RC24123 (W0611-189) November 28, 2006
Computer Science

IBM Research Report

Dynamic Policy Disk Caching for Storage Networking

Eric Van Hensbergen
IBM Research Division
Austin Research Laboratory
11501 Burnet Road
Austin, TX 78758

Ming Zhao
University of Florida

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

Dynamic Policy Disk Caching for Storage Networking

Abstract

Storage networking is becoming increasingly important, especially with the emergence of SAN over IP technologies. It plays an important role in resource consolidation and management for various systems, spanning from a large-scale data center to a set of interconnected workstations. However, existing storage systems may not scale well under the load from relatively large number of clients. Local disk caching has the potential to solve this problem by leveraging data locality with client-side storage, but it is lacking for typical storage networks. Addressing to this, this paper presents a general block-level disk cache, named dm-cache, built on top of Linux's device-mapper architecture. It can be transparently plugged into a client for any storage systems, and supports dynamic customization for policy-guided optimizations. Experimental evaluation based on file system benchmarks and a typical application has been conducted to investigate its performance over iSCSI in a typical blade center. The results show that dm-cache can significantly improve the performance and scalability of the storage system by orders of magnitude.

1 Introduction

The desire for increased density and maintainability along with recent advances in networking technology and bandwidth have led to a growing academic and commercial interest in storage networking. Consolidating storage resources for a data center, compute node cluster, or set of workstations eases the administrative tasks of software installation, data backup, and hardware replacement. It produces economies of scale for storage equipment, such as RAID arrays, by sharing it among several systems. Remote storage resources also provide an easy medium for sharing common information and applications across many client machines. The delivery of commercial, blade-based servers [6] [5] [2] adds additional impetus for the introduction of storage networking [16]. The compact form factor of many server blades allows little room for storage, usually allowing for only two laptop-sized disks.

In the research efforts of the 1980s, there was a significant amount of interest in the use of remote disk storage to support not only the long-term storage of user data but also program binaries, libraries and configuration data. For example, the V System [14] created a complete computing system using multiple nodes, some disk-less workstations and some disk-based servers using a novel interconnection protocol, VMTP [15]. The Plan 9 Research Operating System [29] also attached disk-less workstations and disk-less compute servers to a specialized central file server. Leveraging centralized network storage for root file systems in order to simplify systems management has made a come back both in large scale HPC clusters [19] [4], workstation environments [32] [13], and servers [28] [34].

Modern storage networking technologies naturally divide into two categories: distributed file systems, referred to as network-attached storage (NAS) and remote block device access mechanisms, called storage-area networks (SANs). The difference between the two is in the level of the interface that they offer to remote data. NAS offers access to files using a standard distributed file system protocol, such as the Network File System (NFS) [9] or the Common Internet File System (CIFS) [23], while SAN provides block-level access to remote disk resources. Today, these storage networking technologies generally are implemented using standard, layered protocols based on the TCP/IP protocol stack. In particular, with the emergence of SCSI over IP (iSCSI [33]), ATA over Ethernet (AOE [12]) and Fibre Channel over IP [31], the SAN environment is increasingly using TCP/IP, and the distributed file system protocols used by NAS are almost universally based on it.

Caching of remote file systems within local memory and on local disks has long been a component of network-attached storage systems. Sun's cachefs, the AutoCacher [25], Andrew File System [22], Plan 9's cfs [1] and Coda [24] all had the ability to use local disks in order to improve data access efficiency, facilitate disconnected operation and reduce load on centralized network-attached storage. More recently, David Howells' FS-Cache [20] provided a generalized local disk caching facility for the Linux kernel.

What has been lacking is a similar local-disk caching facility for storage networks such as fibre channel, iSCSI, and AoE. The exclusive access nature of SAN technologies reduce or eliminate the need to maintain coherence, further increasing the performance benefits of local caching. We have attempted to address this gap with a generalized block cache facility named dm-cache, built on top of Linux's device-mapper [3] block device virtualization framework. It can be transparently plugged into the clients of different storage systems, and supports dynamic customization for policy-guided optimizations. Extensive experiments have also been conducted to investigate its performance over iSCSI in a blade center. The results show that by leveraging data locality with dm-cache, the performance of the system can be improved by orders of magnitude, especially when the number of clients is relatively large.

The rest of this paper is organized as follows. Section 2 introduces the design and implementation of dm-cache. Then two usage examples are discussed in Section 3. Section 4 presents the experimental evaluation. Section 5 examines the related work, and Section 6 concludes the paper.

2 Design and Implementation

2.1 Background

Our approach leverages the block device virtualization provided by the device-mapper framework. Device-mapper is a block device mapping facility available in Linux kernel since the 2.6 releases. It is a component required by the LVM2 (Logical Volume Management toolset) to support the mapping between logical volumes and physical storage devices. Furthermore, it also provides a generic framework for block device virtualization. A virtual block device is typically created through device-mapper's userspace library, which communicates with its kernel component via an ioctl interface. Block I/O operations issued on the virtual device are forwarded by device-mapper to an I/O handling target, which is mainly responsible for mapping the concerned virtual blocks to the actual blocks of the underlying block device. For instance, a linear mapping target (dm-linear) is used by LVM2.

A mapping table is used by device-mapper to set up the mapping and the target. For example, the following table maps the virtual device's block address range, which starts at sector 0 and has a length of 65536 sectors (512 bytes per sector), to the linear target. The parameters after the target name are target-specific, and they are here to specify that this virtual block segment should be mapped onto /dev/sdb, starting from sector 0.

```
echo 0 65536 linear /dev/sdb 0
```

A growing list of kernel modules has been developed for pluggable device-mapper targets, such as stripe, mirror, snapshot and raid. Some targets do more than just mapping block addresses, e.g. an encryption target (dm-crypt) provides transparent encryption of blocks; dm-raid implements software RAID support. The logic and management of the proposed block-level disk cache are also developed as a new target, called dm-cache, based on the device-mapper framework. The rest of this section discusses in details the design and implementation of dm-cache.

