Optimizations in SAN and DAS

A Thesis
Submitted for the Degree of
Master of Science
in the Faculty of Engineering

by

M C Dharmadeep

Department of Computer Science and Automation
Indian Institute of Science
Bangalore – 560 012

JANUARY 2007

Abstract

This thesis consists of three parts.

In the first part, we introduce the notion of device-cache-aware schedulers. Modern disk subsystems have many megabytes of memory for various purposes such as prefetching and caching. Current disk scheduling algorithms make decisions oblivious of the underlying device cache algorithms. In this work, we propose a scheduler architecture that is aware of underlying device cache. We also describe how the underlying device cache parameters can be automatically deduced and incorporated into the scheduling algorithm. In this work, we have only considered adaptive caching algorithms as modern high end disk subsystems are by default configured to use such algorithms. We implemented a prototype for Linux anticipatory scheduler, where we observed, compared with the anticipatory scheduler, upto 3 times improvement in query execution times with Benchw benchmark and upto 10 percent improvement with Postmark benchmark.

The second part deals with selecting a primary network partition in a clustered shared disk system, when node/network failures occur. In this work, we give an algorithm for fault-tolerant proactive leader election in asynchronous shared memory(disk here) systems, and later its formal verification. Roughly speaking, a leader election algorithm is proactive if it can tolerate failure of nodes even after a leader is elected, and (stable) leader election happens periodically. This is needed in systems where a leader is required after every failure to ensure the availability of the system and there might be no explicit events such as messages in the (shared memory) system. Previous algorithms like DiskPaxos are not proactive. We also show how our protocol can be used in clustered shared disk systems to select a primary network partition. We have used the state machine approach to represent our protocol in Isabelle HOL logic system and have proved the safety property of the protocol.

The third part deals with implementing cooperative caching for Redhat Global File System. Redhat Global File System is a clustered shared disk file system. Coordination between multiple accesses is done using lock manager. Whenever a read is done, a lock on the inode is acquired in shared mode and the data is read from the disk. And, when a write needs to be done, exclusive lock on the inode is acquired and data is written to the disk; this requires all nodes holding the lock to write their dirty buffers/pages to disk and invalidate all the related buffers/pages. DLM(Distributed Lock Manager) is a module that implements the functions of a lock manager. DLM has some support for range locks, although it is not being used by GFS. In this work, we describe the changes made to the locking protocol and DLM, which is leveraged to implement cooperative caching in GFS. Experiments with micro benchmarks on our prototype implementation reveal that reading from a remote node over a gigabit Ethernet can be upto 8 times faster than reading from a enterprise class SCSI disk.

Contents

List of Figures

List of Tables

Chapter 1

Introduction-SAN & DAS

Storage area network(SAN) is a network designed to connect storage devices such as disks, tapes, to the workstations. The storage devices provide the block I/O interface which is used by the workstations. The HBA(Host Bus Adapter) driver and the SCSI driver make the storage devices on SAN appear as local devices to the workstation.

Storage Area network can be classified into two categories.

- One with a separate network to perform Block I/O operations; for example, a FC network. Sample configuration of this type is depicted in **Figure** 1.1. The three workstations on the top of the figure are connected to the JBODs via FC switches. There can be fault tolerance at multiple levels in this case. For example: there can be multiple HBA on a workstation; each workstation can have multiple paths to the JBODs.

- One in which the block I/O interface is provided by servers to which the storage devices are attached. Block I/O operations take place over the regular network. Sample configuration of this type is depicted in **Figure** 1.2. The three nodes use the block I/O interface provided the two nodes connected to the disks. These three nodes perform block I/O operations over the regular LAN(Ethernet in the figure). The two nodes that are serving the disks are internally connected to the disks via SAN.

DAS(Direct Attached Storage) is a scenario in which a single workstation is connected to a external storage directly; ie without any intervening switches. Usually, the external storage has some intelligent controller that is capable of virtualizing storage, scheduling the requests in the device queue, creating and maintaining redundancy(eg: RAID). A sample DAS configuration is shown in **Figure** 1.3. In the figure, a RAID disk subsystem is connected to the workstation directly. The workstation has a HBA(Host Bus Adapter) that communicates with the RAID device. The HBA driver and the SCSI driver make it appear as a local disk to the workstation.

1.1 GFS cluster components

Most of the work done in this thesis used Redhat GFS cluster somewhere or the other. Redhat GFS and the main components are briefly described below. GFS defines a locking interface which is implemented by different lock managers; for example: GULM, DLM. In our work we have used DLM(Distributed Lock Manager). Hence in this thesis we will only describe DLM.

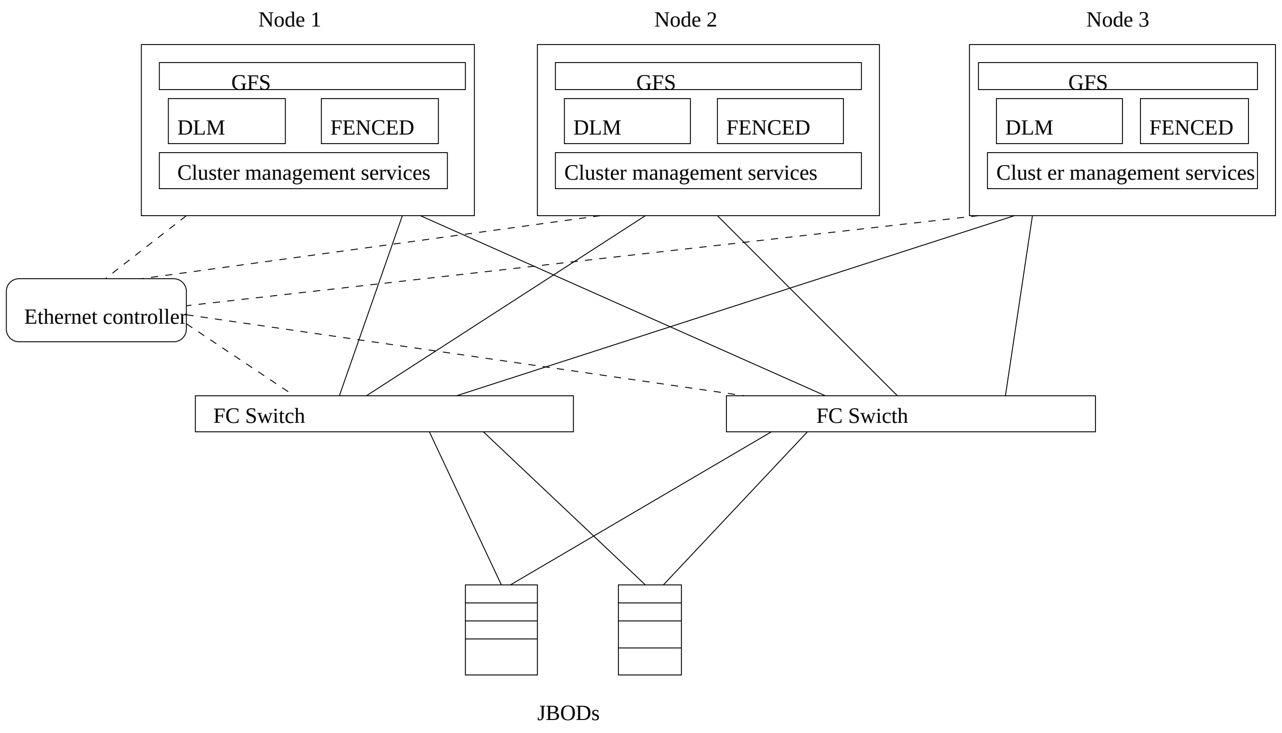

Figure 1.1: SAN configuration 1

Redhat GFS is a clustered shared disk filesystem. Each node keeps track of the nodes that are reachable via network from that node. At any given point in time, this defines the view of a node. A membership protocol is run so that all nodes in the view of a node, agree on the membership of that view. After the membership algorithm terminates, a primary network partition is selected. It is the nodes in this network partition that are allowed to access the disks. Once a primary network partition is selected, nodes not in the primary network partition are blocked from accessing the disk. This process is called fencing. Usually this is done by disabling the ports on the switches through which these nodes are accessing the disks. Once this is done, recovery is done to bring the system to a consistent state. This process is repeated every time a node failure is detected by a node.

The main components of a Redhat GFS cluster are:

- **CMAN(Cluster Manager):** This part of GFS cluster runs the group membership algorithm and selects the primary network partition.
- **CCSD(Cluster Configuration Service Daemon):** All the information on nodes, FC switches, fencing methods is provided in a file by the system administrator. The CCS Daemon reads this file and makes information available to other subsystems of GFS cluster. For example, the CMAN subsystem gets the the quorum rule from here.
- **FENCED(Fence Daemon):** The Fence daemon is responsible for performing the fencing operation. It gets the name of the fencing scripts and its arguments from the CCS daemon.
- **DLM(Distributed Lock Manager):** DLM manages the various resources and corresponding locks on these resources requested by the lock_dlm running on various nodes.

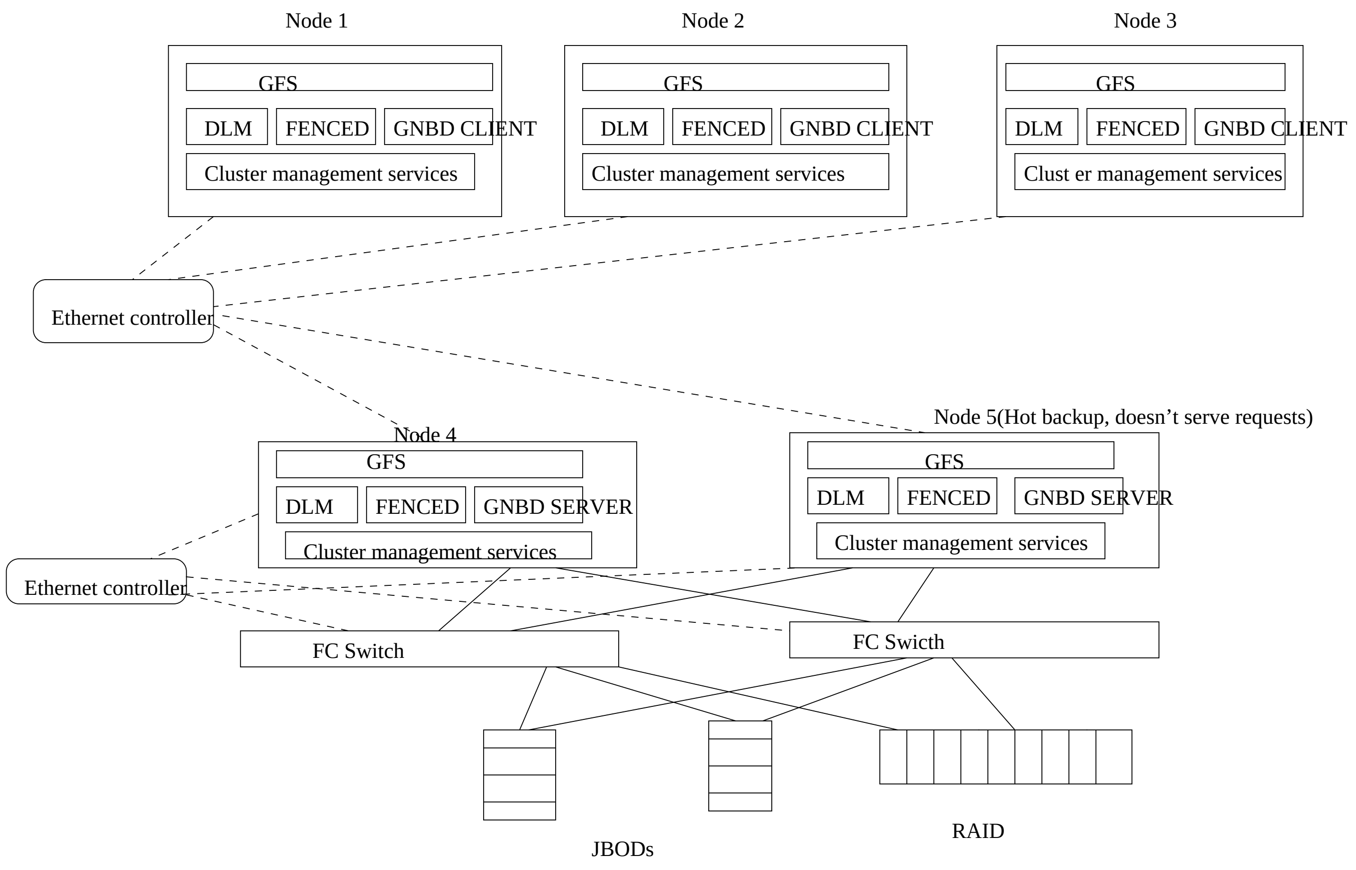

Figure 1.2: SAN configuration 2

- **LOCK_DLM:** The lock_dlm part of GFS implements the locking functions required by the GFS. In the process, it communicates with DLM.
- **GFS(Global File System):** The GFS subsystem contains the file system specific code. GFS communicates with DLM via its lock_dlm module.

1.2 Our contribution

This thesis is composed three optimizations:

- The first optimization, device-cache-aware schedulers, applies both to SAN and DAS. To be more specific, it only applies to SAN of second type as mentioned above. Caching and prefetching is done at various level in modern day UNIX based systems. They are done at file system level, block device layer level(usually only prefetching is done here) and the device cache level. Only misses at the first two levels of caching arrive at the block I/O scheduler. The schedulers that are currently in use, make decisions oblivious of the underlying device cache. In this work, we propose a scheduler architecture that is aware of underlying device cache. We found considerable improvements over existing scheduler, for high end disk subsystems . In case of DAS, the changed scheduler runs on the workstation. In case of SAN, it runs on the workstation running the NBD(Network Block Device) server.

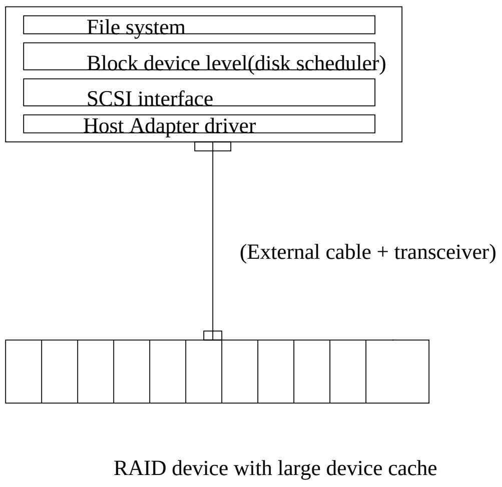
File system
Block device level(disk scheduler)
SCSI interface
Host Adapter driver
(External cable + transceiver)
RAID device with large device cache

Figure 1.3: Direct Attached Storage

- The second optimization is regarding the primary network partition selection in SAN in face of node/network failures. This happens after the group membership algorithm terminates(CMAN does this in case of GFS) and before the fencing takes place(Fenced in case of GFS cluster). Most of existing implementations use a quorum based rule. One such rule is: select the network partition which has more than half the total number of nodes. As one can see such a rule can fail; for example, in a two node cluster. Other implementations use fencing which can also fail without some support from the fence devices. We propose a proactive disk based consensus algorithm to select the primary network partition in clustered shared disk systems. We also propose another algorithm for Brocade FC switches which uses a property of the switches.

- The third optimization is cooperative caching in SAN, specifically for Global File System. GFS locking interface supports both shared and exclusive locks. Whenever data needs to be read from a file, it acquires a lock on the inode and reads the data from the disk. It does this even if the data is present at a remote node that previously held the lock. Whenever a write needs to be done, a exclusive lock is acquired on the inode and data is written to the disk. This requires all the nodes which are holding the lock for this inode, to flush data and metadata to the disk. Later, when a node requests a shared lock, the exclusive lock on the node which performed the write is demoted. After the shared lock is acquired, the node again reads the data from the disk. In this work, we propose and evaluate a cooperative caching protocol using which some of these reads can be eliminated. We have also measured the performance improvement when this is combined with device-cache-aware anticipatory scheduler in SAN of type 2(as described above).

The thesis is organized into three parts. Each part deals with a optimization in the same order as above. The bibliography relating to a optimization is at the end of the part.

Part I

Device-Cache-Aware Schedulers

Abstract

Modern disk subsystems have many megabytes of memory for various purposes such as prefetching and caching. Current disk scheduling algorithms make decisions oblivious of the underlying device cache algorithms. In this work, we propose a scheduler architecture that is aware of underlying device cache. We also describe how the underlying device cache parameters can be automatically deduced and incorporated into the scheduling algorithm. In this work, we have only considered adaptive[1] caching algorithms as modern high end disk subsystems are by default configured to use such algorithms. We implemented a prototype for Linux anticipatory scheduler, where we observed, compared with the anticipatory scheduler, upto 3 times improvement in query execution times with Benchw benchmark and upto 10 percent improvement with Postmark benchmark.

[1]Adaptive device caching algorithms vary the prefetch size depending on the amount of contiguous data requested.

Chapter 2

Introduction and Motivation

Current Linux kernel has disk schedulers like deadline, CFQ, and anticipatory that are at the block device level. Some of these algorithms, such as deadline or anticipatory, schedule requests to reduce the amount of time spent in seek operations. This is accomplished by scheduling requests to reduce the overall seek time. In addition, an anticipatory scheduler may also wait for a better request than the ones currently in the queue. While such scheduling algorithms make use of prefetching done at the device cache due to the temporal and spatial contiguity of current sequence of disk requests , they completely neglect the fact that these prefetched buffers are in the device cache for a while before they are evicted.

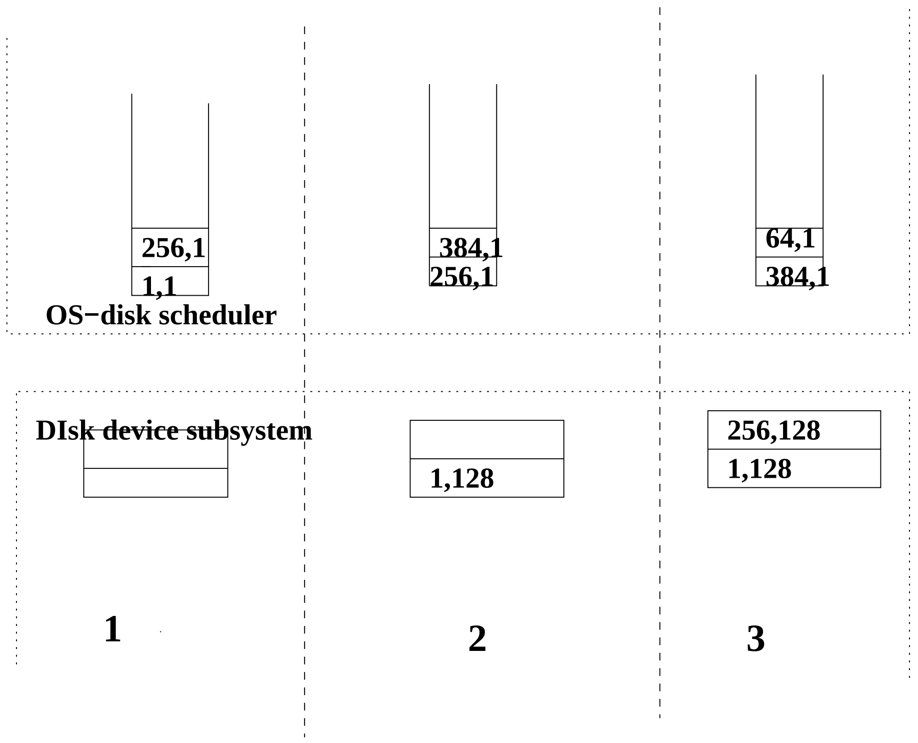

Figure 2.1: Scheduler Architecture

To illustrate this point, suppose the device cache has a capacity to hold 2 128-sector tracks and that the remaining part of the track after the request is also fetched into the cache. Let (a, b) denote a request for b sectors starting from the a^{th} sector. Suppose four requests are made in the following order: $(1, 1)$, $(256, 1)$, $(384, 1)$, $(64, 1)$, and the request for $(64, 1)$ arrives after the scheduler dispatches the request for $(256, 1)$. Assuming device cache follows LRU replacement policy and the requests are served using an elevator algorithm, the contents of the cache after each request would be the following respectively:[1] $\{(1, 128)\}$, $\{(1, 128), (256, 128)\}$, $\{(256, 128), (384, 128)\}$, $\{(384, 128), (64, 65)\}$. But one can easily see that $(64, 1)$ can served before $(384, 1)$ with zero seek time. The same is illustrated in **Figure** 2.1.

