*, pp. 224–237, 1997.
- [17] A. D. Birrell, A. Hisgen, C. Jerian, T. Mann, and G. Swart, “The Echo distributed file system,” Tech. Rep. 111, Palo Alto, CA, USA, 10 1993.
- [18] C. Gray and D. Cheriton in *In Proceedings of the 12th ACM Symposium on Operating System Principles*, (Litchfield, AZ), pp. 202–210, December 1989.
- [19] T. Anderson, M. Dahlin, J. Neefe, D. Patterson, D. Roselli, and R. Wang, “Serverless network file systems,” in *In Proceedings of the 15th Symposium on Operating System Principles. ACM*, (Copper Mountain Resort, Colorado), pp. 109–126, December 1995.
- [20] M. Rosenblum and J. K. Ousterhout, “The design and implementation of a log-structured file system,” *ACM Transactions on Computer Systems*, vol. 10, no. 1, pp. 26–52, 1992.
- [21] G. R. Ganger and Y. N. Patt, “Metadata update performance in file systems,” in *Proceedings of the USENIX 1994 Symposium on Operating Systems Design and Implementation*, (Monterey, CA, USA), pp. 49–60, 14–17 1994.
- [22] P. Leach and D. Naik, “A command internet file system (cifs/1.0),” technical report, December 1997.