2.2 Architecture

By leveraging the device virtualization provided by device-mapper, we can conveniently and transparently intercept kernel file systems' block I/O requests, and perform the necessary cache operations for them. A disk cache is constructed by the creation of a virtual block device through device-mapper, and a mapping between the original device which is usually accessed through a storage network, and the local device that is used to store the cache contents. The virtual device can be used in the same way as the original device, and then the block I/Os that are issued onto the virtual device will be handled by dm-cache. Figure 1 illustrates the architecture of a storage system with the use of dm-cache.

Dm-cache operates at the granularity of block I/Os. The following operations are typically involved in a remote data access when a disk cache is used (Figure 1). (1) A user application accesses a file located in the file system mounted from the virtual block device. (2) It triggers the kernel file system's block I/O onto the virtual block device. The request is handled by device-mapper which then passes it onto the dm-cache target. Dm-cache checks whether the concerned sectors are already stored in the cache device or not. (3) If yes, then the request is performed on the local cache. (4) Otherwise, it is forwarded to the remote device accessed through the storage network protocol, e.g. FCP, iSCSI and AoE, etc. The results returned from the remote device are forwarded back to the application and also inserted into the disk cache. Then next time when these sectors are requested again, they can be directly satisfied from the cache without contacting the remote storage. Note that if the block I/O is a

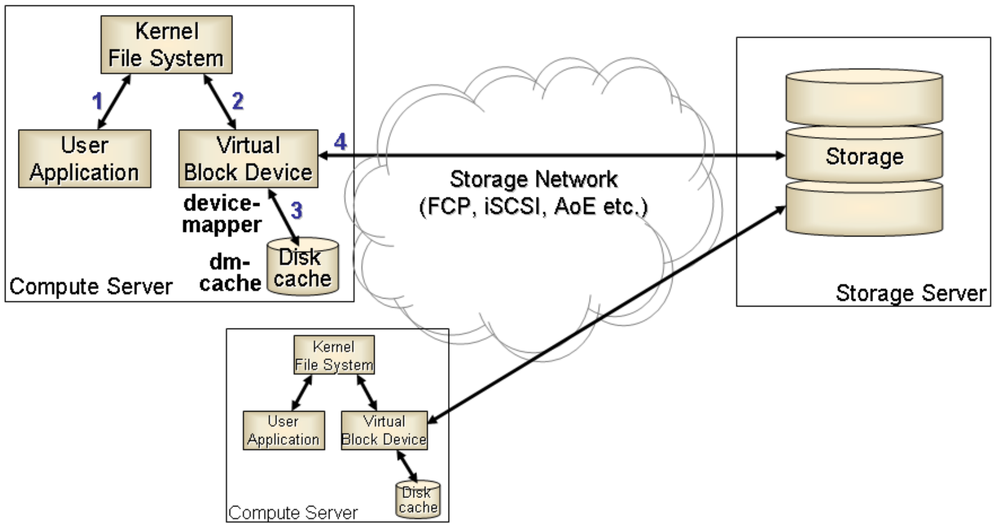


Figure 1: The architecture of a storage system with the use of dm-cache disk caching.

write operation, the cache logic is slightly more complicated, which will be explained later in Section 2.5.

In the above discussion the first two steps are the same as in any regular I/O operation, which means that the cache is completely transparent to both user applications and kernel file systems. This is an important merit of our approach, which is inherited from the device-mapper's block device virtualization. The existence and operations of a disk cache are encapsulated within device-mapper and dm-cache, and to the outside the presented virtual block device works just like a regular block device.

On the other hand, dm-cache is also completely agnostic of the kernel file systems, which is a significant advantage of this design over other disk caching approaches. The simplicity and stableness of the Linux generic block device layer interface, which dm-cache works with, has greatly facilitated a robust implementation and maintenance of the dm-cache code. Working at the block level also lets dm-cache inherently support all storage network protocols, and therefore it can seamlessly fit into any storage systems.

2.3 Structure

A disk cache consists of two parts, the actual data cached from the original block device, and the metadata associated with the cache. Data are organized as blocks and stored on the cache device. Data blocks are typically in the size of multiple sectors, and organized into sets, as in traditional set-associative cache designs. The block size and cache associativity are both configurable per disk cache (see Section 2.5 for details). A hash table based algorithm is currently used to map the blocks from the original block device to the cache frames.

When a block is inserted into a cache set, the algorithm firstly tries to find an invalid block, i.e. an empty frame, to use. If cache replacement turns out to be necessary, the eviction favors clean blocks over the dirty blocks which store the delayed write operations. Then within the same type of eviction candidates, the LRU (Least Recently Used) policy is used to pick the best one for the replacement. The cache algorithm is implemented in a way that the above policies can be realized with only one scan of the entire set.

Modern disk drivers and devices often use buffers to coalesce adjacent small accesses in order to improve the utilization of raw disk bandwidth. A naive hash based mapping may cause adjacent blocks from the original disk to be mapped to very distinct locations in the cache disk, and thus lead to poor performance. Addressing to this problem, the cache mapping algorithm is optimized to have a certain number of consecutive blocks mapped into consecutive frames in the cache, so that the cache disk's bandwidth can be effectively utilized.

Compared to other complex, full-associative caching algorithms, our hash-table based set-associative scheme is uncomplicated and thus potentially requires less CPU processing power, and yet it is very flexible for customization. Further, its effectiveness is also proved by the experimental evaluation, as presented in Section 4.

A disk cache's metadata consists of its parameters, such as the capacity, block size, associativity etc., and the dynamic status of each cache block, including the mapping, i.e. the corresponding offset of the cached data in the original block device, and the state. As mentioned above, a cache block can be in three different states, invalid, clean or dirty. In every disk cache, there is a segment at the end of the cache device that is reserved for storing the cache metadata. However, in order to achieve the best performance, the metadata is always kept in memory and only synchronized with the disk when it is necessary.