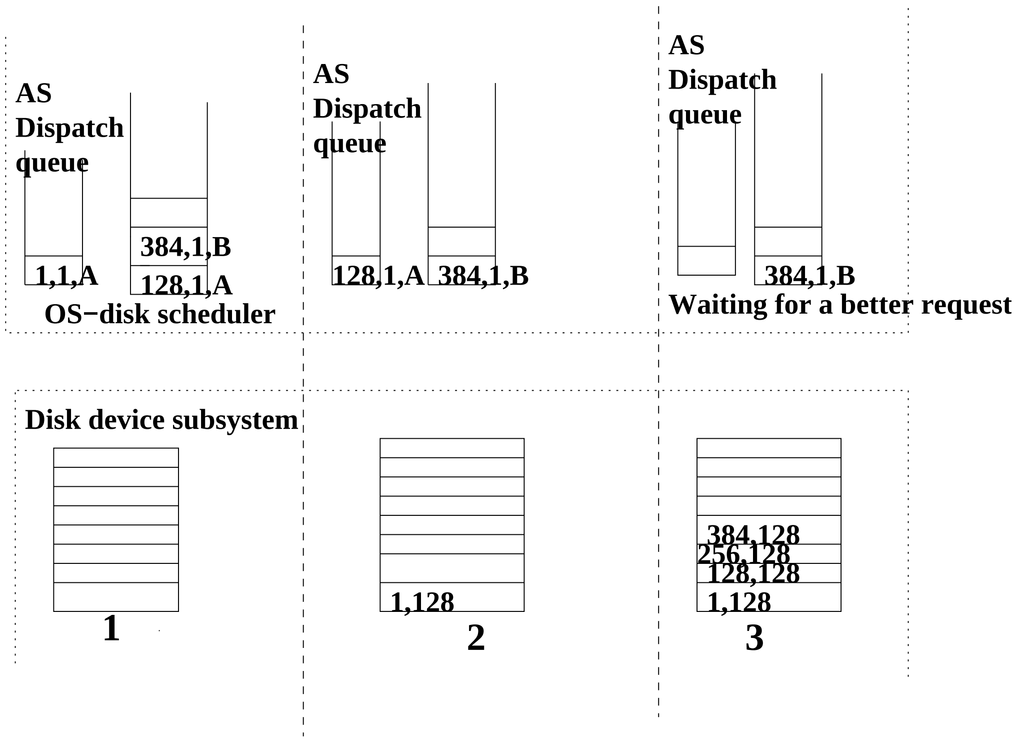

Figure 2.2: Scheduler Architecture

Now consider an anticipatory scheduler; an anticipatory scheduler can wait without dispatching queued requests, if it expects a request involving lesser seek time from the process whose request was the last served and doing this does not result in any request missing its deadline. If the anticipated request is in the prefetch buffer and the "nearest" request in the queue is not "very far" from the current position of the disk head, effective disk bandwidth can be increased by going ahead and dispatching the "nearest" request in the queue. To illustrate this point, suppose the device cache has a capacity to hold 8 128-sector tracks and also has a prefetch rule "If two consecutive tracks, both spatially and temporally, are fetched, also prefetch the next 2 tracks". Let $(a,\ b,\ c)$ denote the process c's request for b sectors starting from the a^{th} sector. Suppose the scheduler queue has synchronous requests in the following order: $(1, 1, A)$, $(128, 1, A)$, $(384, 1, B)$ and the anticipated seek distance of process A is calculated as 1 track. An anticipatory scheduler, after serving the first two read requests, would wait for a "better" request (here around $(256, k, A)$ with $k < 128$) rather

[1]We are assuming that the requested blocks are also cached in this section

than going ahead with the next request in the queue. But the next request in the queue $(384, 1, B)$ can be served immediately if the scheduler knows that the anticipated ″better″ request can be served from the cache in the ″near″ future. The same is illustrated in **Figure** 2.2.

We propose a device-cache-aware scheduler architecture to exploit the large caches present in current devices. For example, Proware RAID device[proware] has a 128MB cache[2], while a single enterprise class SCSI disk such as Seagate ST336752FC has a 8MB cache.

We also need to discover many of the low-level parameters of the device subsystem and make it available to the scheduler. This requires profiling using artificially generated workloads to deduce the parameters.

One can argue that the locality of the read/write requests as seen by the device is destroyed because of caching and read ahead done by the operating system. However, the read-ahead at the file system block level is usually smaller than the number of sectors read into the device cache for a single read request (which is usually a track as reported by the controller,in case of SCSI) and sometimes not contiguous on the disk. In addition, in most operating systems, the buffer/page cache size varies with the amount of free memory available, whereas the device cache memory is used for prefetching/caching most of the time.

Another argument against device-cache-aware scheduler could be that the size of disk cache is small today to have considerable effect on the performance. Our experiments show three-fold improvements on some benchmarks and we expect better results with larger device caches than what we have used. We expect that the device cache sizes will increase, as disks become more dense, to catchup with increase in bus bandwidths and processor speeds. As a part of our future work, we intend to investigate the performance of disk scheduling algorithms on disks having much larger device caches through trace driven simulation.

In our work, we assume disks are not shared between multiple machines. One way to circumvent this problem in case of shared disks, is to have a NBD[3] running on a machine serving the shared disks, and have other machines import disks exported by the NBD server. Any scheduler changes can be incorporated in the NBD server.

Modern disk subsystems use different caching algorithms based on the kind of workload they expect. High end RAID devices,which are meant to be used in server environments, have considerable amount of cache memory (>128MB) and by default configured to run adaptive caching algorithms. These devices also have enough memory to hold >128 requests and intelligently schedule between them. One can try to improve the effective read/write bandwidth by presenting all the requests to the controller. However, there is going to be some time lag between two consecutive request dispatches by the OS. It is not clear to us , what the controller does during this time lag. Moreover, in case the number of requests is large(both sync and async combined) OS still has to decide which requests to dispatch to the device controller. In any case, this issue remains to be studied. Enterprise class SCSI disks, which typically have 4-8MB cache, keep switching between adaptive and non-adaptive algorithms based on the prefetch buffer hit ratio; these can be used in both desktop and server environments. One can refer to sections 4.5.2 and 4.5.3 of [Cheetah36LP-FC] for more details. Others just use non-adaptive caching algorithms. Non-adaptive caching algorithms use fixed number of circular segments.

[2]At the ″extreme″ end of such devices like Lightning 9900[hds], the cache may be as large as 32GB and above frontending large number of disks. Our work does not discuss such large cache devices.

[3]NBD stands for Network Block Device

In our work so far, we have only considered adaptive caching algorithms; adaptive caching algorithms vary the prefetch segment size based on the number of temporally and spatially contiguous segments requested, also known as ARLA(Adaptive Read Look-Ahead) algorithms. We did also try to profile some enterprise class SCSI disks; these are discussed in evaluation section. Automatic deduction of parameters of a device cache that uses a non-adaptive algorithm can be found in [Schindler99].

We have made some assumptions based on the study of a RAID device. For example, our cache profiler, which automatically characterizes the device cache, checks if LRU replacement policy is used in the device cache. If not, no further analysis is done as of now. The disks we have tested so far use the LRU policy. (The assumptions we have made are further discussed in section 3.) Currently our implementation profiles only SCSI device caches and generates a link-able kernel module to make information available to the scheduler.

The rest of the part is organized as follows. In chapter 3, we describe the scheduler architecture which is the basis of our prototype. In chapter 4, we describe the cache profiler and changes to the Linux anticipatory scheduler. In chapter 5, we evaluate the scheduler architecture with our prototype. We then conclude with related and future work.

Chapter 3

Scheduler Architecture

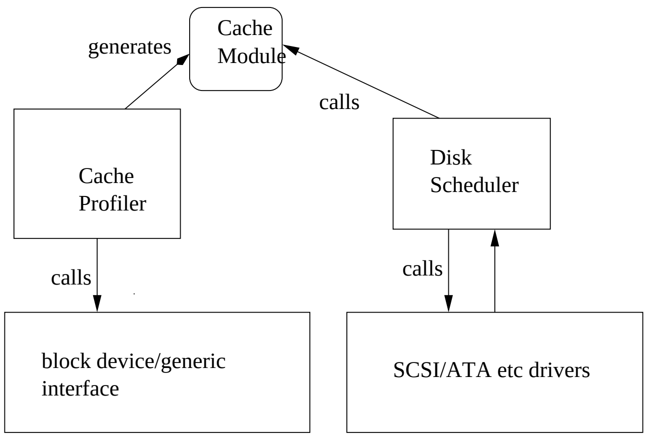

Figure 3.1: Scheduler Architecture

Our cache-aware disk scheduler is incorporated at the block device level in the Linux kernel. Our design has both user and kernel components; the user component deduces device parameters and generates a kernel module that can be used any existing disk scheduler. The proposed scheduler architecture is shown in **Figure** 3.1.

The cache profiler is a user space privileged program that calculates various parameters of the device cache, such as the total number of segments the cache can accommodate, number of contiguous segments prefetched for *k* consecutive (both temporally and spatially) segment reads. A cache segment here is the minimum amount of data fetched for each read request assuming that the request is segment size aligned.

After the various cache parameters are calculated, a loadable kernel module is generated that is linked into the kernel during the runtime. The loadable kernel module tracks the cache contents and generates a hint of the request that could be served from the cache. The default scheduler code in the kernel can be modified to call the cache module to make use of the hint; our modifications are in the anticipatory scheduler of the Linux kernel. Calls are made into the module to update the cache state after a request is serviced, to queue the requests which could be satisfied by cache and to select the next request to be serviced if the queue of cached requests is not empty.

The cache profiler is divided into a storage protocol specific part and a storage protocol independent part. The storage protocol specific part supplies disk geometry parameters such as segment size and an upper bound on disk bandwidth possible (when being served from the magnetic medium) to the storage protocol layer independent part. The disk read/write commands are implemented in the storage protocol specific part. The storage protocol independent part uses the interface provided by the storage protocol dependent part to compute various parameters such as the prefetch size as a function of number of contiguous segments requested, the total number of cache segments, and generates a linkable kernel module that can be used by any disk scheduler.

We have implemented a prototype of the scheduler architecture for the Linux anticipatory scheduler. In next section, we describe the cache profiler and the module generated by it in more detail. We also describe the changes made to the Linux anticipatory scheduler to call into the module.

Chapter 4

Implementation

4.1 Cache Profiler

The cache profiler is a user privileged level program to calculate those parameters of the device cache that could affect the scheduling decisions. It uses a set of virtual functions to calculate these parameters. The implementation of these virtual functions is specific to underlying disk communication protocol. Currently we have implemented the virtual functions only for SCSI devices. This implementation makes use of SCSI generic utils package (sg3_utils package).

We list the assumptions made about the device cache below. We consider only adaptive disk caching algorithms here.

- Data is read into the device cache in multiples of some k number of sectors. This can depend on the underlying protocol. For example, in case of SCSI, *Buffer full ratio* in *Disconnect-Reconnect mode page* gives us this value.

- There is a minimum size of cache which should be empty when data is written into the disk. We assume this is same as the above k; this was found empirically in all disks we have studied. For example, in case of SCSI, *Buffer empty ratio* in *Disconnect-Reconnect mode page* gives this value and is usually equal to the *Buffer full ratio*.

- The amount of prefetch for reads is a function of the amount of contiguous reads (both spatially and temporally) done by the disk driver. Note that this prefetch amount is independent of the data already present in the cache. We assume that the same amount of cache should be emptied when contiguous writes are done to disk.

- The cache controller may or may not cache the requested sectors. If it caches the requested sectors, the cache segments are segment size aligned Otherwise, they can start from any location on the disk. If a read is done some where in the middle of a segment, the remaining part of the segment after the end of the request is also fetched from the disk into the device cache. However, in this case the initial area between the segment boundary and the beginning of the requested block is not fetched and that area of device cache is not available for caching for other requests until reused. Similar rule holds for a write also.

- We assume the cache replacement policy is LRU. The disks we tested on, happen to follow LRU. The current implementation does not modify the scheduler if the cache replacement policy is not LRU.

One can view the above assumptions as a model, currently we have in mind. The model determines the state space in which we will be searching for a solution. It is quite possible that the state space doesn't have solution, in which case we have to go back and refine the model. This model as of now, does not embody the fixed number of circular segments model.

Please note the difference in assumptions from the ones in [Schindler99]: for example, the fixed number of circular segments model assumed in [Schindler99] holds for disks when running in non-adaptive caching mode. We intend to refine the model(or assumptions) in future, so that non-adaptive caching algorithms are also taken care of.

Given the above assumptions, what remains to be determined is the segment size, maximum number of segments the cache can hold at any point in time and the prefetch size as a function of n consecutive segments requested with a segment holding k sectors. Also note that as a consequence of our assumptions, it is sufficient to determine these values by only doing reads from disk; writing to disk might result in data corruption, which we want to avoid.

The underlying storage protocol specific layer code in the cache profiler has to implement the following two primary functions: *getdevparm, executeread*. The *getdevparm* function returns the following underlying parameters: tracks per zone, sectors per track, bytes per sector, interleave factor, number of cylinders, number of heads, rotation rate, maximum disk speed (derived from other parameters), minimum cache read size (for SCSI, same as buffer full ratio; is also segment size) and minimum cache write size (for SCSI, same as buffer empty ratio; is also segment size).

Modern disks are divided into zones; number of sectors per track varies with the zones. So, some of the above parameters actually don't make sense. Modern disk controllers mask out this detail and make it appear as if the sectors per track is same on the whole disk. We found that the cache controller by default uses this interface rather than actual disk geometry. For example, the DISC bit of caching mode page in case of SCSI is by default turned off, which means that disc track boundaries can be crossed while prefetching; this makes sense to us because data should be prefetched based on user accesses rather than underlying disk geometry.

The *executeread* function executes a supplied read request and returns the time taken and the bandwidth realized for the read request. It takes as arguments the amount to data to read, the device file name and number of sectors to skip before starting to read. Currently, we have implemented these functions only for SCSI protocol. Most of the parameters returned by *getdevparm* in this implementation are extracted from the values present in the SCSI mode pages; as mentioned earlier, don't correspond to actual disk geometry. An upperbound on the disk bandwidth is calculated from the values present in *Rigid Disk Geometry mode page* using the following formula.

$(rotation\ rate\ (rev/min)$
$*bytes\ per\ sector*\ sectors\ per\ track*$
$multiplying\ factor) \div (interleave * 60 * 1024 * 1024)$

The multiplying factor, a user supplied parameter, is 2 if data is being served from a single disk. It is equal to the twice number of disks containing data blocks in case of RAID. As mentioned earlier, modern disks are divided into zones; so reading from the outmost zone is faster than reading from the inner ones. The value calculated by the above formula without the multiplying factor lies between the maximum disk bandwidth possible(when being read from the outermost zone) and the minimum disk bandwidth possible (when being read from the inner most zone). We found this factor, between the maximum and minimum bandwidth, is less than two on the disks we used. Note that for disks with high RPM rate, it can be difficult to distinguish between a sequential read and a cache hit. We do random seeks in between to get around this problem. Some disks interleave data on a track; this means consecutive sectors as seen by the host controller on a track are separated by one or more sectors on the actual disk. In such cases, we need to divide by the interleave factor during disk bandwidth calculation. We divide by 60, because $rotation\ rate$ is in revolutions per minute. We divide by $1024*1024$ to get the answer in MB/sec. If the bandwidth observed during a read is greater than the above calculated value, it is inferred as a cache hit.

Before determining the number of segments and the prefetch size as a function of consecutive number of segment reads, some checks are performed to check if the cache controller follows LRU replacement policy or not and also if the controller caches the prefetch segments along with the requested segments or just the prefetch segments. Next, the prefetch size as a function of consecutive number of segment reads is calculated. Then the total number of segments is calculated. The high-level algorithms for these are given below; they are simplified for presentation here. In the actual implementation, certain parts of the algorithm are repeated to get to a confidence level. We require that the disk is not mounted when the cache profiler is being run. Most of these algorithms are similar to ones in [Schindler99]. However, the model assumed is different.

4.1.1 Caching behavior

This is required as some controllers cache the prefetch segments along with the requested sectors while some only cache the prefetch segments. The algorithm used to check if the controller caches the requested segments or not, is quite simple; read a segment with random disk offset, do a seek to a random location and reread the segment. If hit occurs, the controller caches the requested segments also.

4.1.2 Prefetching Behavior

This function checks if prefetching is done or not. If not, the cache profiler simply exits without making any changes to the scheduler. The algorithm to check if the controller does prefetching or not is simple; read a sector at random disk offset which is segment size aligned and then read the sector following the previous read. If a hit occurs, the device prefetches.

4.1.3 LRU Detection

The algorithm to check if the controller follows LRU replacement policy is given in **Algorithm** 1. It is quite possible that a controller using non-adaptive caching algorithm(fixed number of circular segments model) passes this test. As mentioned earlier, so far we have taken into consideration, only adaptive caching algorithms.

```
Output: true if cache follows LRU
$k$ = 1;
while true do
    Read upto $k$ cache segments, each time with a different offset from start of disk;
    /*this uses the executeread function*/
    Reread the same $k$ cache segments again in same order;
    /* when being reread, if the controller caches the requested sectors also, the reread starts
    from the beginning of the request made in first step of while loop. Otherwise, it starts
    from the end of the request */
    if No Cache Miss then
        $k++$;
    end
    else
        if the first missed segment is the first segment read then
            return true;
        end
        else
            return false;
        end
    end
end
```

Algorithm 1: Algorithm to determine if controller follows LRU

4.1.4 Prefetch size deduction

Our study of some devices such as Proware RAID device or Seagate SCSI disks have shown some interesting behavior. On a small contiguous segment read, caching is expected to be the better strategy whereas prefetching to be the better at larger ones. There is of course a natural limit to the prefetch size imposed by the device cache. We give the observed values for the prefetch for Proware and Seagate devices in the **Table** 4.1 (we have mentioned values upto 5 contiguous segments reads, though we have conducted experiments with upto 20 contiguous segment reads): Note that, the Seagate disk we have used, disables prefetches if

Table 4.1: Prefetch segments as a function of number of segments read

segments read	Proware SB3160	Seagate ST336752FC
1	$1 + 0 = 1$	$0 + 2^3 = 8$
2	$2 + 2^3 = 10$	$0 + 2^3 = 8$
3	$3 + 2^4 = 19$	$0 + 2^3 = 8$
4	$4 + 2^4 = 20$	whole cache used for prefetching
5	$5 + 2^4 = 21$	whole cache used for prefetching

it finds out that prefetch buffer hits are not happening. The precise point at which it disables prefetching is still not clear to us. The cache profiler is carefully written so that it doesn't result in prefetch buffer misses. Moreover, the cache size is only 8MB; a device-cache-aware scheduler may not result in a performance gain.