There are two issues with these approaches. The first one is the limited kernel memory for storing the metadata. When a cache's capacity is large and its block size is small, the amount of metadata can be considerably large. To solve this problem without reducing the cache size, the block size needs to be increased so that the cache can operate at a larger granularity of disk sectors, which, as a side effect, will cause the cache to use more aggressive prefetching. A memory usage threshold can be set in dm-cache, and when the available quota is not enough to create a cache, an error will be reported to suggest the user to use a larger block size or a smaller cache size. The other issue is about reliability when the in-memory metadata are not consistent with the disk cache contents, which will be discussed in Section 2.6.

2.4 Deployment

Dm-cache is implemented as a loadable kernel module to the Linux 2.6 releases. Hence, its deployment does not require any kernel modifications, given that device-mapper is already deployed in the mainstream Linux kernels by default. This is very important in that dm-cache can be conveniently integrated with the existing Linux compute servers. A disk cache can be created using the device-mapper userspace tool, dmsetup. For example, the following command line creates a disk cache with the desired parameters for a block device.

```
  echo 0 131072 cache /dev/sdb
/dev/sda6 0 8 65536 256 | dmsetup
create cache1
```

It uses dmsetup to create a virtual device called cache1, which appears on the compute server as /dev/mapper/cache1, and passes the mapping table, which includes the parameters after the echo command, to the dmsetup program for setting up the cache. The first two parameters correspond to the size of the original block device. The third parameter tells device-mapper to use our new cache target for the mapping of this virtual device. The following parameters are specific to dm-cache. They specify that the device to be cached is /dev/sdb and the local device used to store the cache contents is /dev/sda6. The other parameters customize the configuration of the cache, which starts at sector 0, operates at the block size of 8 sectors (4KB), has a capacity of storing 64K blocks, and is 256-way associative.

We have also developed another program for the management of disk caches, based on the device-mapper userspace library. It has a much more user-friendly interface, and can automatically find out the correct size of the original device, which is a necessary parameter for the cache setup. Furthermore, this program provides a central management for all the disk caches on the same compute server, which is discussed in the following subsection.

2.5 Dynamic Policy

A session-based semantics is employed to customize and manage disk caches. A session is typically associated with the deployment and execution of an application. A disk cache is created for a session, and removed after the session completes. For each session, its disk cache can be customized according to the application's requirements and characteristics, and the resource utilization policies.

As discussed before, a disk cache's parameters, including capacity, block size and associativity can be configured as desired. For example, a cache's capacity can be configured based on an estimation of the application's working dataset size, or its business value. A cache's block size is usually configured to be 4KB, because kernel file systems operate at the granularity of pages, which are in this size. However, it can also be customized according to the application's data access pattern. For example, if substantial sequential data accesses are expected, then the cache block size can be configured larger to take advantage of more aggressive prefetching.

Another important cache customization is on the choice of write policy: write through or write back. With the write-through policy, a write block I/O is performed on the remote storage as well as in the cache. The two write operations can be conducted simultaneously and both asynchronously, and dm-cache does not need to wait for the cache write to finish before it acknowledges the original request's completion. So the cache's performance is not affected by the use of this policy. More importantly, write-through guarantees the consistency of data on the storage server, which is very important in some scenarios, e.g. where data loss is very expensive and the client-side storage device is much less reliable than the servers.

When the write-back policy is used, a write request is only performed in the cache and its submission to the storage server is delayed. Therefore, it can alleviate the load of the storage network and server, and also improve the application's performance, especially when the storage system is under a considerable load from multiple clients, as demonstrated by our experimental evaluation.

The delayed writes can be submitted at the end of the session, after the application completes its execution, in which case, the application can potentially achieve the best performance. However, the flush of large volume of dirty cache blocks may hurt the performance of other active sessions. Two approaches can be used to solve this problem. The first one is developed by giving dm-cache the ability of self-throttling bandwidth utilization. As the flushing phase starts, dm-cache increases the bandwidth usage for writing back gradually. It doubles the number of simultaneously submitted blocks every round, and also monitors the average response time for the requests of each round. If a significant deterioration of the responsiveness is detected, it means that the storage system is probably saturated, and thus dm-cache reduces its bandwidth usage by halving the block submission rate.

Another approach leverages middleware for more precise resource utilization monitoring and scheduling, in which the middleware is responsible for tracking the storage server's load and controlling the client to flush the dirty cache blocks when the server is relatively idle. It will be discussed shortly in Section 3 that middleware can play an important role in the effective usage of client-side disk caches. The cache management program discussed above is part of this middleware, acting as the control point on a compute server. For example, it records the setup of each local disk cache on the compute server, and prevents two caches on the same disk volume from overlapping each other. It also generates a unique string for each cache to use as its identifier.

Finally, it is worth to be noted that a disk cache's customization can also be dynamically adapted according to the policy changes. For example, a resource utilization policy may decide to shrink an existing session's cache in order to save space for a more important session. A performance requirement may choose to switch a cache's write policy to write-back so that the application's execution deadline can be satisfied. Through the management program, an existing cache can be signaled to reload its configuration parameters and perform the desired changes. However, such a process may take a considerable time to complete, in which case the cache can be suspended temporarily for carrying out this adaptation.

2.6 Reliability and Failure Handling

Dm-cache disk caching is inherently reliable because the data are always stored on persistent storage. Further, when the storage server is crashed due to failure or not responsive because of overloading, a warm disk cache can even help the client to continue operating till the server is recovered. However, if a cache operation to the cache device does raise an I/O error, then this error will be hidden by dm-cache from the kernel file system and user application. The original request that triggers this cache I/O will be forwarded to the storage server, and the error will be logged for the administrator to examine. Specifically, when a read to a cached block or a write for a write-back block fails, dm-cache will recover this error by completing the request on the remote storage device instead.

A cache's metadata are always maintained in memory and kept consistent with the data blocks on disk. If a compute server shuts down gracefully, dm-cache can also safely flush the metadata to disk so that when the system is up again, the metadata can be reloaded from disk and the cached data blocks can be reused. However, if the compute server is recovered from a crash, then the metadata retrieved from disk may not be in a consistent state with the cached data. To solve this problem, the cache can be configured in three different ways in order to recover from such a client-side failure.