From the values, we can surmise that Proware sets the prefetch size variably (as a non-decreasing function of requested size) till a maximum value whereas Seagate makes a simpler binary decision. Any other strategy such as a non non-decreasing function seems counterintuitive. Hence, our algorithm attempts a simple interpolation strategy. The algorithm to calculate the number of segments prefetched for k consecutive segment reads is given in **Algorithm** 2

```
Input: Upper bound on number of segments(This could be calculated from the cache
       controller memory available (from the manual) and the segment size. This
       information is used to poison the cache with random contents in the while
       loop.), number of consecutive segments to be read.
Output: prefetch segment size
/*The values returned also includes the requested segments if they are cached*/
$prefetch$ = number of segments to read + 1;
while true do
    choose a random disk offset;
    $prefetch\_seg$ = disk offset + $prefetch - 1$;
    read requested number of consecutive segments starting from the disk offset;
    do a random read from the portion of disk not containing these segments;
    /* this increases the time elapsed if the next read done from magnetic medium */
    read the segment starting at $prefetch\_seg$ from the disk;
    if cache hit then
        /* inferred due to the time difference between read from magnetic medium and from
        cache */ $prefetch++$;
    end
    else
        break;
    end
    poison the whole cache with random contents from the portion of disk (these
    should not have been read in the above part of the while loop);
end
$prefetch--$;
if the controller does not cache the requested segments (Alg. 1) then
    $prefetch -=$ number of segments to read;
end
return $prefetch$;
```

Algorithm 2: Algorithm to determine prefetch size for given number of segments requested

First, the number of prefetch segments is determined for 1 to, say, 10 consecutive segment reads. If the difference between the number of segments prefetched when 8 and 9 consecutive segment reads are done is equal to the difference between the number of segments prefetched when 9 and 10 consecutive segment reads are done, then the number of segments prefetched for the rest of the higher values (say, x) is calculated by adding this difference to the number of segments prefetched when one less ($x-1$) consecutive segment reads are done. Otherwise, the process is carried on with higher number of consecutive segment reads until the number of segments prefetched for the rest can be calculated by above means. In other words, if a graph is plotted with number of prefetched segments on Y-axis and number of consecutive segment reads on the X-axis, and if some three points with consecutive x-coordinates and whose x-coordinate value is greater than 7, fall on a straight line, then the number of prefetched segments for values greater than x-coordinates of these points is given by points on this line. We decided to take a minimum of 10 points to avoid doing interpolation prematurely.

4.1.5 Deducing the total number of segments

The algorithm to deduce the total number of segments is given in **Algorithm** 3.

```
Input: Upper bound on the total number of segments, prefetch segment size when the
       number of consecutive segment reads is 1
Output: Total number of cache segments
$lowerbound = 0$;
$upperbound =$ upper bound on number of segments;
$size = 0$;
$newsize = (lowerbound + upperbound)/2$;
while $size \neq newsize$ do
    read $newsize$ number of evenly spaced(which is calculated by taking into account
    prefetch segment size when the number of consecutive segment reads is 1) sectors
    which are segment size aligned from the disk remembering the offset parameter;
    reread the first segment read;
    if cache miss then
        $upperbound = newsize$;
    end
    else
        $lowerbound = newsize$;
    end
    $size = newsize$;
    $newsize = (lowerbound + upperbound)/2$;
    poison the whole cache with random contents from the portion of disk (these
    should not have been read in the above part of the while loop);
end
return (prefetch segment size when number of requested segments is 1) $* size$;
```

Algorithm 3: Algorithm to determine the total number of segments

4.1.6 Generating the kernel module

After the estimation of the parameters, the cache profiler generates a module file containing a macro defining the maximum number of cache segments, a macro denoting if the requested segments are cached or not and a function which returns the number of cache segments prefetched taking as argument the number of consecutive segment reads. It is also possible to implement in such a way that values defined by macros are passed using ioctl; we have not done this so far. The rest of the code (which contains the module initialization code) is common and just printed out into the file. The common part of the code is same for all the disks which fall under the model we have considered. Memory can be saved by generating two modules: one of which is caching model specific, other disk specific. Our prototype currently generates a single module containing both the caching model specific and disk specific code. Details of the common part of code are given below.

The code generated by the cache profiler implements the following virtual functions which are called from the disk scheduler code.

- *process_cached_request_fn:* This function is called from the scheduler code, if a request could be served from the device cache. The function queues the requests in linked lists associated with each of the segments from which all or part of the request could be served. (With our current implementation, it is quite possible that a request is queued in the linked lists mentioned above, and later a cache segment which serves all or part of the request, gets evicted before the request is served. We think this problem can be solved to some extent by not queuing the request in the linked lists mentioned above, if it is very close to the cache segment that is about to be evicted; we have not tried this change so far.)

- *cache_hit_fn:* The scheduler code calls this function when a new request is queued. This function returns *true* if a cache hit occurs. Otherwise, it returns false.

- *update_cache_fn:* The scheduler code calls this function after a request is completed. This function updates the mapping between the cache segments and the disk blocks those cache segments contain. The mapping is maintained by the module generated by the cache profiler. The function which calculates the number of prefetch segments, generated by the cache profiler, is used here.

- *remove_cached_request_fn:* The scheduler code calls this function after the next request to serve is chosen. The function removes the requested function from all the queues corresponding to the cache segments from which this request could be completely or partially served.

- *is_empty_fn:* The scheduler code might call this function before deciding on next request to serve. The function returns *true* if the number of requests pending that could be served from device cache is zero.

- *next_cached_request_fn:* The scheduler code might call this function before deciding on next request to serve. The function returns a request that could be served from cache, if all of the following conditions hold:

  – There is a request which could be served from the cache.
  – Serving this request from the cache will not result in any disk read (not even disk reads involving prefetching of segments).
  – Either the previous request served and this request are not contiguous or serving this request right after the previous request will not result in a disk read.

  If more than one request could be served from the cache, that request is returned whose corresponding cache segments are nearest to the cache segment that will be evicted if a request is served that is not in disk cache.

4.2 Changes to the Linux anticipatory scheduler

We have changed the anticipatory scheduler[Iyer01b] of Linux 2.6.14.4 kernel to make use of cache information, if available. We chose anticipatory scheduler over other schedulers because this is the default scheduler installed with the linux kernel. It would have been better if changes could be made independent of the underlying scheduler algorithm, but we believe this would result in more extensive code changes than changing the code of each scheduler. The changes to Linux[kernel] kernel anticipatory scheduler are described below. Similar changes could be done for other schedulers too.

- *elv_try_merge:* This function is called to merge a new request with a queued request. If an already queued request could be served from the cache, don't merge the new request with the queued request. This is done to prevent situations in which the queued request prior to merge can be served from the cache, but the request after the merge can't be entirely served from cache. The *cache_hit_fn* is called from here for this purpose.

- *as_can_anticipate:* This function is called to check if we can wait for a better request than the ones already in the queue. If, say, 128 sectors from the anticipated sector could be served from the cache, return *false*; the number 128 is good in that most of the request sizes are less than this number. We have tried using request size which is updated after each request is served, in a manner similar to how seek distances are updated; but that led to inferior performance. We have also tried using other constant request sizes as well, which didn't work so well. The performance of the benchmarks in these cases was affected either because of large number of false positives (the actual size of request made is greater than the anticipated request size) or large number of false negatives (the actual size of request made is less than the anticipated request size). The *cache_hit_fn* is called from here for this purpose.

- *as_completed_request:* This function is called after a request is served from the disk. The *update_cache_fn* is called from here to update the mapping being managed by the module generated by the cache profiler.

- *as_remove_queued_request:* This function is called after the next request to serve has been decided. The *remove_cached_request_fn* is called from here to remove the request queued on the linked lists corresponding to the cache segments from which the request could be partially or completely served.

- *as_dispatch_request:* This function is called to choose the next request to serve. The *next_cached_request_fn* is called from this function in the following situations:
  - we are anticipating on a better request, ie *as_can_anticipate* returned *true*
  - a cached request could be served without missing the deadline of any of the queued requests.

- *as_insert_request:* This function is called to insert a request in the scheduler queue. First, this function is modified to check if the request could be served from cache by calling the *cache_hit_fn*. If so, the *process_cached_request_fn* is called to queue the request on the linked list corresponding to the cache segments from which the request could be partially or totally served.

The module generated by the cache profiler takes the major and minor numbers as the arguments, using which it accesses the *block device struct*. The *block device struct* is modified to hold pointers to the functions implemented by the cache profiler generated module. The scheduler code is modified to check if these functions are defined, and if so, to make use of these functions.

Chapter 5

Evaluation

We conducted experiments using Dual Processor AMD Opteron machine with 2 GB RAM with Proware SB-3160 RAID device attached to it and running Linux[kernel] 2.6.14.4 with ext2[Ext2] file system on top of it. The RAID device has 128 MB of cache memory. The controller was configured to use RAID 5 with stripe size of 128KB. This experimental setup is same for all the benchmarks used.

The RAID controller uses adaptive caching algorithm. The cache can hold 1862 128-sector segments. The number of prefetch segments for consecutive segments reads is given by the following function: .

```
Input: Number of consecutive segment reads
Output: Number of prefetch segments
if Number of requested segments is 1 then
    return 1;
end
if Number of requested segments is 2 then
    return 10;
end
return number of request segments + 16;
```

Also, Proware SB-3160 cache controller caches the segments requested along with those prefetched. The value returned by the above function also includes the requested segments that are cached. Although all our experiments were conducted using the RAID device, we ran our cache profiler also on Seagate ST336752FC disk. The prefetch size for consecutive segment(128-sector) reads is given by the following function.

```
Input: Number of consecutive segment reads
Output: Number of prefetch segments
if Number of requested segments is less than 4 then
    return 8;
end
return 80;
```

This disk does not cache the sectors requested. The whole device cache acts as a prefetch buffer if more than 3 consecutive segments (both spatially and temporally) are requested. So, in this case once more than 3 consecutive segments are requested, the *next_request_fn* will return NULL. Moreover, the disk automatically disables prefetching, if prefetch buffer hits do not occur. Our tool is carefully written not to result in prefetch buffer misses, if at all the disk controller does prefetching. The total number of segments is 80 for this disk, when in adaptive caching mode.

5.1 Micro benchmark with varying think times

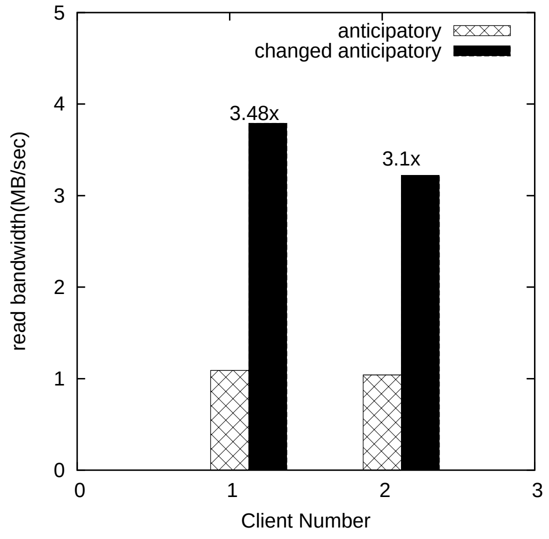

Figure 5.1: Micro benchmark with varying think times, read performance

Figure 5.1 shows the performance improvement of changed anticipatory scheduler over anticipatory for micro benchmarks with varying think times. Two processes read two different files of size 25 MB from the disk. Block device level prefetching of Linux 2.6.14 kernel has been disabled during this experiments(not the file system level prefetching), to make the effect of changed anticipatory more explicit. This was done only for this benchmark, not the following ones. The average latencies of the clients when anticipatory scheduler is used are 3.68 and 3.8 milliseconds respectively. The average latencies when changed anticipatory is used are 1.04 and 1.3 milliseconds respectively.

5.2 Postmark Benchmark

Postmark[Katcher97] is a benchmark designed to model news-server and mail-server workload. We configured Postmark to use file sizes between 500 bytes and 10 MB. We ran the benchmark using upto 3 Postmark processes. In the figures, we have also indicated the percentage of read/write bandwidth improvement over the anticipatory scheduler. There is no considerable difference among the read bandwidth reported by separate Postmark processes in all cases.

Figure 5.2 and **Figure** 5.3 show the read and write performance with the benchmark respectively. Each value plotted is the average value of three trials. The standard deviation among the values in all the three trials is insignificant. One might wonder why there is an

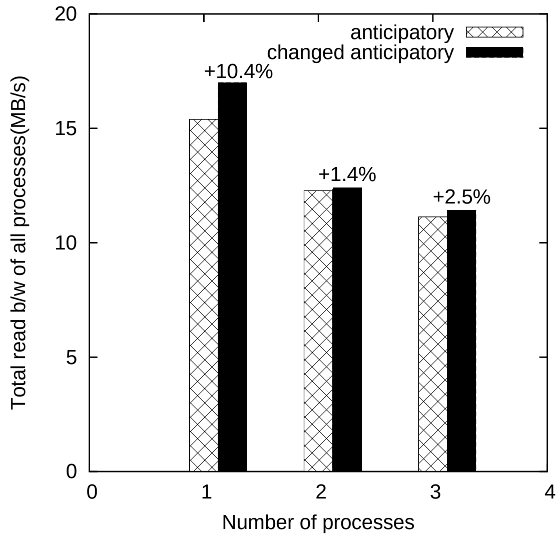

Figure 5.2: Postmark benchmark, read performance

improvement in the read/write bandwidth with the changed anticipatory scheduler over the Linux anticipatory scheduler in the single client case. Note that anticipation is done only for synchronous requests. So, if the scheduler anticipates a better synchronous request, the scheduler waits rather than serving the asynchronous batch requests. But, the changed anticipatory scheduler goes ahead with the asynchronous batch requests, if the anticipated request is likely to be in cache. Also, we have found that when number of processes is 1, deadline and CFQ schedulers perform better than the anticipatory scheduler, but not better than changed anticipatory.Moreover, deadline and CFQ schedulers can also be changed to make use of device cache information. Note that, the write performance is better than the read performance. This is because the RAID device has NV-RAM to which it logs the writes and later writes them to the disk.

5.3 Bonnie Benchmark

Bonnie[Bonnie] benchmark measures the performance of Unix file system operations. The benchmark creates a very large file and performs write/read/seek operations on it. It also creates/deletes a fixed number of files both sequentially and randomly to calculate creation/deletion rates. The file size using which the sequential input/output tests were performed is 5.5 GB. Note that we have plotted the combined bandwidth of all processes. The plotted values are average of 6 trials. Around 1 value of a client in each of the 6 trials deviated upto $+25\%$ in all the cases. The percentage of performance improvement over anticipatory scheduler is also indicated in the figures.

Figure 5.4 and **Figure** 5.5 show the performance of block writes and block reads respectively. Note that, in both the figures bandwidth increases with the number of clients. We think the large stripe size of the RAID used, which results in parallelism, is the reason for this. The total CPU utilization in 2 and 3 clients case is twice and thrice the CPU utilization in 1 client case respectively. Note that, unlike Postmark benchmark, here each client creates only one file and the size of the file is much larger than the files created by the Postmark.

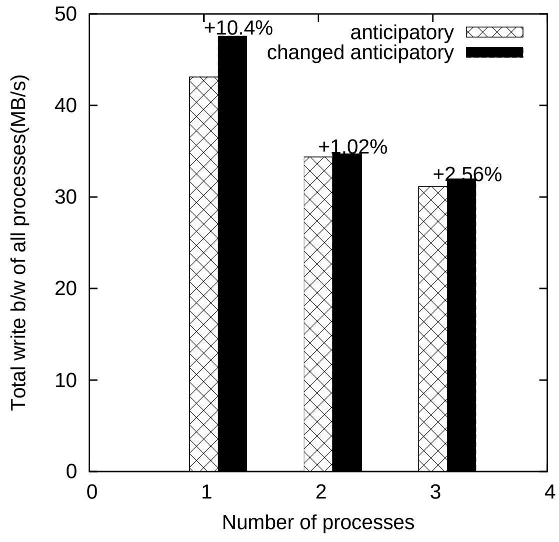

Figure 5.3: Postmark benchmark, write performance

There is no significant difference between anticipatory and changed anticipatory scheduler during create/delete tests. The number of cache hits as reported by the generated module is very low. This is because the locality part has been taken care of by the file level and block device level prefetching of Linux kernel.

5.4 Andrew Benchmark

We ran a modified version of Andrew Benchmark to measure the performance change with our scheduler. In phase 1, the benchmark creates all directories present in Linux 2.6.14 kernel source code. In phase 2, the benchmark copies all files into the directories. In phase 3, recursive stat is done on all files and directories. In phase 4, all files are scanned. In phase 5, the kernel and modules are built. The plotted values are average of three trials. The standard deviation between different trials is negligible. The time taken by different clients in the same trial are almost same.

Figure 5.6 shows the performance of the modified Andrew benchmark. We observed almost no difference in performance between anticipatory and changed anticipatory. The CFQ and deadline schedulers also performed as good as anticipatory. We also noticed that the number of cache hits as reported by the module generated is also very low. Even disabling the block device level prefetching of Linux kernel did not result in performance loss. This suggests that the locality of reference at the block device level for this benchmark is low.

5.5 Benchw

Benchw[Benchw] measures the performance of data loading, index creation and queries in the spirit of TPC-H. Benchw comes with a tool called querygen that generates five different

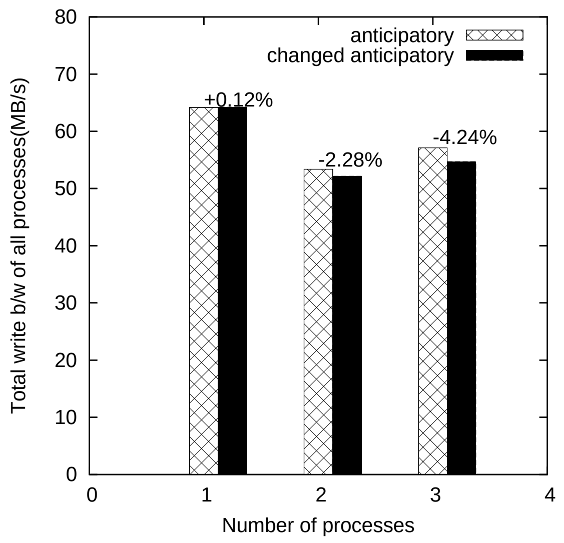

Figure 5.4: Bonnie benchmark, write performance(written in blocks)

queries. **Figure** 5.7 shows the performance of changed anticipatory and anticipatory schedulers for these five queries generated by benchw. The data files using which the tables were populated are of varying sizes with largest file being 1GB. In experiment done, two different clients were executing queries on two different databases being served by two different Postgresql database servers. We found upto 3 times improvement in performance for some queries. There were considerable number of cache hits(around 15that the performance gain doesn't come from the cache hits alone. As mentioned earlier, there are scenarios in which we go ahead with next request rather than anticipating. This also contributes to the performance gain.