The first solution is to invalidate the entire cache and discard all the cache contents when the system is recovered. In this configuration, the cache metadata in memory does not need to be synchronized with the cache disk, but write-back caching cannot be used otherwise data may be lost when a failure happens. The disadvantage of this approach is that cached data are wasted after the recovery and the performance may be hurt without using write-back. However, because it avoids the overhead from synchronizing metadata, it can also be efficient given that failures are rare and write operations are much more infrequent than reads. In addition, it is also suitable when the reliability of the compute server's local disks are not trusted.

The second solution synchronizes the in-memory metadata with the disk whenever the metadata is changed, i.e. when a cache insertion, invalidation or write-back has taken place. This configuration has the highest synchronization overhead compared the others, but it can be used when the compute server is not in a stable operating state. The third solution is a trade-off between the first two, in which only the metadata of dirty cache blocks are always kept consistent on disk. In this case, only the insertion or submission of a write-delayed block triggers the synchronization between memory and disk. This approach can guarantee the data integrity and also achieve a very good performance. Therefore, it is used as the default configuration in most scenarios.

3 Usage Examples

3.1 Dynamic Application Environment Instantiation

An Application Environment (AE) is an encapsulation of an execution environment, including the O/S, libraries and other software installations that are customized for a particular application. This is a concept that is often used by data centers, where multiple AEs are hosted on dynamically allocated pool of resources to provide differ-

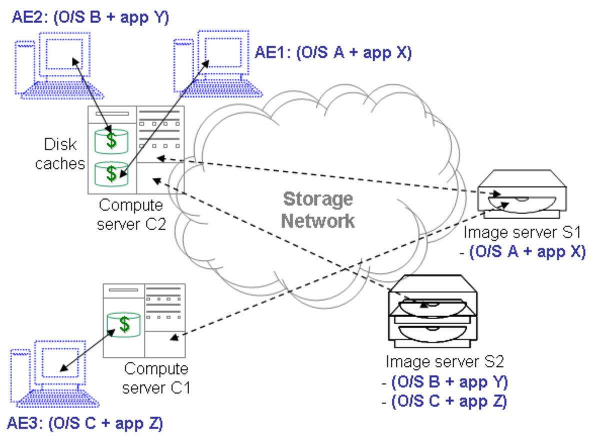


Figure 2: Dynamic Application Environment Instantiation.

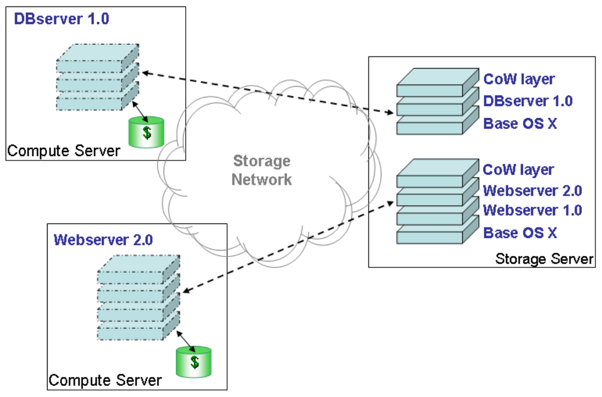


Figure 3: Cluster Software Provisioning.

ent application services (Figure 2). The AEs are stored as images on the central image servers, and dynamically deployed and destroyed on the compute servers on demand. An AE can be directly instantiated on a dedicated physical node ([26]), or multiple AEs can be instantiated within individual isolated virtual machines which share the underlying physical resources ([35]).

Such a scenario can be realized on top of a storage network to support high-performance image accesses. Local disk caches can be deployed on the compute servers to further speed up the instantiation of AEs. A disk cache is bound to a specific AE with a unique identifier, as mentioned in Section 2.5, that describes the AE. However, it can be persistent on the compute server across the AE's lifecycles. Then when the middleware decides to deploy an AE, it can match the AE with the available client-side disk caches, and choose the matched one to use, so that the cached data can be leveraged to achieve instantaneous instantiation.

3.2 Cluster Software Provisioning

The system management of a computing infrastructure is becoming increasingly complex. For example, a small or medium-sized firm typically has a cluster of servers running software suites for internal use as well as services for customers. The provisioning of a software component is often a long and manual process, entailing the installation and configuration of the base O/S and the necessary software. Further, it has to be repeated whenever a reprovision is required.

Such a process can be greatly simplified and expedited by employing role-based software management and storage virtualization technologies ([28]). A key software component, such as a base O/S, a web server and a database server, can be designated a role and stored in a storage layer on the storage server. The later-on upgrades to the software component will lead to different versions of the same role and be encapsulated in individual Copy-on-Write (CoW) layers. In this way, the provisioning of a software component only involves giving the compute server accesses to the corresponding role's storage layers (Figure 3). A local disk cache can be established along with the provision in order to relieve the storage server's load and improve the software's performance. The cache is also bound to the software component with an identifier that describes the provision. Then when a reprovision occurs to the compute server, the disk cache can be checked against the new provision in order to decide whether it can be reused or should be invalidated.

4 Experimental Evaluation

4.1 Setup

This section evaluates the performance of the proposed disk cache for a typical storage network. Experiments were conducted in a blade center, where 1 of the blades was dedicated as the storage server, and 8 others were used as compute servers to run the applications. Each blade has two dual-core 2.4GHz Opetron processors with 2GB of memory and a 73GB 10K RPM SCSI disk exhibiting an average seek time of 4.1ms, and runs Ubuntu 6.06 with 2.6.17.7 kernel. The blades are interconnected via Gigabit Ethernet, and iSCSI is used as the storage network protocol. The storage server uses iSCSI enterprise target 0.4.13 [8] and the compute servers use Open-iSCSI 1.0-485 [7] as the initiators. Each compute server is allocated an independent 8GB disk volume on the storage server, and accesses the storage data through a local ext3 file system with the default parameters.