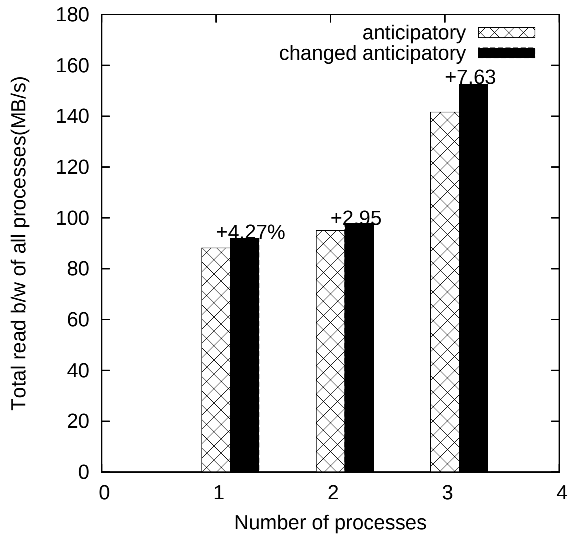

Figure 5.5: Bonnie benchmark, read performance(read in blocks)

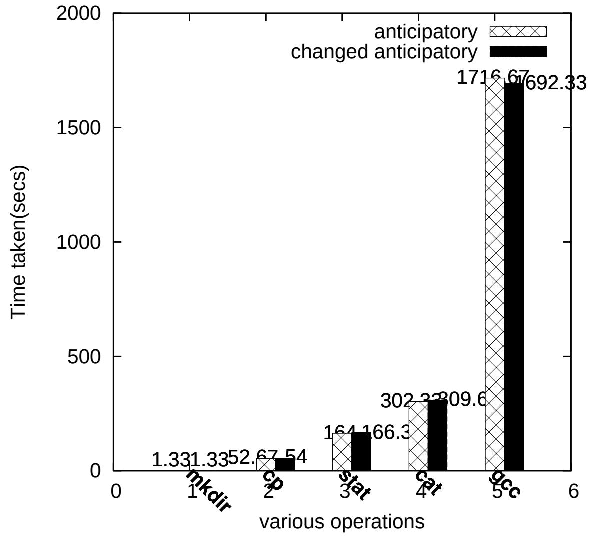

Figure 5.6: Andrew benchmark, 1 client performance(time taken in secs)

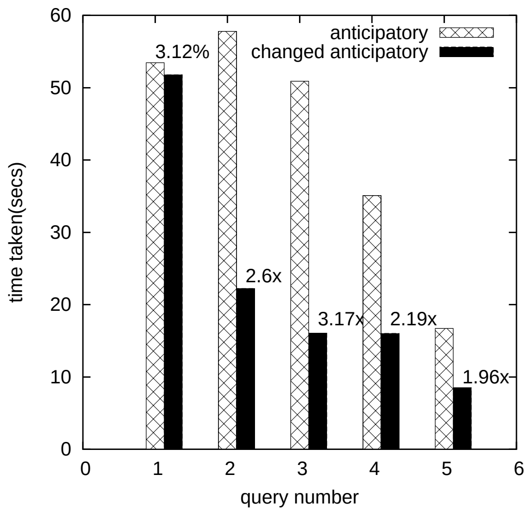

Figure 5.7: benchw benchmark, two clients using two different database servers

Chapter 6

Related Work

DIXtrac [Schindler99] is a tool to automatically characterize disk drives. These parameters can be fed into DiskSim [Ganger98] to simulate the disk.Their work only considers non-adaptive caching algorithms. The tool exits if it discovers that the cache controller is running an adaptive caching algorithm. Most of the algorithms we have used are very similar to ones they have used. However, the state space(the model of disk cache behavior) in which we are searching for a solution is different from the one in which they are searching for a solution. Moreover, we are only interested in disk subsystems with large caches which most probably would be designed for server workloads. Apart from the underlying device cache parameters, their tool also automatically deduces various other disk parameters that are required to simulate a disk. Extraction of SCSI disk drive parameters is also described in [Worthington95]. Disk drive modeling is described in [Ruemmler94]. In [Talagala00], the authors change micro benchmark to efficiently extract a subset of disk geometry and performance parameters.Automatic de-construction of RAID devices is discussed in [Denehy04]. They used x-means [Pelleg02] algorithm to recognize various RAID parameters such as RAID level, stripe size etc. The method we used to recognize a cache hit is to compare with maximum disk bandwidth which is calculated from the device supplied parameters. Their tool fires requests so that the effects of the device cache are nullified, whereas our goal is to find the behavior of disk cache. There are several works[Burnett02, Schindler02] in which information at lower layers is automatically deduced and made available to the upper layers.

The current hierarchy of caching and prefetching in a system running 2.6 Linux kernel is shown in **Figure** 6.1. The caches involved are inclusive caches; data would be cached both at the disk controller level and the file system level. One way of making caching exclusive is described in [Wong02]. The introduce a new disk device level command 'DEMOTE' to move buffer from the file system level to the disk cache. Modern disk subsystems so far, don't support this operation. However, in case an NBD(Network Block Device) is being used, one can change the NBD server interface to support this operation. We think this combined with a device-cache-aware scheduler could result in better performance and also be space efficient. The design considerations and performance evaluation of disk device cache are also discussed in [Smith85]. There are several other works [Lee97, Forney02] in which authors either to improve performance or eliminate double caching by the modifying the buffer cache.

Deceptive idleness and anticipatory scheduling as a way to overcome it are discussed in [Iyer01a, Iyer01b]. They found lot of performance improvement over disks used in desktop

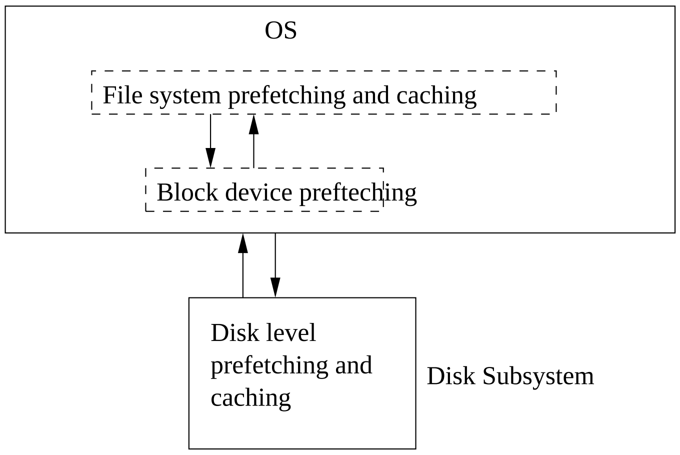
OS
File system prefetching and caching
Block device preftechjing
Disk level prefetching and caching
Disk Subsystem

Figure 6.1: Caching and prefetching hierarchy in Linux kernel

environments and over a enterprise class SCSI disk. However, in case of disk subsystems with large caches, there are scenarios in which we found that the Linux deadline scheduler performs better than the anticipatory scheduler. But the device-cache-aware anticipatory scheduler performed better than these schedulers(deadline and CFQ) on disks with large caches. Moreover,deadline and CFQ can be also be made device-cache-aware; we have not done this so far. Some other works in disk scheduling area are in cited below.

Freeblock scheduling [Lumb00], has been proposed to increase disk bandwidth utilization by potentially servicing asynchronous requests enroute to the synchronous ones. YFQ, a scheduling algorithm that allows application to set aside for exclusive use of the disk bandwidth is described in [Bruno99]. Various proportional share schedulers are also introduced in [Walspurger95, Waldspurger94]. The authors propose and evaluate several disk scheduling optimizations that enhance the aggregate disk throughput. In [Eggert05], the authors present a generic kernel-level mechanism to use the idle resource capacity in the background without slowing down the foreground use, which they have used in FreeBSD disk scheduler. A rotation latency sensitive disk scheduler is described in [Huang00]. They proposed a mechanism to predict the current disk head position, and heuristics to estimate the disk service time for requests, using which they have implemented their scheduler.

Performance of various disk scheduling algorithms are evaluated in [Seltzer90, Worthington94]. The authors of [Worthington94] also discuss efficient use of prefetching done by the device cache in scheduling algorithms like SPTF. This involves keeping track of contents of prefetch buffers and the disk head position. Modern disk subsystems also cache the previously prefetched data for some time before they are evicted. As far as we know, they have not tried to make use of this fact in their work. Variants of CSCAN and SATF are evaluated in [Thomasian02]. A two level scheduling framework, consisting of application class independent scheduler and a set of application class specific schedulers, is described in [Shenoy98].

Chapter 7

Conclusions and Future work

In this work, we proposed a scheduling architecture that is aware of underlying device cache. We have also shown how to automatically extract the parameters of adaptive caching algorithms used in some of the modern storage devices and incorporate the same information into the scheduling algorithm. We have implemented a prototype of the scheduler architecture using Linux anticipatory scheduler, where we observed upto 3 times improvement in query execution times for Benchw benchmark and upto 10% improvement in bandwidth with Postmark benchmark.We intend to validate the cache profiler on more high end disks. We also intend to study adaptive caching algorithms used in disk drives through trace driven simulations and study the impact of various cache parameters such as prefetch buffer size, number of cache segments, cache replacement policy on the performance.

Bibliography

[Schindler99] J. Schindler and G. R. Ganger. Automated disk drive characterization. Technical Report CMU-CS-99-176, Carnegie Mellon University, 1999.

[Ganger98] G. R. Ganger, B. L. Worthington, and Y. N. Patt. The DiskSim simulation environment. Technical Report CSE-TR-358-98, Dept. of Electrical Engineering and Computer Science, Univ. of Michigan, February 1998.

[Denehy04] Timothy E. Denehy, John Bent, Florentina I. Popovici, Andrea C. Arpaci-Dusseau and Remzi H. Arpaci-Dusseau. Deconstructing Storage Arrays, Proceedings of the 11th international conference on Architectural support for programming languages and operating systems, Pages 59-71, Year of Publication: 2004, ISSN:0362-1340.

[Burnett02] Nathan C. Burnett, John Bent, Andrea C. Arpaci-Dusseau and Remzi H. Arpaci-Dusseau. Exploiting Gray Box Knowledge of Buffer-Cache Management, Proceedings of the General Track: 2002 USENIX Annual Technical Conference, p 29-44, June 10-15, 2002.

[Bonnie] T. Bray. The Bonnie File System Benchmark. http://www.textuality.com/bonnie

[Katcher97] J. Katcher. Postmark: A New File System Benchmark. Technical Report TR-3022, Network Appliance Inc., Oct 1997.

[Pelleg02] Dan Pelleg and Andrew W. Moore. X-means: Extending K-means with Efficient Estimation of the Number of Clusters, Proceedings of the Seventeenth International Conference on Machine Learning, p.727-734, June 29-July 02, 2000.

[Schindler02] Jiri Schindler, John Linwood Griffin, Christopher R. Lumb and Gregory R. Ganger. Track-Aligned Extents: Matching Access Patterns to Disk Drive Characteristics, Proceedings of the Conference on File and Storage Technologies, p.259-274, January 28-30, 2002.

[Talagala00] N. Talagala, R. Arpaci-Dusseau and D. Patterson. Micro-Benchmark Based Extraction of Local and Global Disk, University of California at Berkeley, Berkeley, CA, 2000.

[Worthington95] Bruce L. Worthington, Gregory R. Ganger, Yale N. Patt and John Wilkes. On-line extraction of SCSI disk drive parameters, Proceedings of the 1995 ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems, p.146-156, May 15-19, 1995, Ottawa, Ontario, Canada.

[Smith85] Alan J. Smith. Disk cache-miss ratio analysis and design considerations, ACM Transactions on Computer Systems(TOCS), Volume 3, Issue 3 (August 1985), Pages: 161-203, Year of Publication: 1985, ISSN:0734-2071

[Arpaci-Dusseau01] Andrea. C. Arpaci-Dusseau and Remzi. H. Arpaci-Dusseau. Information and control in gray-box systems, Proceedings of the eighteenth ACM symposium on Operating systems principles, Pages: 43-56, Year of Publication: 2001, ISBN: 1-58113-389-8

[Lee97] Donghee Lee, Sam H. Noh, Sang Lyul Min and Yookun Cho. Efficient Caching Algorithms for Two-level Disk Cache Hierarchies, Proceedings of the 8th Annual Symposium on Combinatorial Pattern Matching.

[Forney02] Brian C. Forney, Andrea C. Arpaci-Dusseau and Remzi H. Arpaci-Dusseau. Storage-Aware Caching: Revisiting Caching for Heterogeneous Storage Systems, Proceedings of the 1st USENIX Conference on File and Storage Technologies, SESSION: Performance and Modeling, Article No. 5, Year of Publication: 2002.

[Bruno99] J. Bruno, J. Brustoloni, E. Gabber, B.Özden, and A. Silberschatz. Disk scheduling with quality of service guarantees. In IEEE ICMCS, June 1999.

[Huang00] L. Huang and T. Chiueh. Implementation of a rotation latency sensitive disk scheduler. Technical Report ECSL-TR81, SUNY, Stony Brook, Mar. 2000.

[Iyer01a] S. Iyer and P. Druschel. The effect of deceptive idleness on disk schedulers. Technical Report CSTR01-379, Rice University, June 2001.

[Iyer01b] S. Iyer and P. Druschel. Anticipatory scheduling: a disk scheduling framework to overcome deceptive idleness in synchronous I/O, Proceedings of the eighteenth ACM symposium on Operating systems principles, Pages: 117-130, Year of Publication: 2001, ISBN: 1-58113-389-8.

[GNU] http://www.gnu.org/

[kernel] http://www.kernel.org/

[Lumb00] C. Lumb, J. Schindler, G. Ganger, D. Nagle, and E. Riedel. Towards higher disk head utilization: Extracting free bandwidth from busy disk drives. In 4th USENIX OSDI, Oct. 2000.

[Seltzer90] M. Seltzer, P. Chen, and J. Ousterhout. Disk scheduling revisited. In USENIX Winter Technical Conference, Jan. 1990.

[Shenoy98] P. Shenoy and H. Vin. Cello: A disk scheduling framework for next generation operating systems. In ACM SIGMETRICS, June 1998.

[Ruemmler94] C. Ruemmler and J. Wilkes. An introduction to disk drive modeling. IEEE Computer, 27(3):17-28, 1994.

[Worthington94] B. Worthington, G. Ganger, and Y. Patt. Scheduling algorithms for modern disk drives. In ACM SIGMETRICS, 1994.

[Walspurger95] C. Waldspurger and W. Weihl. Stride scheduling: Deterministic proportional resource management. Technical report, MIT/LCS/TM-528, June 1995.

[Waldspurger94] C. Waldspurger and W. Weihl. Lottery scheduling: Flexible proportional-share resource management. In 1st USENIX OSDI, Nov. 1994.

[Thomasian02] A. Thomasian and C. Liu, Disk scheduling policies with lookahead, ACM SIGMETRICS Performance Evaluation Review, v.30 n.2, September 2002.

[Eggert05] L. Eggert, Joseph D. Touch, Idletime scheduling with preemption intervals, ACM SIGOPS Operating Systems Review, v.39 n.5, December 2005.

[Ext2] R. Card, T. Ts'o and S. Tweedie. Design and Implementation of the Second Extended Filesystem, Proceedings of the First Dutch International Symposium on Linux, ISBN 90-367-0385-9.

[torque] http://sg.torque.net/sg/

[proware] http://www.proware.com.tw/products/products.htm

[hds] http://www.hds.com/products_services/storage_systems/enterprise_storage/

[Cheetah36LP-FC] "Cheetah X15 36LP FC Disc Drive product Manual, Volume 1"

[Wong02] T. M. Wong, J. Wilkes, "My Cache or Yours? Making Storage More Exclusive", Proceedings of the General Track: 2002 USENIX Annual Technical Conference, 161-175, 2002.

[Benchw] "http://sourceforge.net/projects/benchw".

Part II

Primary Network Partition Selection in Clustered Shared Disk Systems

Abstract

In this work, we give an algorithm for fault-tolerant proactive leader election in asynchronous shared memory systems, and later its formal verification. Roughly speaking, a leader election algorithm is proactive if it can tolerate failure of nodes even after a leader is elected, and (stable) leader election happens periodically. This is needed in systems where a leader is required after every failure to ensure the availability of the system and there might be no explicit events such as messages in the (shared memory) system. Previous algorithms like DiskPaxos[Gafni00] are not proactive.

In our model, individual nodes can fail and reincarnate at any point in time. Each node has a counter which is incremented every period, which is same across all the nodes (modulo a maximum drift). Different nodes can be in different epochs at the same time. Our algorithm ensures that per epoch there can be at most one leader. So if the counter values of some set of nodes match, then there can be at most one leader among them. If the nodes satisfy certain timeliness constraints, then the leader for the epoch with highest counter also becomes the leader for the next epoch(stable property). Our algorithm uses shared memory proportional to the number of processes, the best possible. We also show how our protocol can be used in clustered shared disk systems to select a primary network partition. We have used the state machine approach to represent our protocol in Isabelle HOL[Nipkow02] logic system and have proved the safety property of the protocol.

Chapter 8

Introduction and Motivation

In certain systems, a leader is required after every failure to ensure progress of the system. This can be solved in shared memory systems by electing a leader every period. This problem admits a trivial solution: let $T \bmod n$ be the leader for T^{th} epoch, where n is the number of nodes. There are problems with this solution.

- Electing a different leader for each epoch is costly. For example, in clustered shared disk systems, recovery has to be done every time primary network partition changes. We need a "stable" leader election[Aquilera01]: failures of nodes other than the leader should not change the leader.
- Failed nodes are also elected as leaders in some epochs.

In this work, we use the concept of a proactive leader election in the context of asynchronous shared memory systems. In such an election, a leader renews a lease every so often. The stable leader election[Aquilera01], appropriate in asynchronous network systems, differs from proactive leader election, as in the latter no failures are needed to trigger leader election; leaders are "elected" on a regular beat. This is needed in the systems we are interested in such as clustered shared disk systems where there are no interrupts to notify events that happen through the shared medium; all the communication is through shared disks.

Network partition is a possibility in a clustered shared disk system. A primary network partition is selected to ensure consistency of data on the shared disk. This is ensured by using a rule such as: the number of nodes in the primary network partition is at least $\lfloor \frac{n}{2} \rfloor + 1$, where n is the total number of nodes in the cluster.

Another way of selecting a primary network partition is by the use of fence devices. Fencing is used in clustered shared disk systems to prevent nodes in the non-primary network partitions from accessing disks (to ensure mutual exclusion for shared resources such as disks). Examples of fencing methods are: reboot nodes not in primary partition, disable ports corresponding to nodes not in the primary partition in the network, etc. However, even in this case, there is still a chance that all nodes get fenced. Moreover, once a primary partition is selected, if the nodes in primary network partition fail, then the system comes to halt even if there are nodes that are alive in a different partition.

Consider a simple case of a network containing two nodes.

- Suppose we assign unequal weights to the nodes and require the total weight of nodes in the primary partition to be strictly greater than that of the smaller one. If the node with higher weight dies, the system comes to halt in spite of the node with lower weight being alive.

- Suppose fencing is used. Let us suppose fencing succeeds after a network partition and a primary partition is selected. Now if the only node in primary network partition dies, then the system comes to halt.

Networks with higher number of nodes can also run into similar situations.

In this work, we give an algorithm for fault-tolerant proactive leader election in asynchronous shared memory systems, and later its formal verification. For example, in the second case, because of leases used in our algorithm, even if the node in the primary partition dies and the other node has already been fenced, in the subsequent epochs, the previously fenced node will get elected as the leader and the system becomes available again. Our work has been inspired by Paxos[Lamport98] and DiskPaxos[Gafni00] protocols; however, these protocols do not have the lease framework[1] incorporated into them.

We assume a model in which individual nodes can fail and reincarnate (this happens if the fencing method is reboot in clustered shared disk systems) at any point in time. Nodes have access to reliable shared memory[2]. There is no global clock accessible to all the nodes, but each node has access to a local counter which is incremented every T secs (whose accuracy has to be within the limits of a drift parameter) and restarts the protocol; for the current leader, this is extending the lease. The set of nodes which have the same counter value are said to be in the same epoch. It is quite possible that at a given instant, different nodes are in different epochs. In this work, we propose a protocol that elects a leader for each epoch. We guarantee that there is at most one leader per epoch. Moreover, if the nodes in the system satisfy certain timeliness conditions, then the leader for the epoch with highest counter value also becomes the leader for the next epoch.

The rest of the work is organized as follows. In chapter 9, we describe the related work. In chapter 10, we describe the model and algorithm informally and also give details of the algorithm. In chapter 11, we discuss the encoding in Isabelle and the main invariant used in proving the safety property. In chapter 12, we discuss implementation issues. And we conclude with chapter 13.

[1]The Paxos paper mentions the use of leases but no details are given.

[2]We believe this not to be a serious constraint. There are methods for implementing fault tolerant wait free objects from atomic read/write registers[Jayanti92]; they can be used for constructing reliable shared memory. Also, in clustered shared disk systems, shared "memory" can be realized with some extra effort by using hot swappable mirrored disk (RAID) devices with hot spares.

Chapter 9

Related Work

It is is well-known that there is no algorithm to solve consensus in asynchronous systems [Fischer85, Dolev87, Loui87]. Failure detectors have been proposed to solve the problem in weaker models. The failure detector Ω for asynchronous network systems can be used to implement a leader oracle. Roughly speaking, the leader oracle running at each node outputs a node which it thinks is operational at that point in time. Moreover, if the network stabilizes after a point, then there is a time after which all operational nodes output the same value. Implementations of the the leader oracle and failure detectors in asynchronous network systems augmented with different kinds of assumptions are described in several works including [Lamport98, Prisco97, Larrea00, Chu98].

The leader oracle (Ω) augmented with view numbers introduced in [Aquilera01] is similar to the problem considered here. Here, in addition, a view changes if either the current leader, or the network links or both do not satisfy certain timeliness conditions. But we are interested in asynchronous shared memory systems that are more suited for clustered shared disk systems. Consensus in asynchronous shared memory systems with various failure detectors is studied in [Wai-Kau94]. DiskPaxos[Gafni00], which has inspired this work, is similar to our protocol except that it does not have the lease framework.