In this particular setup, each compute server's local disk has an O/S installation, which is used to support the applications as well as dm-cache. However, this is not always necessary since the servers can be network-booted (e.g. through NFS root), in which case the client-side disks can be fully utilized for caching. For all the experiments, the disk cache on each compute server was configured with 8GB of capacity, 1024-way associativity and write-back policy, and used 4KB block size unless otherwise noted.

Three benchmarks were selected for this evaluation, including two typical file system benchmarks, IOzone and PostMark, and a typical application, kernel compilation. In order to investigate the scalability of the storage system, the benchmarks were executed with various number of compute servers. When multiple servers were considered, each of them executed the same benchmark concurrently and accessed the storage server in parallel. In this case, the results averaged across the involved servers are reported. In addition, every test was started with cold memory buffer and disk cache by unmounting the file system and flushing the disk cache beforehand.

The rest of this section presents the results and analysis from the experiments with these benchmarks. We focus on the performance comparison between two cases: iSCSI with the use of dm-cache (labeled as **dm-cache** in the results) and plain iSCSI without disk caching (labeled as **plain iSCSI** in the results).

4.2 IOzone

IOzone [27] is a commonly used benchmark that analyzes the file system performance for a given computer platform. It typically measures the throughput for read and write operations on a large file with a variety of access patterns. In our experiment it was used to investigate the performance of dm-cache for large volume of sequential reads and writes. Two different IOzone tests were considered: **read/reread**, in which a 4GB input file was sequentially read and then reread once; **write/rewrite**, in which the program sequentially wrote and then rewrote a 4GB file once. Note that because the dataset is larger than the available memory on a compute server, it is guaranteed that when the file is reread it cannot be completely served from the server's memory buffer so that we can have a realistic measurement of the storage system. In addition, a block size of 256KB is used by dm-cache to take advantage of more aggressive prefetching.

4.2.1 Read/Reread

Figure 4(a) shows the throughputs for the read and reread phases as the number of clients, the compute servers, scales from 1 to 8. The read phase started with cold caches, which were filled up as the input file being read by IOzone. The overhead from writing blocks into cache is high compared to the speed of reading blocks across the iSCSI network. Therefore, the throughput of this phase is hurt by the use of dm-cache. However, as the storage server gets loaded up from more clients, the throughput of iSCSI drops substantially and so does the difference between dm-cache's and plain iSCSI's throughput.

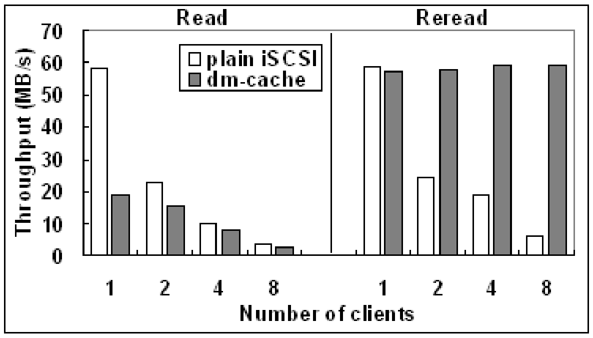
Once the caches become warm, the leverage of data locality helps to improve the performance, which is demonstrated by the reread phase. However, the use of memory buffer is inadequate due to its limited capacity and the performance of iSCSI still drops quickly as more clients access the storage server at the same time. In contrast, the use of dm-cache can completely satisfy the clients' data accesses from local disks, and thus the reread phase's throughput is not impacted by the number of clients at all. As the result, the throughput of the dm-cache case for 8 clients is higher than plain iSCSI by more than 9 folds.

The total runtime of the IOzone read/reread test is illustrated in Figure 4(b) for the different cases. In the dm-cache case, the benchmark was executed consecutively twice, while the second run starts with cold memory buffer but warm disk cache. The results further demonstrate that the availability of warm disk cache can greatly improve the scalability of the storage system. The dm-cache case's speedup over plain iSCSI grows to more than 13 folds when 8 clients are in use.

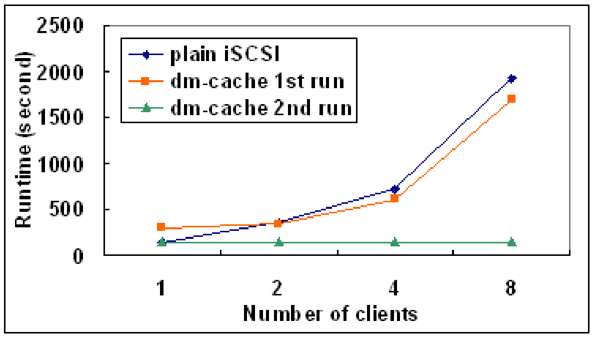
4.2.2 Write/Rewrite

The second IOzone experiment used the write/rewrite test. The write phase also started with cold caches, but different from the read phase of the previous test, the writing of cache blocks is more efficient than the writing of data over iSCSI, and hence the dm-cache case has higher throughput than plain iSCSI (see Figure 5(a)). Further, as more clients writing concurrently, the storage server gets saturated with plain iSCSI and its throughput drops significantly. But in the dm-cache case the throughput remains the same regardless of the number of clients since each client is writing to its own local disk. Similar trend can also be observed in the rewrite phase.