Light weight leases for storage centric coordination is introduced in [Chockler04] that requires $O(1)$ shared memory (independent of the number of nodes). Their work assumes a model similar to the timed asynchronous model [Cristian98] with the safety property involving certain timeliness conditions. We prove the safety property of our protocol assuming asynchronous shared memory model but it requires n units of shared memory. This matches the lower bound in Chockler and Malkhi[Chockler02], where they introduce a new abstract object called *ranked register* to implement the Paxos algorithm and show that it cannot be realized using less than n read/write atomic registers.

Chapter 10

The Algorithm

10.1 An Informal Description of the Model and Algorithm

We consider a distributed system with $n > 1$ processes. Each node has a area allocated in the shared memory (actually, a shared disk) to which only it can write and other nodes can read. Processes can fail and reincarnate at any point in time. We call a system stable[1] in $[s, t]$ if at all times between s and t the following hold:

- The drift rate between any two nodes or drift rate of any node from real time is bounded by δ.
- The amount of time it takes for a single node to read a block from the shared memory and process it or write a block to the shared memory is less than r secs.
- The time it takes to change the state after a read or write is negligible compared to r.

We require the second assumption, because in our case shared disk serves as the shared memory. We assume that the system is stable infinitely often and for a sufficient duration so that leader election is possible. We assume that each of the nodes know the value of δ and r. Each node has access to a local timer which times out every T secs. We assume $T >> 3nr(1 + \delta)$; this will be motivated later.

The counter value of each node is stored in local memory as well in the shared memory; it is first written to the shared memory area and then written to the local memory. The counter values of all nodes are initialized to 0. When a process reincarnates, it reads the counter value from its shared memory area and then starts the timer. When the timer at a node expires, it increments its counter value and restarts the timer.

Each node is associated with a node id which is drawn from the natural number set. We assume that each node knows the mapping *nodeid* between the shared memory addresses and the node ids. We assume that the shared memory is reliable[1].

10.1.1 Safety Property

The safety property of the protocol requires that if a node with counter value v becomes leader, the no other node with counter value v ever becomes leader.

[1]Please note that this is different from the meaning of stable in "stable leader election". Context should make clear what is being meant.

10.1.2 Informal Description of Algorithm

Each block in shared memory allocated to a node consists of a counter value, a ballot number and proposed leader node id. When the counter of a node is incremented, it starts the protocol for the new epoch. During each of the phases, if a node finds a block with higher counter value, it sleeps for that epoch.

- In Phase 0, each node reads the disk blocks of all other nodes and moves to phase 1.
- In Phase 1, a node writes a ballot number to the disk. It chooses this ballot number that is greater than any ballot number read in the previous phase. If none of the blocks read after writing to the disk have a higher ballot number, the node moves to phase 2. If the node finds a block with higher ballot number, it restarts Phase1. This phase can viewed as selecting a node which proposes the leader node id.
- In Phase 2, a node proposes the node id of the leader and writes it to disk. This value is chosen so that all nodes that have finished the protocol for the current epoch agree on the same value. If none of the blocks read after writing the proposed value to the disk have a higher ballot number, the node completes the protocol for this epoch. If there is a block with a higher ballot number, it goes back to Phase1. If the proposed value is same as that of this node id, this node is the leader. Otherwise, it sleeps for this epoch.

10.2 Detailed description of the Algorithm

Each node's status ($nodestatus$) can be in one of the five states: *Suspended, Dead, Leader, PreviousLeader, Participant*.

- A node is in *Suspended* state, if it withdrew from the protocol for the current epoch.
- A node is in *Dead* state, if it has crashed.
- A node is in *Leader* state, if it is the leader for the current epoch.
- A node is in *PreviousLeader* state, if it was the leader for the previous epoch and is participating in the protocol for the current epoch.
- A node is in *Participant* state, if it is participating in the protocol for the current epoch and is not the leader for the previous epoch.

A block in the shared memory location allocated to a node is of form *(ctrv_d, pbal, bal, val)*, where *ctrv_d* is the counter value of the node (in the shared memory (actually, *disk*)) which has write permission to it, *pbal* is the proposed ballot number of that node, and *bal* is equal to the proposed ballot number *pbal* for which *val* was recently set. After each read or write to the shared memory, depending of whether some condition holds or not, the system moves from one phase to another. In each of the phases, *phase0*, *phase1* and *phase2*, before reading the blocks from the shared memory, each node clears its existing blocks read in the previous phase. To make our algorithm concise, we have used the phrase *"Node n rereads disk blocks of all nodes"* in each of the phases; this operation need not be atomic in our model.

We use $disk\ s\ n$ to represent the block of node n in the shared memory in state s. Also, let *blocksRead* $s\ n$ represent the blocks of nodes present at node n in state s. The state of the system is made up of: *state* of each of the nodes, *blocks* of each of the nodes in shared memory, *phase* of each of the nodes, the counter value of each of the nodes and *blocksRead* of each of the nodes.

Let $A(disk\ s\ n)$ denote the projection of component A of block $disk\ s\ n$. For example, $pbal(disk\ s\ n)$ denotes the $pbal$ component of block $disk\ s\ n$. Similarly, $A(B :: set)$ denotes the set composed of projection of component A of all blocks in set B. We use $ctrv\ s\ n$, $nodestatus\ s\ n$ and $phase\ s\ n$ to denote the counter value, state and phase of node n in state s respectively. Note that $ctrv$ is the value at the node whereas $ctrv\_d$ is the value at the disk.

There is an implicit extra action in each phase (omitted in the given specification for brevity): $Phase\{i\}Read\ s\ s'\ n\ m$, which says that node n in *phase*$\{i\}$ reads the block of node m and the system moves from state s to state s'. State variables not mentioned in a state transition below remain unchanged across the state transition.

For facilitating the proof, we use a history variable *LeaderChosenAtT*, but actually not needed in the algorithm: *LeaderChosenAtT s t* = *k* if *k* is the leader for epoch *t* in state *s*. Also, $LeaderChosenAtT\ s'\ t = LeaderChosenAtT\ s\ t$ unless the value for *t* is changed explicitly.

Phase0

<u>Action:</u> Node n (re)reads disk blocks of all nodes including itself.
Changes the state to *Suspended* if there exists a
node with higher counter value. Otherwise, moves to
phase 1. Formally,

<u>Case:</u> $\exists\ br \in blocksRead\ s\ n.\ ctrv\_d(br) > ctrv\ s\ n$
<u>Outcome:</u> $nodestatus\ s'\ n = Suspended.$

<u>Case:</u> $\neg\exists\ br \in blocksRead\ s\ n.\ ctrv\_d(br) > ctrv\ s\ n.$
<u>Outcome:</u> $phase\ s'\ n = 1$

Phase1

<u>Action:</u> Write a value greater than *pbal*s of
all blocks read in previous phase to the disk. Formally,
$disk\ s'\ n = (ctrv\ s\ n,\ pbal',\ bal(disk\ s\ n),\ val(disk\ s\ n))$
where $pbal' = \text{Max}(pbal(blocksRead\ s\ n)) + 1.$
Node n rereads disk blocks of all the nodes.
If there exists a block with higher counter value, move
to *Suspended* state. If there exists a block with
higher *pbal*, restart phase 1. Otherwise, move to
phase 2. Formally,

<u>Case:</u> $\exists\ br \in blocksRead\ s\ n.\ ctrv\_d(br) > ctrv\ s\ n.$
<u>Outcome:</u> $nodestatus\ s'\ n = Suspended.$

<u>Case:</u> $\exists\ br \in blocksRead\ s\ n.\ ctrv\_d(br) = ctrv\ s\ n$
$\&\ ((pbal(br) > pbal(disk\ s\ n))$
$|\ (pbal(br) = pbal(disk\ s\ n)$

$\& \; nodeid(br) \; > \; n))$

<u>Outcome:</u> Node n restarts *Phase1*.

<u>Case:</u> $\neg\exists \; br \; \in \; blocksRead \; s \; n. \; ctrv\_d(br) \; > \; ctrv \; s \; n$
$\quad | \; ((ctrv\_d(br) \; = \; ctrv \; s \; n)$
$\quad\quad \& \; (pbal(br) \; > \; pbal(disk \; s \; n)))$
$\quad | \; ((ctrv\_d(br) \; = \; ctrv \; s \; n)$
$\quad\quad \& \; (pbal(br) \; = \; pbal(disk \; s \; n))$
$\quad\quad \& \; (nodeid(br) \; > \; n))$

<u>Outcome:</u> $phase \; s' \; n \; = \; 2$.

Phase2

<u>Action:</u> Write the proposed leader node id to the disk, where the node id is chosen as follows: if no other node with same counter value has proposed a value, set it to this node id; otherwise, set it to the value of the block with highest *bal* whose proposed value is non-zero. Formally,

$disk \; s' \; n \; = \; (ctrv \; s \; n, \; pbal(disk \; s \; n), \; pbal(disk \; s \; n), \; proposedv)$

where $proposedv \; =$

n if $(\forall \; br \; \in \; blocksRead \; s \; n. \; ctrv\_d(br) \; = \; ctrv \; s \; n$
$\quad\quad \longrightarrow \; val(br) \; = \; 0)$

else

m where $(m \; = \; val(br)$
$\quad \& \; bal(br) \; = \; \text{Max}(bal(\{br| \; br \; \in \; blocksRead \; s \; n$
$\quad\quad \& \; ctrv\_d(br) \; = \; ctrv \; s \; n$
$\quad\quad \& \; val(br) \; \neq \; 0\})))$

Node n rereads the blocks of all the nodes.
If there exists a node with higher counter value, move to *Suspended* state. If there exists a node with higher *pbal* restart from phase 1. Otherwise, if the proposed node id is same as the id of this node, this node is the leader. If the proposed node id is not same as the id of this node, move to *Suspended* state. Formally,

<u>Case:</u> $\exists \; br \; \in \; blocksRead \; s \; n. \; ctrv\_d(br) \; > \; ctrv \; s \; n.$

<u>Outcome:</u> $nodestatus \; s' \; n \; = \; Suspended.$

<u>Case:</u> $\exists \; br \; \in \; blocksRead \; s \; n. \; ctrv\_d(br) \; = \; ctrv \; s \; n$
$\quad \& \; ((pbal(br) \; > \; pbal(disk \; s \; n)$
$\quad\quad | \; (pbal(br) \; = \; pbal(disk \; s \; n)$
$\quad\quad\quad \& \; nodeid(br) \; > \; n))$

<u>Outcome:</u> Node n restarts *Phase1*.

<u>Case:</u> $\neg\exists \; br \; \in \; blocksRead \; s \; n. \; ctrv\_d(br) \; > \; ctrv \; s \; n$
$\quad | \; ((ctrv\_d(br) \; = \; ctrv \; s \; n)$
$\quad\quad \& \; (pbal(br) \; > \; pbal(disk \; s \; n))$
$\quad | \; ((ctrv\_d(br) \; = \; ctrv \; s \; n)$

```
                              & $(pbal(br) \;=\; pbal(disk\ s\ n))$
                              & $(nodeid(br) \;>\; n))$
        Outcome: if $val(disk\ s\ n) \;=\; n$
                 then $nodestatus\ s'\ n \;=\; Leader$
                      & $LeaderChosenAtT\ s'\ (ctrv\ s\ n) \;=\; n$
                 else $nodestatus\ s'\ n \;=\; Suspended).$
```

Fail

```
        Outcome: $nodestatus\ s'\ n \;=\; Dead$
```

ReIncarnate

```
        Outcome: $nodestatus\ s'\ n \;=\; Suspended$
```

IncrementTimer

*Case:*Node n timer expires.
If this node is the leader in previous epoch, update the counter value on disk and move to phase 2. Otherwise, update the counter value on disk, reset *bal* and *val* on disk and move to phase 0.Formally,

```
        Outcome: if $nodestatus\ s\ n \;=\; Leader$
                 then $nodestatus\ s'\ n \;=\; PreviousLeader$
                      $disk\ s'\ n \;=$
                      $(ctrv\ s\ n \;+\; 1,\ pbal(disk\ s\ n),\ bal(disk\ s\ n),$
                      $val(disk\ s\ n))$
                      & $ctrv\ s'\ n \;=\; ctrv\ s\ n \;+\; 1$
                      & $phase\ s'\ n \;=\; 2$
                 else $nodestatus\ s'\ n \;=\; Participant$
                      & $disk\ s'\ n \;=\; (ctrv\ s\ n \;+\; 1,\ pbal(disk\ s\ n),\ 0,\ 0)$
                      & $ctrv\ s'\ n \;=\; ctrv\ s\ n \;+\; 1$
                      & $phase\ s'\ n \;=\; 0$
```

Note that with our protocol, it is quite possible that a particular block on disk and the block corresponding to it in $blocksRead$ of some node do not match. But this doesn't compromise the safety property mentioned below. However, for any node n, if the block corresponding to $disk\ s\ n$ is in its $blocksRead$, it will be same as that of $disk\ s\ n$.

safety property: $LeaderChosenAtT\ s\ t \neq 0 \longrightarrow$
$$\forall\, s',\, m.\; ((\, m \;\neq\; LeaderChosenAtT\ s\ t \;\&\; ctrv\ s'\ m \;=\; t) \longrightarrow nodestatus\ s'\ m \neq\; Leader)$$

The safety property of the protocol says that if a node with counter value *T* becomes leader then no other node with counter value *T* ever becomes leader.

To ensure liveness of the protocol in Timed Asynchronous Model, one can use leader election oracle mentioned in [Chockler04] with Δ equal to T, and δ equal to r, to choose a node in *IncrementTimer*. If a node is not elected by the leader oracle, it sleeps for approximately $3nr(1+\delta)$ secs and then starts the protocol. If the leader oracle succeeds in electing a single leader, that particular node has to write to its block and read all other blocks, at most thrice, so the execution would take at most $3nr(1+\delta)$ in a stable period. Actually, this number can be reduced to $(n+1)r(1+\delta)$, if the output of the leader oracle in previous epoch is same as that of leader's id for current epoch. This is because the previous epoch leader directly moves to *phase2* after it increments its timer. So, if no other node is in *Participant* state when the previous epoch leader is participating in the protocol, it at most has to write to its block twice and read blocks of all other nodes once. This would take at most $(n+1)r(1+\delta)$ secs in a stable period. While proving the safety property of the protocol in asynchronous model, the timing constraints are not required. Hence, in the actual specification of the algorithm in Isabelle, we have not encoded the timing constraints.

Chapter 11

Encoding in Isabelle and its proof

We have used the state machine approach to specify the protocol in Isabelle [Nipkow02]. The encoding of the state machine in Isabelle is similar to the one given in [sourceforgeDP]. Note that in *phase1* and *phase2*, we first write to the disk and then read the blocks of all nodes. Furthermore, in the specification of the algorithm above, we have used the phrase *"Node n restarts from phase1"*. We have realized this by associating a boolean variable $diskWritten$ with each node. We require it to be $true$, as a precondition for any of the cases to hold. When a node writes to the disk in *phase1* or *phase2*, it sets $diskWritten\ s'\ n$ to $true$ and sets $blocksReads'\ n$ to empty set. In addition, we require all blocks to be read as a precondition for any of the cases to hold. And by a node n restarting from phase 1, we mean that $diskWritten\ s'\ n$ is set to $false$ and $phase\ s'\ n$ is set to 1.

In the specification of the protocol in Isabelle, we have three phases while we had only two phases in the informal description. This is not essential, but we have done it for better readability of the specification. In the 3^{rd} phase, a node does nothing except changing its state to *Leader* or *Suspended*. Furthermore, in the specification for *phase0*, we deliberately split the case 2 of *phase0* into two cases anticipating optimizations later.

The proof is by method of invariants and bottom-up. However unlike [Gafni00], the only history variable we have used is *LeaderChosenAtT*, where *LeaderChosenAtT(t)* is the unique leader, if any exists, for the epoch *t*, otherwise it is zero. The specification of the protocol, the invariants used and the lemmas can be found in [MCD06]. The proof of the lemmas is quite straightforward, but lengthy because of the size of the protocol.

The main invariant used in the proof is the $AFTLE\_INV4$ & $AFTLE\_INV4k$. $AFTLE\_INV4$ requires that, if a node is in phase greater than 1 and has written its proposed value to the disk, then either $MaxBalInp$ is true or there exists a block br, either in $blocksRead$, or on disk which it is about to read, which will make this node to restart from $phase1$. Formally,

$$
\begin{array}{ll}
AFTLE\_INV4\ s\ \equiv & \\
& \forall\, p.\ ((phase\ s\ p\ >=\ 2) \\
& \qquad \&\ (diskWritten\ s\ p\ =\ True))\ \longrightarrow \\
& ((MaxBalInp\ s\ (bal(disk\ s\ p))\ p\ val(disk\ s\ p)) \\
& \mid (\exists\, br.\ ((br\ \in\ blocksRead\ s\ p) \\
& \qquad \&\ Greaterthan\ br\ (disk\ s\ p))) \\
& \mid (\exists\, n.\ ((\neg hasRead\ s\ p\ n) \\
& \qquad \&\ Greaterthan\ (disk\ s\ n)\ (disk\ s\ p)))
\end{array}
$$

where

- $Greaterthan\ br\ br' \ \equiv$
$$\begin{array}{l}((ctrv\_d(br) > ctrv\_d(br')) \\ \mid ((ctrv\_d(br) = ctrv\_d(br')) \\ \quad \&\ (pbal(br) > pbal(br'))) \\ \mid ((ctrv\_d(br) = ctrv\_d(br')) \\ \quad \&\ (pbal(br) = pbal(br')) \\ \quad \&\ (id(br) > id(br'))))\end{array}$$

- $MaxBalInp$ requires that, if the proposed value of node n is val, then any other node with same counter value as that of n and $(bal, nodeid)$ greater than that of node n, has val as its proposed value. Formally,

  $MaxBalInp\ s\ b\ m\ val \ \equiv$
$$\begin{array}{l}(\forall\ n.\ ((val > 0) \\ \qquad \&\ (ctrv\ s\ n = ctrv\ s\ m) \\ \qquad \&\ ((bal(disk\ s\ n) > b) \\ \qquad\quad \mid ((bal(disk\ s\ n) = b) \\ \qquad\qquad \&\ (n > m)))) \ \longrightarrow \\ \qquad val(disk\ s\ n) = val) \\ \&\ (\forall\ n.\ (\forall\ br.\ ((val > 0) \\ \qquad\qquad \&\ (br\ \in\ blocksRead\ s\ n) \\ \qquad\qquad \&\ (ctrv\ s\ m = ctrv\ s\ n) \\ \qquad\qquad \&\ (ctrv\_d(br) = ctrv\ s\ n) \\ \qquad\qquad \&\ ((bal(br) > b) \\ \qquad\qquad\quad \mid ((bal(br) = b) \\ \qquad\qquad\qquad \&\ (nodeid(br) > m))) \ \longrightarrow \\ \qquad\qquad val(br) = val)))\end{array}$$

- $hasRead\ s\ p\ q \ \equiv$
$$(\exists\ br\ \in\ (blocksRead\ s\ p).\ nodeid(br)\ =\ q)$$

$AFTLE\_INV4k$ requires that, if a node n is not the leader in previous epoch, then for any node distinct from n which is in phase greater than 1 and whose counter value is less than that of n, one of the following hold: its $pbal$ is less than $pbal$ of n, it moves to $Suspendedor\ Dead$ state, moves to phase 1. Formally,

$AFTLE\_INV4k\ s \ \equiv$
$$\begin{array}{l}\forall\ p.\ (\forall\ n.\ ((n \neq p) \\ \qquad \&\ (ctrv\ s\ n > ctrv\ s\ p) \\ \qquad \&\ (phase\ s\ p >= 2) \\ \qquad \&\ (diskWritten\ s\ p) \\ \qquad \&\ (val(disk\ s\ p) = p) \\ \qquad \&\ ((phase\ s\ n > 1) \\ \qquad\quad \mid ((phase\ s\ n = 1)\end{array}$$

$$\begin{array}{l} \quad\&\ (diskWritten\ s\ n)))) \ \longrightarrow \\ ((pbal(disk\ s\ n) > pbal(disk\ s\ p)) \\ \ |\ (pbal(disk\ s\ n) = pbal(disk\ s\ p) \\ \ \&\ (n > p)) \\ \ |\ (\exists\ br\ \in\ blocksRead\ s\ p.\ Greaterthan\ br\ (disk\ s\ p)) \\ \ |\ (\neg hasRead\ s\ p\ n))) \end{array}$$

First we proved that the invariant holds for the initial state and then we proved that if the invariant holds before a state transition, then it also holds after a state transition.

The first part of the invariant $AFTLE\_INV4$ is similar to the main invariant in [Gafni00]. The second part $AFTLE\_INV4k$ is new. We could not prove $AFTLE\_INV4$ by itself; we had to strengthen it by adding $AFTLE\_INV4k$ to be able to be prove it. The place where this invariant is needed is in *IncrementTimer*. The need for strengthening arises due to the one round optimization in the protocol. If a node A is the leader for epoch T, another node B is the leader for epoch $T + 1$ with *pbal* smaller than that of A's and A increments its counter value and moves to *Phase2*, then this invariant could be violated. This is what is ruled out by $AFTLE\_INV4k$. $ATFLE\_INV4k$ says that if node A is the leader for a particular epoch, then any node other than A, which has a counter value greater than that of A and which had written to the disk in *Phase1*, has *pbal* greater than that of A. We could not prove $AFTLE\_INV4k$ alone either. When *incrementTimer* event occurs, if two nodes with same *ctrv_d* are leaders in s, then this invariant could be violated. This is exactly what is ruled out by $AFTLE\_INV4$.

The following are the only assumptions we used in the proof, apart from the axioms that each of the possible values of *nodestatus* are distinct from one another. Let us denote the set of all *nodeids* by S.

$\boxed{S\ \neq\ \{\},\quad finite\,S,\quad s\ \in\ S\ \longrightarrow\ s\ \neq\ 0}$

Note that as a consequence, our protocol holds even if the number of nodes participating in the protocol is 1. But, in this case, leader election is trivial. We need the second assumption because we are often required to use the following rule which had that assumption as one of the premises.

$\boxed{finite\,A;\ A\ \neq\ \{\};\ x\ \in\ A\ \Longrightarrow\ x\ <=\ Max\ A}$

We have used HOL-Complex logic instead of just HOL logic of Isabelle anticipating use of *real set* later.

In the protocol specification, we chose *nodeids* from the natural number set. In spite of that, we had to state that none of the nodeid's is equal to 0 as a axiom. Furthermore, for each state transition, we had to mention the state variables that do not change along with those that change. There are some results which we could not prove using Isabelle, like

$\boxed{nodestatus\ s\ \neq\ (nodestatus\ s)(n\ :=\ Leader)\ \Rightarrow\ nodestatus\ s\ n\ \neq\ Leader}$

which was created during the proof of a lemma by a method named *auto*. In such cases, we had to backtrack to find a alternate path which avoids such a situation. Furthermore, we had to explicitly prove and pass certain results to the theorem prover because it could not recognize these patterns. (The method *auto* could prove these results.)

One such example is the following.

$\boxed{(\forall\ x\ \in\ P.\ Q(x))\ \Longrightarrow\ (\forall\ x.\ (x\ \in\ P)\ \longrightarrow\ Q(x))}$

More such examples can be found in the proof given in [MCD06].

Chapter 12

Selecting a Primary Network Partition in Clustered Shared Disk Systems

One can use the above protocol to select the primary network partition in clustered shared disk systems. In the following discussion, we assume that the nodes in the same network partition are loosely time synchronized, i.e., modulo the drift parameter. Once consensus is reached on node id of the leader, each node can check if the node id is present in its membership set. If it is present, it knows that it is part of the primary partition.

Note that in the asynchronous shared memory model, it is quite possible in our protocol that two different nodes in two different network partitions become leaders for different epochs at the same time instant (due to drift), although likely to happen only infrequently in practice. But, in clustered shared disk systems, once a primary partition is selected, nodes in network partitions other than the primary partition are fenced before the recovery is done. So even if two nodes from two different network partitions become leaders at the same time, at most one network partition would access the disk.

Moreover, with our protocol the number of leaders at any point in time is bounded by the difference between the highest counter value and the lowest counter value in the system. This value can be controlled (made closer to 1) by using time synchronization. This is also the maximum number of nodes which at a given point in time contend for the fence devices. Previously, the number of nodes contending for the fence devices is same as the number of the nodes. Reducing the contention on the fence devices increases the probability with which fencing suceeds.

With existing methods, fencing does not work always correctly. In the process of implementing the protocol on Redhat Cluster GFS, we realized that there is a way in which fencing can always be made to work with Brocade fibre channel switches[1] that allow only one admin telnet login at a time. So each node can login into every switch first in a predefined order (for example in the order in which they appear in the configuration file), then check if it has been fenced in any switch, if so logout from all switches and return a error; otherwise fence all the nodes which are not in its partition, unfence ones in its partition, and once finished then logout of all the switches. Although this method works with Brocade switches, it need not work in general. Note that even this method can fail if the only node

[1]Fibre Channel (FC) is a specialized data link layer for storage devices. FC switches are similar in function to gigabit ethernet switches.

in the current primary partition fails in a two node cluster.

Our protocol requires some set of disks to be outside the fencing domain which it can use as the shared memory. We think such a scenario is not rare because when different nodes are accessing different disks, no fencing is required between them. If fencing uses the Brocade switch property, when a leader gets elected for a new epoch, it can use the fencing method mentioned above with the modification that before returning an error it unfences itself. Example two node network and the fencing method is illustrated in **Figure** 12.1.

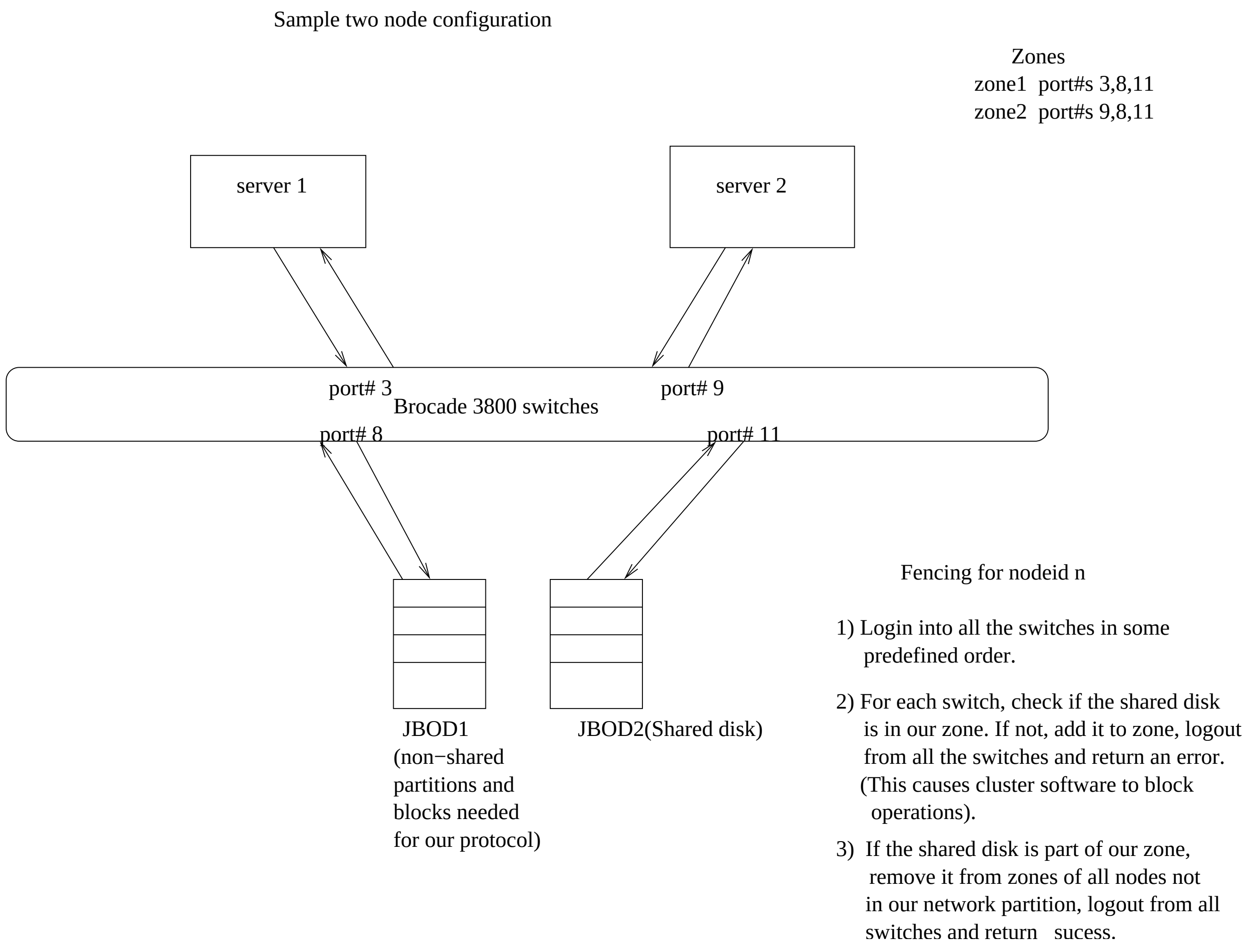
Sample two node configuration
Zones
zone1 port#s 3,8,11
zone2 port#s 9,8,11
server 1
server 2
port# 3
Brocade 3800 switches
port# 9
port# 8
port# 11
JBOD1 (non−shared partitions and blocks needed for our protocol)
JBOD2(Shared disk)
Fencing for nodeid n
1) Login into all the switches in some predefined order.
2) For each switch, check if the shared disk is in our zone. If not, add it to zone, logout from all the switches and return an error. (This causes cluster software to block operations).
3) If the shared disk is part of our zone, remove it from zones of all nodes not in our network partition, logout from all switches and return sucess.

Figure 12.1: Example two node cluster configuration

Note as a consequence of the safety property, if the highest *ctrv_d* in the system is *T*, then recovery/fencing would have been done at most *T* times. Furthermore, once a primary partition is selected and the leader in the primary partition is in the epoch with highest counter value and no more partitions/failures occur in the primary partition, then the *leader* for this epoch will be elected as the leader for the next epoch if the system is in stable period. In this case, fencing and recovery need not be done again. Furthermore, one more optimization that could be done in *Phase0*: when a node finds that there exists a block with higher *pbal* or same *pbal* from a higher node id, it changes its state to *Suspended*. We believe similar optimizations can be done in *Phase1* and *Phase2* too. But this would require that once a node is selected as a leader, it inform the nodes in its partition through the network which otherwise can be avoided assuming nodes in same network partition are (loosely) time synchronized.

Chapter 13

Conclusion

In this work, we have given a protocol for proactive leader election in asynchronous shared memory systems. We have specified the protocol and proved the safety property of the protocol using Isabelle [Nipkow02] theorem prover. We have also shown how one can use the protocol to choose a primary network partition in clustered shared disk systems. As a part of future work, we intend to specify the leader oracle protocol mentioned in [Chockler04] using Isabelle and also use it to prove the liveness property of our protocol with the the leader oracle in timed asynchronous model. We also intend to incorporate the fencing part in the protocol and prove its correctness. A prototype implementation is currently in progress and we intend to experimentally understand the relationship between δ, r and the number of nodes n. The complete theory files along with the technical report are accessible at *http://agni.csa.iisc.ernet.in/˜dharma/ATVA06/*.

Bibliography

[Gafni00] E. Gafni and L. Lamport. "Disk Paxos," In Proceedings of the International Symposium on Distributed Computing, pages 330-344,2000.

[Lamport98] L. Lamport. "The part-time parliament," ACM Transactions on Computer systems *16* (1998) 133-169.

[Nipkow02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. "Isabelle/HOL – A Proof Assistant for Higher-Order Logic," volume 2283 of LNCS. Springer, 2002.

[sourceforgeDP] "http://afp.sourceforge.net/browser_info/current/HOL/DiskPaxos/"

[Jayanti92] Prasad Jayanti, Tushar Deepak Chandra, and Sam Toueg. "Fault-tolerant wait-free shared objects," In Proceedings of the 33^{rd} Annual Symposium on Foundations of Computer Science, 1992.

[Chockler04] Chockler, Gregory and Dahlia Malkhi. "Light-Weight Leases for Storage-Centric Coordination," MIT-LCS-TR-934 Publication Date: 4-22-2004.

[Aquilera01] Marcos K. Aquilera, Carole Delporte-Gallet, Huques Fauconnier and Sam Toueg. "Stable Leader Election," In Proceedings of the 15^{th} International Conference on Distributed Computing, 2001. Pages: 108-122.

[Lampson96] Lampson, B. "How to build a highly available system using consensus," In *Distributed Algorithms*, ed. Babaoglu and Marzullo, LNCS 1151, Springer, 1996, 1-17.

[Prisco97] R. De Prisco, B. Lampson, and N. Lynch. "Revisiting the Paxos algorithm," In Proceedings of the 11^{th} Workshop on Distributed Algorithms(WDAG), pages 11-125,Saarbrücken, September 1997.

[Larrea00] M. Larrea, A. Fernández, and S. Arévalo. "Optimal implementation of the weakest failure detector for solving consensus," In Proceedings of the 19^{th} IEEE Symposium on Reliable Distributed Systems, SRDS 2000, pages 52-59, Nurenberg, Germany, October 2000.

[Chu98] F. Chu. "Reducing Ω to $\diamond W$," Information Processing Letters, 67(6):293-298, September 1998.

[Wai-Kau94] Wai-Kau Lo and Vassos Hadzilacos. "Using Failure Detectors to Solve Consensus in Asynchronous Shared-Memory Systems," In Proceedings of the 8^{th} International Workshop in Distributed Algorithms, 1994. Pages: 280-295.

[Fischer85] Michael J. Fischer, Nancy A. Lynch and Michael S. Paterson. "Impossibility of distributed consensus with one faulty process," Journal of the Association for Computing Machinery, 32(2): 374-382, April 1985.

[Dolev87] Danny Dolev, Cynthia Dwork and Larry Stockmeyer. "On the minimal synchronism needed for distributed consensus," Journal of the ACM, 34(1):77-97 , January 1987.

[Loui87] Michael C. Loui and Hosame H. Abu-Amara. "Memory requirements for agreement among unreliable asynchronous processes," In advances in Computer Research, volume 4, pages 163-183. JAI Press Inc., 1987.

[Chockler02] G. Chockler and D. Malkhi. "Active Disk Paxos with Infinitely Many Processes," Proceedings of the 21st ACM Symposium on Principles of Distributed Computing. (PODC), August 2002.

[MCD06] "http://agni.csa.iisc.ernet.in/~dharma/ATVA06/document.pdf"

[Cristian98] F. Cristian and C. Fetzer. "The timed asynchronous system model," in Proceedings of the 28th Annual International Symposium on Fault-Tolerant Computing, Munich, Germany, June 1998, pp. 140-149.

Part III

Cooperative caching in Redhat's Global File System

Abstract

The Redhat Global File System (GFS) is a clustered shared disk file system. The coordination between multiple accesses is through a lock manager. On a read, a lock on the inode is acquired in shared mode and the data is read from the disk. For a write, an exclusive lock on the inode is acquired and data is written to the disk; this requires all nodes holding the lock to write their dirty buffers/pages to disk and invalidate all the related buffers/pages. A DLM (Distributed Lock Manager) is a module that implements the functions of a lock manager. GFS's DLM has some support for range locks, although it is not being used by GFS.

While it is clear that a data sourced from a memory copy is likely to have lower latency, GFS currently reads from the shared disk after acquiring a lock (just as in other designs such as IBM's GPFS) rather than from remote memory that just recently had the correct contents. The difficulties are mainly due to the circular relationships that can result between GFS and the generic DLM architecture while integrating DLM locking framework with cooperative caching. For example, the page/buffer cache should be accessible from DLM and yet DLM's generality has to be preserved. The symmetric nature of DLM (including the SMP concurrency model) makes it even more difficult to understand and integrate cooperative caching into it. Note that GPFS has an asymmetrical design.

In this paper, we describe the design of a cooperative caching scheme in GFS. To make it more effective, we also have introduced changes to the locking protocol and DLM to handle range locks more efficiently. Experiments with micro benchmarks on our prototype implementation reveal that, reading from a remote node over gigabit Ethernet can be upto 8 times faster than reading from a enterprise class SCSI disk for random disk reads. Our contributions are an integrated design for cooperative caching and lock manager for GFS, devising a novel method to do interval searches and determining when sequential reads from a remote memory perform better than sequential reads from a disk.

Chapter 14

Introduction and Motivation

The Redhat Global File System (GFS)[GFS] is a clustered shared disk file system. It uses a locking interface that has been implemented by various lock managers. Redhat clusters' DLM (Distributed Lock Manager) implements this locking interface. DLM is a symmetric fault tolerant lock manager. The locking interface and DLM closely follow the implementation in VAX cluster[VCP93]. Whenever a read operation is performed on a file, a lock on the inode is acquired and the data is read from the disk. And, when data needs to be written to a file, the lock on inode is acquired in exclusive mode; this requires all nodes holding the lock on the file to flush dirty buffers/pages to the disk and invalidate the buffers/pages corresponding to this lock.

Using a global cache in addition to local buffer/page cache has two advantages; this global cache is realized by cooperative sharing of local memory of individual nodes in the whole system. One, if a read done happens to be in the global cache, it can be fetched directly from the global (remote or local) memory rather than from the disk, as disk reads can be slow. Two, if some node fails and the node recovers, the part of buffer/page cache contents which are present at the remote nodes need not be read from the disk, when a read on those contents happens. Also, in scenarios where the disk bandwidth is the bottleneck, cooperative caching can reduce the load on the disk, thus making it more scalable.

Integrating cooperative caching with existing buffer/page cache can be quite difficult. This is due to the cyclic dependencies introduced between GFS and DLM due to cooperative caching. For example, GFS calls into DLM to acquire/release locks and DLM needs to call into GFS to update/read buffer/page cache. All this should be achieved maintaining the generality of DLM. Due to the multiplicity of layers that interact with each other, a working design is non-trivial. As an example, tracking and accessing the buffer/page cache happens at many places: while updating page/buffer from the remote memory over network, while writing pages to the network, while flushing page/buffer cache to the disk, while reading from the disk and also while reclaiming the buffers/pages. Moreover, without proper tuning, sequential reads from remote memory can be slower than same reads from the disk and this is the reason why GPFS currently does not use cooperative caching[shah].

zFS[Rodeh03] is a clustered shared disk object-based file system, which also implements cooperative caching. However, zFS code is not symmetric; the transaction and lease managers running on separate nodes take care of concurrent accesses and updates. Note that another successful design (GPFS) does not use cooperative caching.

We have changed GFS and DLM to keep track of the global cache contents. We have leveraged the already existing support for range locks in DLM. We have used these changes to improve the performance of GFS with DLM; on micro benchmarks, it can be up to 8 times on our prototype implementation. We assume that the nodes are on a trusted network and the only failures that can happen are fail stop failures.

Our main contributions are the design and implementation of a cooperative caching scheme for GFS and integrating it with lock manager and devising a novel scalable method to do interval searches. We also made some scheduler optimizations which can result in better performance, if a NBD server is used (see Sec. 18.4).

The rest of the paper is organized as follows. In chapter 15, we briefly describe clustered shared disk systems, GFS and DLM locking mechanisms. In chapter 16, we briefly describe the cooperative caching scheme and describe the changes to the locking protocol and DLM. In chapter 17, we present some of our experimental results. We conclude with discussion, related work and conclusions.