Consequently, the runtime of the benchmark is greatly improved when dm-cache is used, and at the point where 8 clients are used, the speedup is about 8 folds (Figure 5(b)). Note that the time needed for a client to write back its locally cached dirty blocks is not counted into the runtime, since the write-back happened after the benchmark's execution was completed – the computing resources allocated to this task were released and its output data were safely stored. In addition, the dirty data can be submitted when the storage server is relatively idle, as discussed in Section 2.5, so that its impact on the perfor-

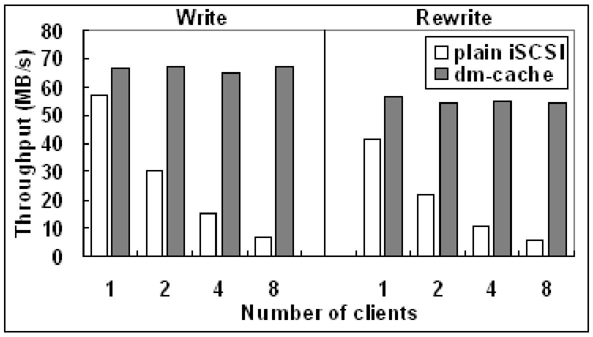


(a) Throughput

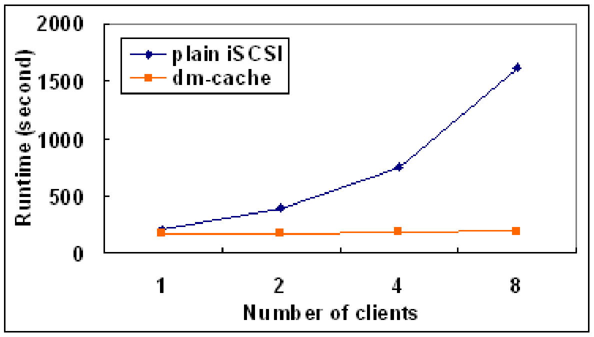


(b) Runtime

Figure 4: The throughput and runtime of IOzone read/reread test.



(a) Throughput



(b) Runtime

Figure 5: The throughput and runtime of IOzone write/rewrite test.

mance of other tasks can be minimized.

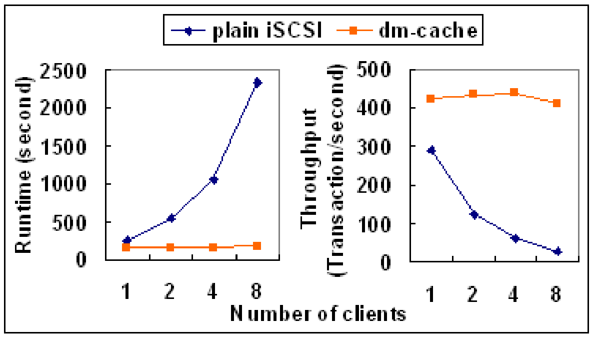
4.3 PostMark

PostMark [21] is another popular file system benchmark, but has very different data access pattern than IOzone. It simulates the workloads from emails, news and web commence applications, by performing a variety of file operations on a large number of small files. The benchmark starts with the creation of an initial pool of files, then issues a number of transactions, including create, delete, read and append, on the pool, and finally removes all the files. In contrast to the uniform, sequential data accesses with the IOzone read and write tests, the storage is randomly accessed with a mixed sequence of operations during the execution of PostMark.

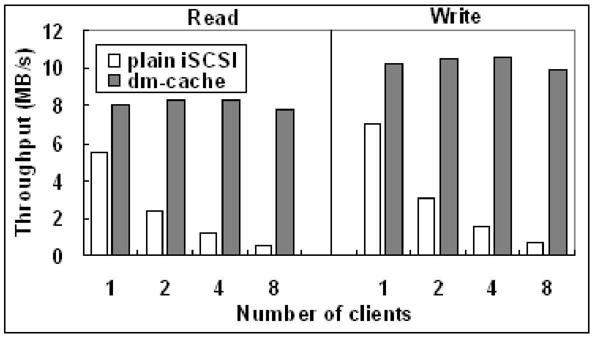
In our experiment, the initial number of files is 8K and the number of transactions is 64K, where the files range between 1KB and 64KB in size, and the transactions are equally distributed between create and delete, and between read and append. With this configuration, the entire execution of PostMark involves 1.3GB of reads and 1.7GB of writes on the storage device. But because the data being read are generated by the program's earlier write operations and the available memory on a compute server is larger than the size of the reads, the operations that are sent out to the storage network are eventually mostly writes.

The total runtime and transaction rate of PostMark's execution are illustrated in Figure 6(a) for the plain iSCSI and dm-cache cases with various numbers of clients. They both show that the use of disk caching achieves much better performance as well as scalability than plain iSCSI. With only one client running PostMark, the performance of iSCSI is improved by 1.5 times with dm-cache. As the number of clients increases, iSCSI's performance drops dramatically, while the disk caches help to prevent the performance degradation. When 8 clients execute the benchmark concurrently, the speedup of dm-cache versus plain iSCSI is already about 15 folds.

Figure 6(b) further breaks down the advantage of using disk caching by comparing the throughputs of PostMark's read and write operations. Similar observations can also be made based on these data.



(a) Runtime and throughput



(b) Throughput of read and write operations

Figure 6: The runtime and throughputs of PostMark.

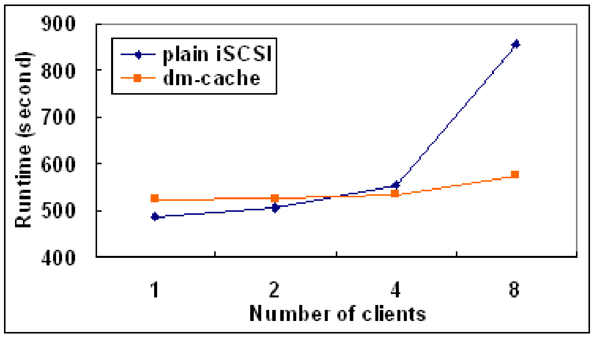


Figure 7: The runtime of kernel compilation.

Table 1: I/O Volume (GB)

	plain iSCSI		dm-cache		difference	
	read	write	read	write	read	write
IOzone read	62.89		32.30		30.59	
IOzone write		64.07		32.07		32.01
PostMark		15.41		5.25		10.16
Kernel compilation	1.50	2.83	1.40	2.60	0.17	0.26

Table 2: Storage Server CPU Consumption Ratio

	IOzone read	IOzone write	PostMark	Kernel compilation
plain iSCSI : dm-cache	2.98 : 1	2.11 : 1	3.03 : 1	1.14 : 1

4.4 Kernel Compilation

Kernel compilation represents the typical file system usage in a software development environment. Very different from the above two file system benchmarks, it is a rather CPU intensive real-world application. The kernel used in the experiment is Linux 2.6.17.7, and the compilation generates more than 5 thousands of object files which involves hundreds bytes of reads and writes on the storage device. To leverage the available CPUs on the compute servers, the compilation was conducted with 4 parallel jobs.

Figure 7 shows the total runtime of the kernel compilation on iSCSI with and without disk caching. Because this task is much less I/O intensive than IOzone and PostMark, the benefit of using disk caches is not that obvious when the number of clients is small. With only one client compiling the kernel, the use of dm-cache introduces a 7% overhead relative to plain iSCSI. With 4 clients, the dm-cache case is only faster than plain iSCSI by 4%. However, when 8 compilations are executed by the compute servers concurrently, their average runtime on plain iSCSI rises up significantly. In contrast, the runtime in the dm-cache case only increases moderately, and consequently it is 50% faster than the case without disk caching. These results reveal that even for a not I/O intensive task, the storage system still has serious scalability problems. Nonetheless, the use of dm-cache can very effectively resolve these problems by taking advantage of the client-side storage and the data access locality.

4.5 Storage Server Utilization

Finally, we conclude this section with another group of data, which explain the scalability of plain iSCSI and iSCSI with dm-cache by comparing their utilization of the storage server as the aforementioned benchmarks are executed by the 8 clients concurrently. Table 1 lists the volume of I/O that is served by the storage server during the execution of the benchmarks (the write-back phases are included). The data from the file system benchmarks, IOzone and Postmark, both show that a substantial amount of I/O is saved on the storage server through the use of client-side disk caching. The I/O saved during the kernel compilation is only hundreds of megabytes per run. However, considering that it represents the typical applications that are frequently running on the compute servers, e.g. as in a data center, the save of the storage servers I/O load will still be considerable.

Serving the iSCSI I/O requests consumes not only the storage server's I/O bandwidth, but also its CPU cycles, which are spent as the requests traverse the kernel layers of networking, iSCSI server and storage device driver. Table 2 compares the CPU consumption between plain iSCSI and dm-cache for the various benchmarks (the write-back phases are also included). It can be seen that an order of magnitude of CPU processing power is also saved on the storage server when dm-cache is used with the storage network.

It is understood that some of the scalability issues exhibited by this iSCSI-based storage system can be attributed to the fact that a single disk is used to serve the concurrent client accesses. However, as demonstrated by the above experiments, the performance bottlenecks appear even for a small number of clients. Therefore, it is reasonable to believe that the same issues will still persist for a better-equipped and more load-balanced storage system, when the number of clients is relatively lager than the current setup.

5 Related Work

Caching is a classic idea that has been very successfully employed by many different types of computer systems. It improves a system's performance by exploiting temporal and spatial locality of references and providing high-bandwidth, low-latency access to cached data. Naturally, caching has also been implemented in various distribute systems in order to hide network latency. Most distributed file systems leverage the client-side memory for caching data and metadata. For example, the NFS protocol allows the results of various NFS remote procedure calls to be cached in a client's memory [9].

However, memory caching is often not sufficient due to its limited capacity and non-persistent nature. Therefore, disk caching has been proposed to further take advantage of data locality. A disk cache can complement the memory buffer and form an effective cache hierarchy. There are several distributed file system solutions that also exploit the advantages of disk caching. Sun's cachefs, the AutoCacher [25], AFS [22], Plan 9's cfs [1] and Coda [24] all have the ability to utilize local disks to improve the efficiency of remote data access, facilitate disconnected operation and reduce load on centralized network-attached storage. More recently, a generalized filesystem caching facility, FS-Cache [20], has been provided for the Linux kernel.

In the context of wide-area/Grid systems, the use of disk caching becomes especially important because WAN typically has much higher latency and lower bandwidth. In such systems middleware is often involved in the cache management. Globus GASS [11] caches the files that are used by a Grid job on the execution host in order to speed up the subsequent executions. BAD-FS [10] improves the performance of batch workloads by using cache volumes to hold read-only data and scratch volumes to buffer local modifications, and also exposing the control of volumes to the scheduler. GVFS [35] customizes disk caching based on application-tailored knowledge, which has been employed to support the instantiation of virtual machines across Grids. Collective [13] is a cache-based management system for delivering virtual machine based appliances.

Compared to these filesystem-level disk caching solutions, our approach is a general block-level disk cache that inherently supports all file system and storage network protocols, and can be deployed in storage systems spanning from SAN to WAN. Furthermore, there are several advantages of building a general disk cache at the block-level than at the filesystem-level. Firstly, the latter approach has to deal with the various complex and often changing interfaces of DFSs. For example, NFS V3 has 21 different procedure calls, while NFS V4 uses the compound procedure call which can combine 37 different operations. In contrast, dm-cache works on the generic block layer interface, which is very simple and stable, and only needs to handle block read and write.

Secondly, in order to intercept a DFS's I/O requests and implement the caching functionality, it is necessary to either modify the DFS, which hurts the applicability, or use a loopback server to proxy the requests, which incurs performance overhead. These restrictions do not apply to dm-cache, since it is based on the kernel's native block device virtualization. In addition, using file system to cache data often has the problem of double buffering (the same data are cached in both the DFS's pages and the pages of the cache's file system), which also does not happen in block-level disk caching. Finally, dm-cache is based on storage network protocols, which typically have much higher performance and scalability than the DFS protocols which filesystem-level disk caches rely on [30].

Although disk caching is typically lacking for existing storage networks, there are several related solutions that have also recognized its importance. STICS [18] introduces a SCSI-to-IP appliance that encapsulates the functionality of protocol conversion and caching. iCache [17] is a iSCSI driver that implements local disk caching. Compared to them, dm-cache is a more generalized disk cache that can be transparently plugged into the clients of different storage networks. In addition, we have demonstrated the scalability of our approach with experiments on a relatively large number of clients, which is lacking from those two papers.

6 Conclusions and Future Work

Storage networking is becoming increasingly important in resource consolidation and system management. However, existing storage systems may not scale well under the load from relatively large number of clients. Addressing to this problem, we have developed a general block-level local disk cache, named dm-cache, built on top of Linux's device-mapper architecture. Our experimental evaluation has demonstrated that it can significantly improve a storage system's performance and scalability, by leveraging the data locality with client-side storage.