Chapter 15

GFS cluster and DLM locking mechanism

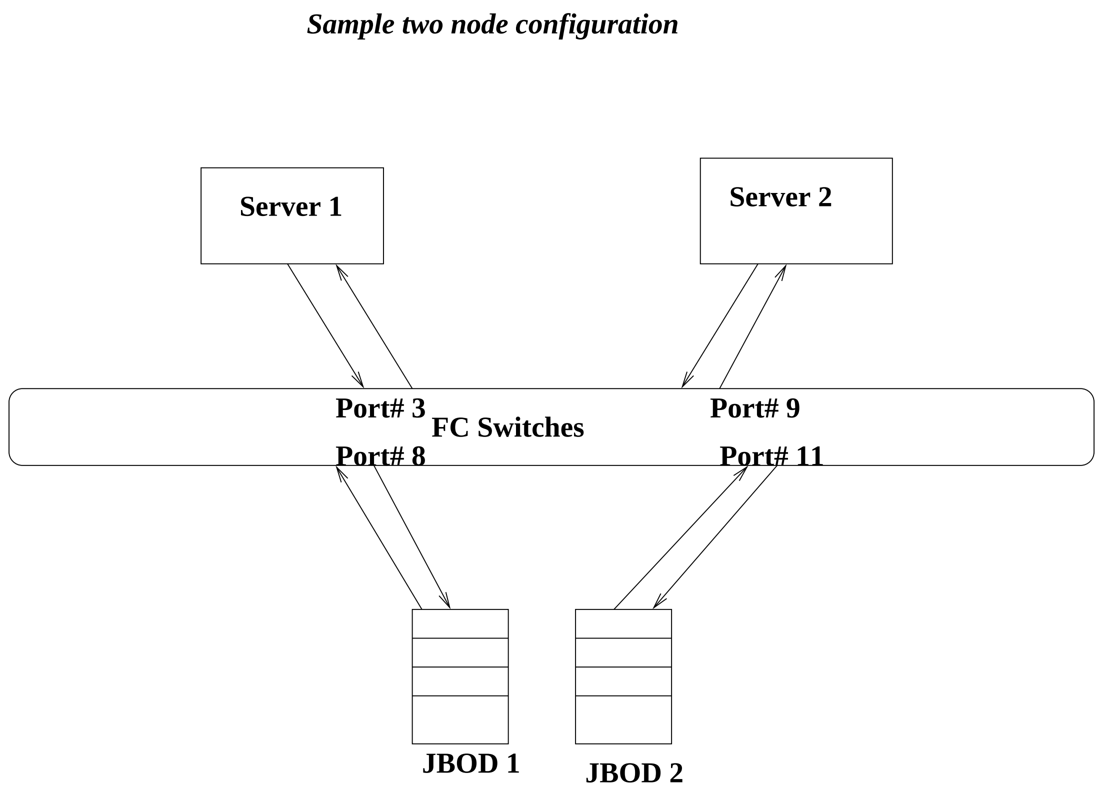

Figure 15.1: Sample two node cluster

A sample two node cluster is depicted in **Figure** 15.1. A group membership algorithm runs on nodes in the cluster which keeps track of the current membership of the cluster at each node. CMAN (Cluster Manager) module of Redhat GFS does this job. After the membership is decided, a primary network partition is selected that is allowed to access the shared disk. Primary network partition can be selected in many ways; for example, by selecting the partition that has more than the half the total number of nodes. Once a primary network partition is selected, the nodes which are not in the primary network partition are blocked from accessing the shared disk. In case of FC switches, this can be achieved by disabling the ports through which the nodes not in primary network partition are accessing the disks. This process is known as fencing. After fencing is done, GFS goes through recovery mechanism to bring the system to a consistent state. When GFS is first installed on the disk (using mkfs.gfs), the name of cluster needs to be provided. When the file system is mounted, it reads the name of the cluster and either registers or joins the service with this name being managed by the CMAN (cluster manager).

Multiple nodes accessing the same file system, coordinate accesses to the disk using a lock manager. The lock manager supports both shared and exclusive locks. A node acquires a lock and then reads the data from the disk. If a exclusive lock is requested by a remote node, dirty metadata and data are first flushed to disk and then the lock is released.

GFS supports journaling of both metadata and data. Each node in the cluster has some space allocated to it on the disk, to which it journals the metadata (and also data if it is requested by setting the appropriate option). During the recovery process, the journals of nodes not in primary network partition are replayed to bring the system to a consistent state. The file system space on the disk is divided into resource groups. Blocks for a file are allocated from resource groups. The allocation algorithm tries to allocate all blocks of a file from the same resource group. The inode in GFS occupies a complete GFS block (default 4096 bytes). When the size of data in file is small, the data is stored in the inode. All indirect blocks are located at same height in a GFS file. GFS uses hashed directory structure to make searching for directories faster.

GFS uses locks of various types; for example: inode locks, resource groups locks, metadata locks. GFS uses the following virtual functions which are called at various points during the lock acquiring/releasing phase; each type of lock has its own implementation for these virtual functions.

- **go_xmote_th:** This function is called before the request is made to the DLM.
- **go_xmote_bh:** This function is called after DLM has processed the request and replied back.
- **go_lock:** This function is called if lock is acquired for the first time on this node.
- **go_unlock:** This function is called when a process releases the lock. Note that, the node does not notify the DLM about the released lock at this point.
- **go_drop_th:** This function is called before a lock is released by a node.
- **go_drop_bh:** This function is called after DLM has processed the lock release request.

15.1 DLM locking overview

DLM (Distributed lock Manager) implements the GFS locking mechanism. Corresponding to each lock *gfs_glock* used by GFS, there exists a *dlm_lkb* structure that is managed by DLM. A *gfs_glock* structure is identified by *ln_number* and *ln_type*; for example, for a lock on a inode, the *ln_number* is set to the inode number and *ln_type* is set to *LM_TYPE_INODE*. Locks with different *ln_type* values are considered non-conflicting. A lock managed by DLM is identified by a lock id. Corresponding to each resource in GFS on which a lock can be acquired, there exists a master *dlm_rsb* structure, that is managed by DLM. The *dlm_rsb* structure has

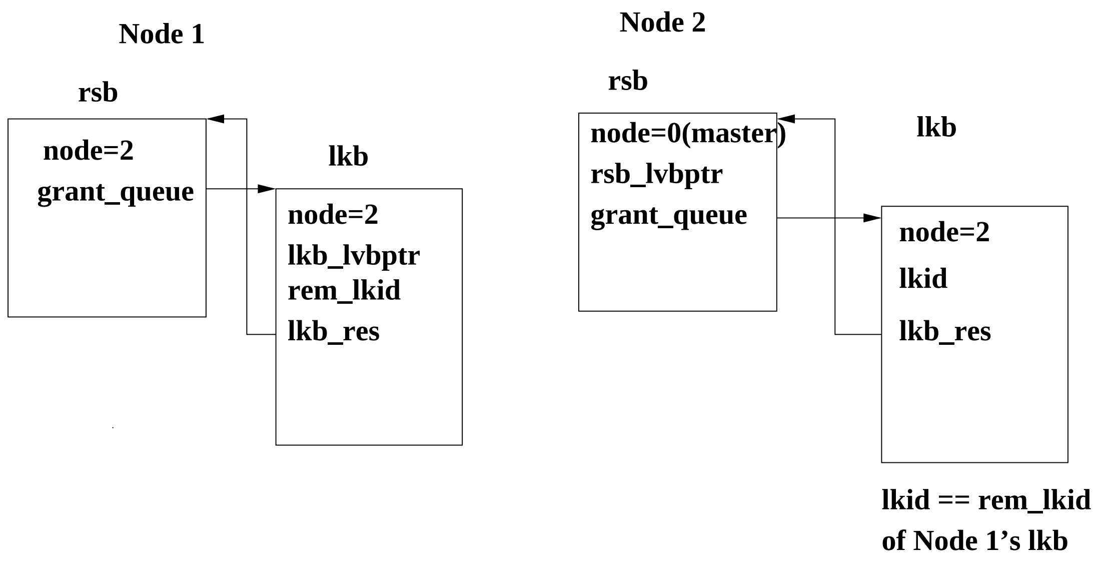

Figure 15.2: DLM locking mechanism overview

three separate queues for granted, convert and pending requests. Each node in the cluster manages a part of a global resource table which defines the mapping between the resource and the node that contains the master resource block. The node containing the resource table for a given resource is determined by the hashing the resource name that is part of *dlm_lkb*.

In **Figure** 15.2, a lock is requested by node 1. The DLM layer running on node 1 creates *dlm_lkb* and *dlm_rsb* structures corresponding to the lock and consults the global resource table(which could be on another node) to determine the node containing the master *rsb*; from here onwards, we shall use *rsb* (*lkb*) and *dlm_rsb* (*dlm_lkb*) interchangeably. In **Figure** 15.2, that happens to be node 2; this means that the *rsb* on node 1 is the slave *rsb*. Then node 1 sends a request for the lock to node 2. Node 2 creates the master *rsb* and create a *lkb* corresponding to this request and links it on the appropriate queue. The node id field of the *rsb* at node 1 contains the id of the node which contains the master *rsb*; in our case it is 2. The master *rsb* has the node id field set to 0. The *lkb* structure linked to *rsb* on node 2, has node id field set equal to the id of the node requesting the lock. On node 1, the same field is set to 0.

When a node fails, during the recovery process, the global resource table is rehashed and the master *dlm_rsb*s are reconstructed from the locks that are being held by the nodes in the primary network partition.

Chapter 16

Design and implementation cooperative caching scheme

16.1 Overview of the cooperative caching scheme

The cooperative caching protocol is briefly described below.

- When DLM get a lock request on a range of a file, whose contents are not present in the global memory, a flag is set indicating that contents are not present in the global memory and reply is sent to the GFS. GFS reads the contents and updates the global memory being managed by DLM with these contents. During this period, no further lock requests with ranges overlapping the range of the granted lock, are granted. Once the global memory is updated, all non-conflicting requests are granted.
- Before an exclusive lock is dropped by a node, the dirty buffers/pages are written to the disk and in the process of releasing the lock, the global memory is also updated with these contents.
- If the data corresponding to the range of a lock requested is in global memory, the data is transfered to the requesting node along with the lock reply.
- Global pages are released when the resource block reclamation daemon runs. Local buffer/page caches are released when *gfs_glock* reclamation daemon runs.
- When failures occur, the lost global pages are not immediately read from the disk. Rather a lazy strategy is followed. As nodes request/release locks the lost pages get updated.

16.2 Changes to DLM

DLM already has support for range locks which is not being used by GFS. We extended this functionality in DLM to support global memory, which we leveraged to implement cooperative caching in GFS.

Each master *rsb* holds the global memory structures for the resource it represents. The global memory is updated when a lock is requested for the first time for a portion or all of the resource and when a exclusive lock is released. A node can has option to turn on/off a flag that tells the master rsb whether to use global memory for this lock or not.

Figure 16.1 and **Figure** 16.2 shows the data structures used by the *dlm_rsb* and *dlm_lkb* to keep track of global and local caches respectively.

The *dlm_rsb* structure holds a *page_range* structure for each lock requested on this resource. Though this is some what space inefficient, this makes it easy to search for a range in *dlm_rsb* and copy. Moreover, the most space consuming part is the global page cache. So as long as we optimize for cache pages, we are on the safer side. The *page_range* structure has *range_start* and *range_end* as its members. The member pages in this range are kept in a doubly linked list. There are two hash tables to make searching easier. One keeps *page_range* structures in its hash table. The hash value is computed based on the range start and range end. The other makes searching for individual pages being managed by this *dlm_rsb* faster. This is used when a new lock is requested. The structure managing the global pages also has space to hold upto 10 *dlm_lkb* pointers that have the index of this page included in their range. This makes operations like *queue_conflict* faster, which requires to find all *lkb*s whose ranges overlap with a given range.

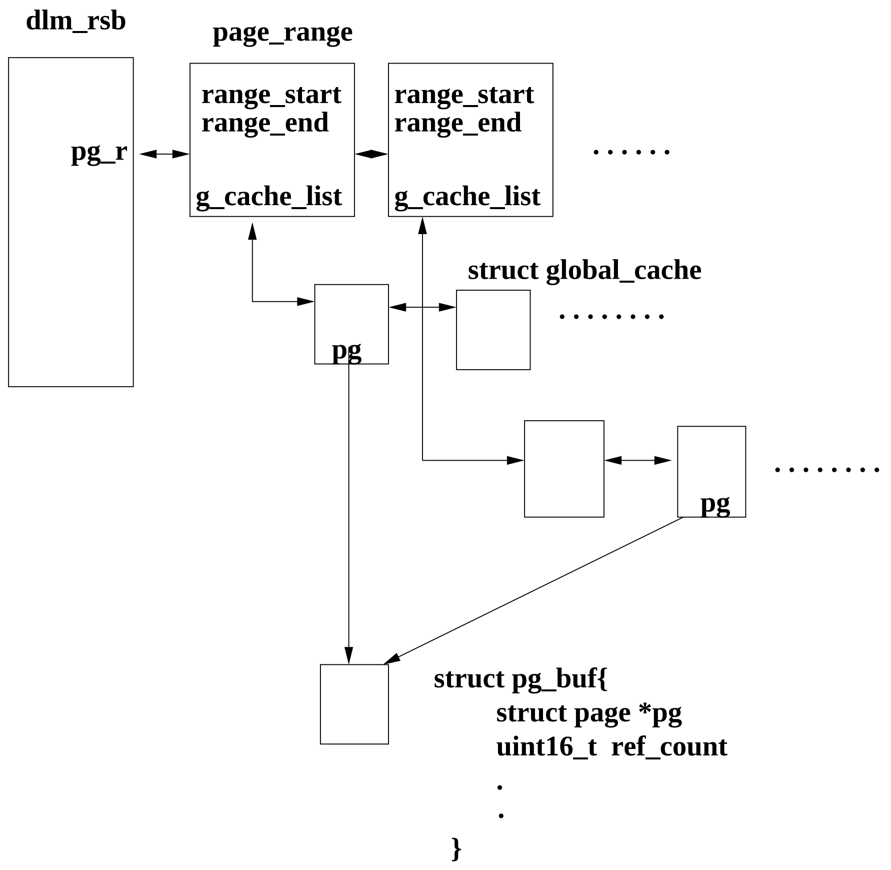

Figure 16.1: Changes to *dlm_rsb* data structures

The *gfs_glock* and *dlm_lkb* structures keep track of the buffers and pages used by this lock in *ccache* structure. Note that DLM layer is supposed to be independent of the module/application which uses it. So, DLM has to be written so that this generality is preserved. DLM exposes a interface, which upper layers have to define and pass it to DLM. Some of the functions, the upper layers that intend to use cooperative caching should define are

- *lock_fn*: to lock the memory to which data would be written or read from.
- *unlock_fn*: to unlock the memory to which data would be written or read from.
- *update_fn* (optional): to mark the memory as uptodate.
- *param*: parameter passed to the above functions.

Note that, the *ccache* structure is created only on the node requesting the lock; this means there is no duplication.

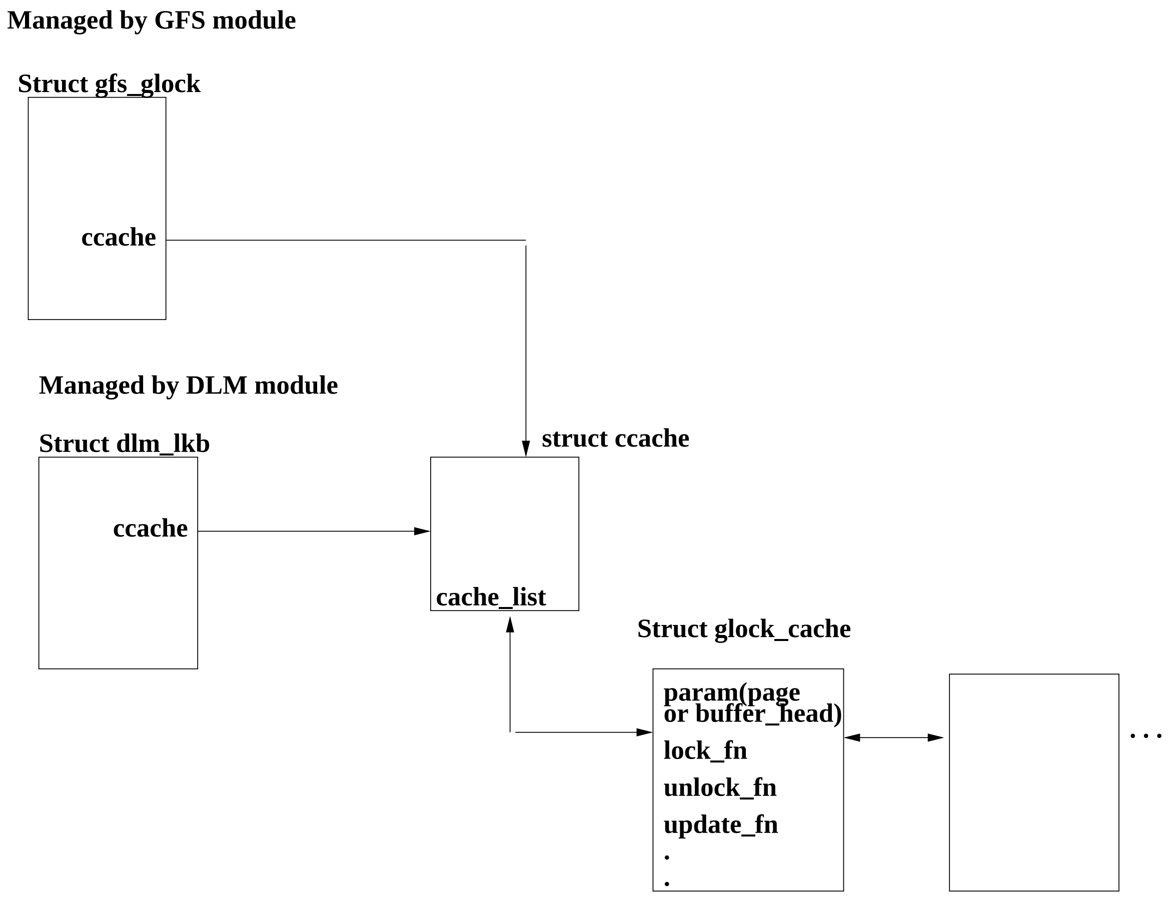

Figure 16.2: Changes to *gfs_glock* and *dlm_lkb* data structures

When a lock is requested with cooperative caching enabled, if the cache contents corresponding to the range on which lock is requested are not in the global cache, the DLM layer sets a flag and returns the reply to the requesting node. When the node that requested the lock detects this flag, it calls a function. It is the job of this function to tell the DLM layer on the node that has master *dlm_rsb*, the contents with which to fill the global cache. Until the contents of the global cache are marked uptodate, no further locks conflicting with this lock are granted by the master *rsb*. When a lock requested in exclusive mode is released, the contents are sync-ed to the global memory. When a node requests a lock in shared mode and if the contents are in the global memory, the contents are transfered along with the reply to the requesting node.

DLM has its own communication layer above the kernel sock layer. The existing interface restricts the size of messages that can be sent or received, which is the size of a kernel page. We extended this layer to support messages of arbitrary sizes, in the process of implementing cooperative caching. Moreover, we did not create any extra copies of page/buffer cache before transmitting over the network. We used the page/buffer cache even here.

Currently, global memory reclamation happens at the *dlm_rsb* level. We feel, it can be more efficient if it is done at page level.

After a failure happens and the system recovers, the global memory contents that are lost, are not read from the upper layer; for example, in case of GFS, DLM doesn't need GFS to supply the lost cache contents immediately. Rather a flag is set to depict that recovery on this *dlm_rsb* has happened, and as upper layers release or request locks, the global memory gets updated.

16.3 Cooperative caching in GFS

We leveraged the changes made to DLM to support cooperative caching for GFS. Currently, cooperative caching is done for inodes and files. So, if the length of file/directory is small, and if the inode/data blocks happens to be in global memory, no extra reads need to be done from the disk. However, if the size of file/directory is large, the indirect blocks are to be read from the disk.

The functions in *ccache* structure which are used by DLM are *lock_buffer*,*unlock_buffer*,*set_buffer_uptodate* for *buffer_head*; and *lock_page*, *unlock_page*, *set_page_uptodate* for a *page*.

Though when a file is opened in shared mode, no range locks need to be acquired by the kernel to read it, the GFS code requests another range lock with different *ln_type==LM_TYPE_FILE* and with same *ln_number*, in addition to the shared lock on the inode. It is when, this lock is being acquired the buffer/page cache is updated. The virtual functions used by GFS code during lock acquiring/releasing phase for a lock of this *ln_type* are defined as follows:

- **go_xmote_th**: The memory required for buffer/page cache are allocated at this point and linked into the appropriate structures managed by the VFS layer.