Based on dm-cache, our future research will be focused on further improving the effectiveness and efficiency of storage network caching by embedding more intelligence into the cache management, especially on the decision of prefeching, replacement and write-back; building cooperative caching scheme to allow clients sharing cached data in a peer-to-peer manner; and integrating disk caching with cluster file systems that permit concurrent access to the shared storage volumes.

References

[1] Cfs - cache file system. Plan 9 Manual Pages.

[2] Dell poweredge blades. http://www.dell.com/blades.

[3] Device-mapper. http://sourceware.org/dm/.

[4] Exterme cluster administration toolkit. http://xcat.org.

[5] Hp bladesystem. http://www.hp.com/products/blades.

[6] Ibm bladecenter. http://www.ibm.com/systems/bladecenter.

[7] Open-iSCSI Project. http://www.open-iscsi.org/.

[8] The iSCSI Enterprise Target Project. http://iscsitarget.sourceforge.net/.

[9] *NFS Illustrated*. Addison-Wesley, 2002.

[10] BENT, J., THAIN, D., ARPACI-DUSSEAU, A., AND ARPACI-DUSSEAU, R. Explicit control in a batch-aware distributed file system. In *Proceedings of the First USENIX/ACM Conference on Networked Systems Design and Implementatio* (2004).

[11] BESTER, J., FOSTER, I., KESSELMAN, C., TEDESCO, J., AND TUECKE, S. Gass: a data movement and access service for wide area computing systems. In *IOPADS '99: Proceedings of the sixth workshop on I/O in parallel and distributed systems* (New York, NY, USA, 1999), ACM Press, pp. 78–88.

[12] CASHIN, E. L. Ata over ethernet: putting hard drives on the lan. *Linux Journal*.

[13] CHANDRA, R., ZELDOVICH, N., SAPUNTZAKIS, C., AND LAM, M. S. The collective: A cache-based system management architecture. In *Proceedings of the Second Symposium on Networked Systems Design and Implementation (NSDI '05)* (May 2005).

[14] CHERITON, D. R. The v distributed system. *Communications of the ACM 31*, 3.

[15] CHERITON, D. R. Vmtp: A transport protocol for the next generation of communication systems. In *Proceedings of the 1986 ACM SIGCOMM Conference on Communication Architectures and Protocols* (1986), ACM, pp. 406–415.

[16] G. PRUETT, E. A. Bladecenter systems management software. *IBM Journal of Research and Development*.

[17] HE, X., YANG, Q., AND ZHANG, M. A caching strategy to improve iscsi performance. In *LCN '02: Proceedings of the 27th Annual IEEE Conference on Local Computer Networks* (Washington, DC, USA, 2002), IEEE Computer Society, p. 0278.

[18] HE, X., ZHANG, M., AND YANG, Q. Stics: Scsi-to-ip cache for storage area networks. *J. Parallel Distrib. Comput. 64*, 9 (2004), 1069–1085.

[19] HENDRIKS, E. A., AND MINNICH, R. G. How to build a fast and reliable 1024 node cluster with only one disk. *The Journal of Supercomputing*.

[20] HOWELLS, D. Fs-cache: A network filesystem caching facility. *Proceedings of the 2006 Linux Symposium*.

[21] KATCHER, J. Postmark: A New Filesystem Benchmark. Technical Report TR3022, Network Appliance, 1997. http://www.netapp.com/tech_library/3022.html.

[22] KAZAR, M. L. Synchonization and caching issues in the andrew file system. CMU-ITC-88-063.

[23] LEACH, P., AND PERRY, D. Cifs: A common internet file system. *Microsoft Interactive Developer*.

[24] M. SATYANARAYANAN, E. A. Coda: a highly available file system for a distributed workstation enviornment. *IEEE Transactions on Computers*.

[25] MINNICH, R. G. The autocacher: A file cache which operates at the nfs level. In *Proceedings of the Winter USENIX Technical Conference* (San Diego, CA, 1993), p. 77C83.

[26] MOORE, J., AND CHASE, J. Cluster On Demand. Technical Report CS-2002-07, Duke University, Dept. of Computer Science, May 2002.

[27] NORCUTT, W. The IOzone Filesystem Benchmark. http://www.iozone.org.

[28] OLIVEIRA, F., PATEL, J., HENSBERGEN, E. V., GHEITH, A., AND RAJAMONY, R. Blutopia: Cluster life-cycle management. *IBM Research Report RC23784*.

[29] PIKE, R., PRESOTTO, D., DORWARD, S., FLANDRENA, B., THOMPSON, K., TRICKEY, H., AND WINTERBOTTOM, P. Plan 9 From Bell Labs. *Usenix Computing Systems* (1995).

[30] RADKOV, P., YIN, L., GOYAL, P., SARKAR, P., AND SHENOY, P. A performance comparison of nfs and iscsi for ip-networked storage. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies* (2004), p. 101C114.

[31] RAJAGOPAL, M., ET AL. Fibre channel over tcp/ip (fcip). Internet Draft Document, draft-ietf-ips-fcovertcpip-06.txt.

[32] SAPUNTZAKIS, C., BRUMLEY, D., CHANDRA, R., ZELDOVICH, N., CHOW, J., LAM, M. S., AND ROSENBLUM, M. Virtual appliances for deploying and maintaining software. In *Proceedings of the Seventeenth Large Installation Systems Administration Conference (LISA '03)* (October 2003).

[33] SATRAN, J., ET AL. iscsi. Internet Draft Document, draft-ietf-ips-iSCSI-08.txt.

[34] TOM KELLER, E. A. A study of the performance of diskless web servers. *IBM Research Report RC22629*.

[35] ZHAO, M., ZHANG, J., AND FIGUEIREDO, R. J. Distributed file system virtualization techniques supporting on-demand virtual machine environments for grid computing. *Cluster Computing 9*, 1 (2006), 45–56.