- **go_lock**: If the buffer/page cache is uptodate, nothing is done. Otherwise the data is read from the disk. Note that, if the data is present at the global memory, DLM marks buffers/pages as uptodate through the function pointers GFS has passed to it. The code which reads data from the disk gets executed, when part of the request data is not present at the global cache. After this function(go_lock) is called, GFS makes sure that the global memory is updated with this data.

- **go_drop_th**: If the lock was acquired in exclusive mode, flush all dirty buffers/pages to the disk.

- **go_drop_bh**: At this point, the global memory has been updated, if required. So the page/buffer cache is released. This and the previous functions can also be called when the glock reclamation daemon runs, as result of which the page/buffer also get released.

. If the page/buffer cache is entirely fetched from the global memory, no extra reads are done from the disk (except possibly, indirect blocks); which means read ahead is disabled. If some part of it is read from the disk during the lock acquiring and global memory updating phase, read ahead is enabled. When a lock on file which was acquired in exclusive mode is released, the data/metadata is written to the disk as well as updated at the global memory.

Some applications initially allocate a large file and later write portions of it. In such scenarios, the information inside the metadata blocks except the access times doesn't change. In such scenarios, if we are willing to compromise on access times, a shared lock can be acquired on the inode and a exclusive lock with *ln_type==LM_TYPE_FILE* can acquired on the ranges of a file. This way multiple applications can write to a file and read from it concurrently, with active portions seldom being read from the disk. This will result in superior performance especially when the data blocks are not contiguous on the disk.

Chapter 17

Evaluation

We modified Redhat Cluster 1.00.00, to incorporate our changes.The experimental setup consisted of a dual processor Opteron with 2 GB of RAM running 64-bit Linux 2.6.16.29 smp kernel and a dual processor Pentium 1266 MHz with 1.5 GB of RAM running 32-bit smp kernel of same version. Both the machines are connected to JBODs hosting Seagate ST336752FC disks through brocade 3800 FC switch. The Pentium machine uses Qlogic 2300 FC card and the Opteron machine uses Fusion MPT FC card. Both the machines are connected by a Gigabit Ethernet. The size of files considered in these experiments is about 25 MB. So it does use indirect blocks.

17.1 Sequential read

17.1.1 Reads from disk

The sequential read performance for the non-cooperative version and the cooperative version is given in **Figure** 17.1. In both the cases, the data is read from the disk. The performance of the cooperative version is almost same as that of non-cooperative version when the size of reads is 4096, 16384 and 65536. The global memory in these cases was on the same node as that requested the lock. In other cases, there is some performance loss. In these cases, the global memory was at remote node. Since the read is being done for the first time the node requesting the lock has to read the data and sync it to the global memory before the lock acquiring phase is completed; which means that currently updating to global memory is synchronous. We think, we can get almost same performance as the non-cooperative case by making the syncing process asynchronous; in other words, have a kernel daemon whose job is to sync the data to the global memory.

17.1.2 Reads from disk versus reads from remote memory

The sequential read performance for the non-cooperative version and the cooperative version is given in **Figure** 17.2. The non-cooperative version reads the data from the disk and the cooperative version reads the data from the remote memory. We have disabled prefetching for the reads from the disk since we have not yet implemented the prefetching part while reading from remote memory. See discussion for some more details. Clearly, in this case reading from remote memory is faster than reading from the disk. We tried to improve

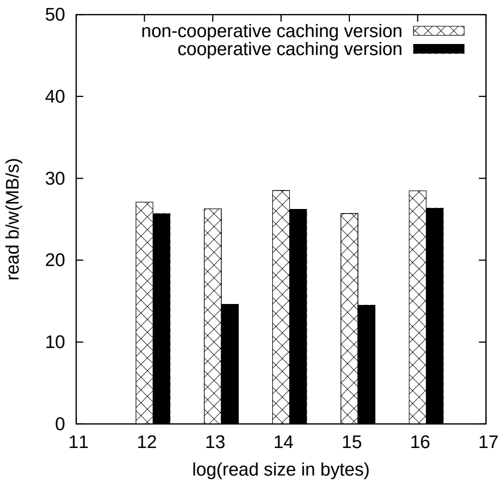

Figure 17.1: Sequential reads from disk performance

the performance of cooperative caching even more by using different TCP algorithms and tuning TCP parameters, but that didn't result in any performance gain.

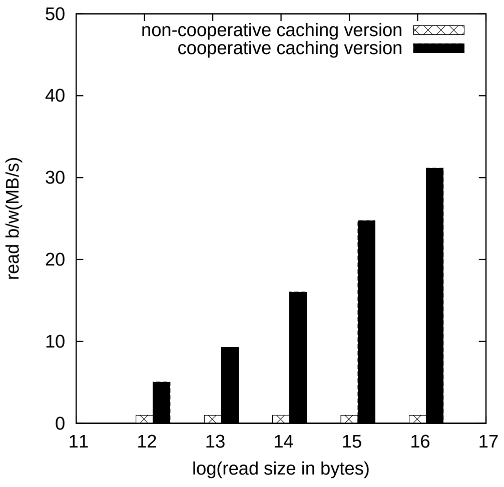

Figure 17.2: Sequential reads from disk versus cache performance

The average latency for reading a $4K$ block from remote memory is 773 microsecs and the latency for reading from disk is 4014 microsecs. The latency involved in reading a $64K$ block from remote memory is 2003 microsecs, whereas it is 64379 microsecs when being read from the disk.

17.2 Random read

17.2.1 Reads from disk

The random read performance for the non-cooperative version and the cooperative version is given in **Figure** 17.3. In both the cases, the data is read from the disk. Note that, in this case, in spite of syncing to global memory synchronously, there is no significant performance loss. This is because the latency incurred while reading from disk is lot larger than latency involved while syncing to the remote memory.

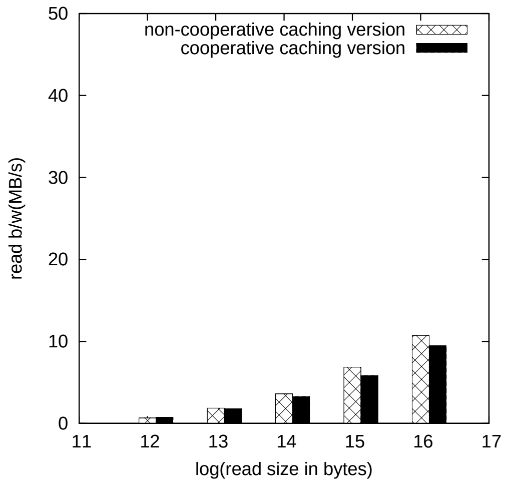

Figure 17.3: Random reads from disk performance

17.2.2 Reads from disk versus reads from remote memory

The random read performance for the non-cooperative version and the cooperative version is given in **Figure** 17.4. The non-cooperative version reads the data from the disk and the cooperative version reads the data from the remote memory. Cooperative caching version is upto 8.8 times faster than the non-cooperative version. File system prefetching was enabled in this case.

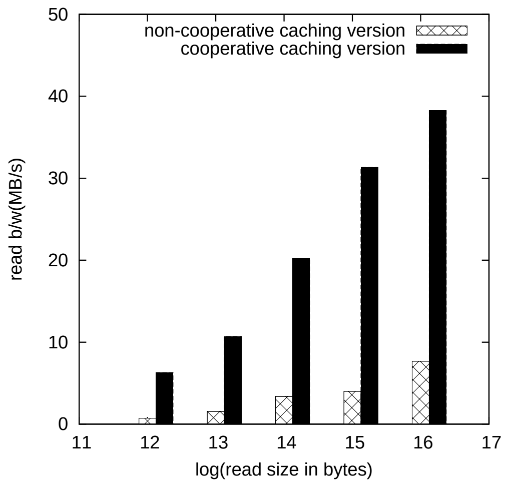

Figure 17.4: Random reads from disk versus cache performance

17.3 Random read/write

17.3.1 Reads from disk

The random transaction performance for the non-cooperative version and the cooperative version is given in **Figure** 17.5. A transaction consists of a read followed by a write or just a read. In both the cases, the data is read from the disk. Note that in this case, there is no performance loss because of syncing to remote memory synchronously. This is because, the latency involved in reading from the disk is much larger than the latency involved in syncing to the global memory. Moreover, dirty data is sync-ed to the remote memory only when the lock on the file range is unlocked at the node level.

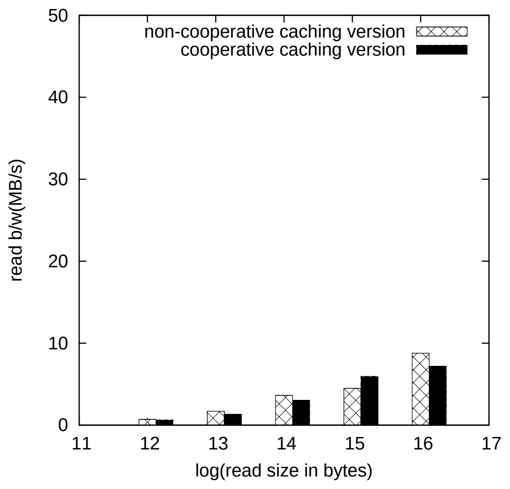

Figure 17.5: Random read/writes from disk versus cache performance

17.3.2 Reads from disk versus reads from remote memory

The random transaction performance for the non-cooperative version and the cooperative version is given in **Figure** 17.6. A transaction consists of a read followed by a write or just a read. In non-cooperative case, the data is read from the disk. In cooperative case, the data is read from the remote memory. The cooperative version is upto 6.7 times faster than the non-cooperative version in this case.

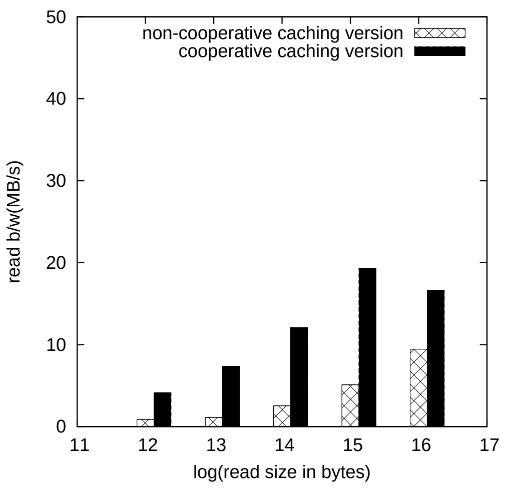

Figure 17.6: Random read/writes from disk versus cache performance

Chapter 18

Discussion

18.1 Searching intervals

As a result of the changes made to DLM to facilitate range locks, the number of locks that are to be managed by DLM has increased. As a result of this, the earlier linked list organization of intervals had become the bottleneck. Note that a range lock, which overlaps with two different range locks on the same file being managed by DLM, can also be requested by some node. One option is to use interval trees to search for the range locks. But interval trees are static; the set of intervals is provided before hand. Even if there is a way to make it dynamic, that would require considerable code changes. Note that we are already using structures to keep track of pages and page ranges in the global memory. We have used two different hash tables, one for the ranges, indexed using a hash computed from the range, and the other for the global pages indexed using a hash computed from the index of the page in the file. The structure which keeps track of a global page, also has space to hold ten *lkb* pointers; the ranges covered by these lkbs include this page index. The idea being that if the number of locks on the file modulo file size is small, then on average there would be less number of locks sharing a global page. In such cases, transversing the whole list of range locks on the file can be avoided. Before making these changes, there were scenarios where reading from the disk was much faster than reading from the remote memory.

18.2 Prefetching from disk

Modern file systems prefetch some amount of data following the current read if they think the reads are sequential. But if the requested data happens to be in the memory of a remote node, prefetching the data following the requested data from the disk increases the latency for this read. So we disable the file system prefetch from the disk for this request, if the whole requested data happens to be in global memory.

18.3 Sequential small reads

Figure 18.1 shows the sequential reads performance with file system prefetching enabled. As evident from the Figure, in this case reading from the disks turns out be faster than reading from remote memory. But, one should note that the latency of first read from the disk is many times larger than the latency of first read from remote memory. The speedup occurs because of prefetching done at the file system level, and to a much smaller extent at the disk cache level. The size of read at which reading from remote memory turns out be faster is between 32KB and 64KB. This suggests the minimum size of data that should be read from the remote memory in a single read. The SCSI disk involved has 8MB of on disk cache and revolves at 10K RPM, is spite of which, file system prefetching plays the major part in enhancing the performance while reading from the disk.

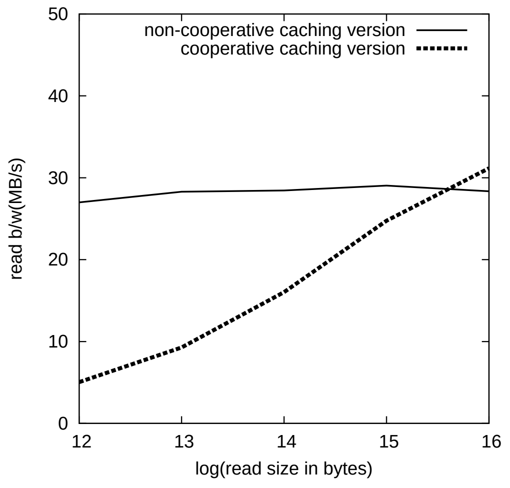

Figure 18.1: Sequential reads from disk versus cache performance on Pentium machine

With filesystem prefetching enabled, the average latency involved while reading from the disk is 143 microsecs for $4K$ reads and 2202 microsecs for $64K$ reads.

18.4 Improvement with changed anticipatory scheduler

The combined disk bandwidth of all nodes can be improved by having a NBD server running on a node and having all other nodes import the block device exported by this node. This results in a single queue system for a single resource (the disk). When all nodes are directly accessing the disk rather than using a NBD server, it becomes a multiple queue system. Moreover, any improvement in scheduler running at the NBD server results in improvement of combined disk bandwidth of all nodes. We have also done some optimizations for the Linux anticipatory scheduler which are described elsewhere. We chose anticipatory scheduler, because it is the kernel's default scheduler. To summarize, we made the anticipatory scheduler aware of underlying device cache. This is useful especially when underlying disk subsystems use large caches. We present some results pertaining to this below. One disadvantage of using a NBD server is that it results in a single point of failure. This can be circumvented by using a hot backup NBD server serving the same disk(s). The hot backup NBD server doesn't do serve any requests until it notices that the primary NBD server has died. Note that, before the backup NBD server can be made online, recovery has to be done to bring the system to a consistent state. The recovery mechanism needs to be studied.

Figure 18.2 shows the configuration used for the experiments. GNBD (Global Network Block Device) server is running on the node to which the Proware SB3160 RAID device is connected. A partition on this RAID device is served by the GNBD server. It has 128

MB of RAM for caching and prefetching purposes. The machines are connected by gigabit Ethernet.

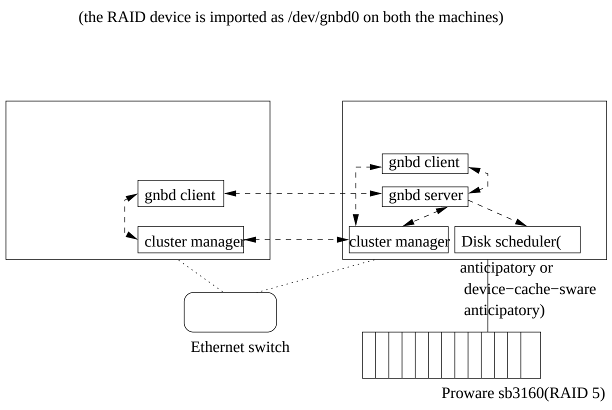
(the RAID device is imported as /dev/gnbd0 on both the machines)
gnbd client
cluster manager
gnbd client
gnbd server
cluster manager
Disk scheduler(anticipatory or device−cache−sware anticipatory)
Ethernet switch
Proware sb3160(RAID 5)

Figure 18.2: Experimental cluster configuration with GNBD server

The performance improvement for random reads over existing Redhat GFS cluster is show in **Figure** 18.3. In the experiment, two nodes do random reads on two different files respectively. The size of files involved is around 45 MB. Note that, the GNBD server does both caching and prefetching. The block device layer of Linux kernel does prefetching. In spite of this various levels of caching and prefetching, we observed upto 22 percent improvement in disk read bandwidth. Moreover, our current implementation updates the data to global memory synchronously, as mentioned earlier. In spite of this, we observed improved performance. We also conducted sequential read experiments, where we did not observe any improvement over the existing implementation when the data is read from the disk. This is because of the caching and prefetching done by the GNBD server and the block device layer which resulted in no cache hits at the device cache level.

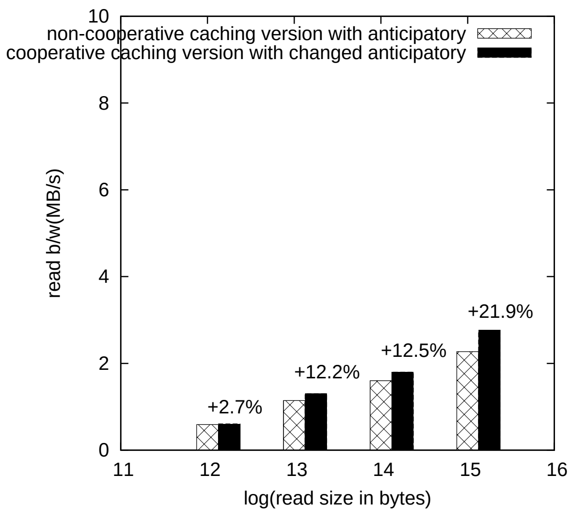

Figure 18.3: Random reads from disk, non-cooperative caching with anticipatory scheduler versus cooperative caching with changed anticipatory

Chapter 19

Related Work

Most of the DLM architecture in GFS is borrowed from [VCP93]. DLM has the notion of Lock Value Block (LVB). The way global memory is tracked is some what similar to the way LVBs are tracked. However, some cases are handled differently:

- when a node first acquires a lock whose complete contents are not present in the rsb.
- when failures occur.
- LVB is not updated as a result of read, write system calls and is not accessed by the VFS layer of Linux kernel.

Moreover, a LVB is over the entire range of the resource and is about 32 bytes.

A lot of work has already been done to achieve cache coherence at microprocessor level. DASH multiprocessor architecture[Lenoski90] uses directory based cache coherence and fetches the data from a remote multiprocessor cluster, if not present in the local cache. Cooperative caching for chip multiprocessors is discussed in [Chang06]. Cache-only memory architecture is discussed in [Hagersten92].

Several cooperative caching schemes are discussed in [Dahlin94]. However, their is no locking aspect involved, as in our case. The cooperative caching scheme which is closest to our scheme is Direct Client Cooperation. In Direct Client Cooperation scheme, a node allows another node to use its memory to store the cache contents that overflow the local memory. So at most there would be only one copy of data/metadata in the system. Moreover, they don't discuss failures of nodes. Peer to peer file systems are discussed in [Dabek01, Muthitacharoen02, Rowstron01]. The ideas used here can also be used when implementing cooperative caching between machines on a wide area network. There are several other works which discuss cooperative caching including [Fan98, Mann94, Voelker98, Annapureddy05]. However, all these works don't have the shared disk aspect..

zFS [Rodeh03] is a scalable distributed file system which using object (files and directories) disks. zFS also uses cooperative caching. However, the locking mechanism and the way the updates are done to disk are different from that of GFS. GFS with DLM is symmetric distributed file system. In case of zFS, lease and transaction manager take care of issues relating to locking and updating to the disk.

Chapter 20

Conclusions and Future work

In this work, we have implemented cooperative caching for Redhat GFS cluster and measured its performance. We found that reading from remote memory is faster than reading from the disk, especially when the reads involve lots of seeks. We have also determined the tuning required to done for the reads from remote memory to be efficient in case of sequential accesses. In case of sequential read from the disk, if the lock on the file is acquired for the first time, synchronously syncing to remote global memory can result in performance loss. This can be avoided by having a kernel daemon that takes care of syncing to remote memory;in other words make syncing to remote global memory asynchronous. We plan to add support for prefetching, asynchronous syncing and cooperative caching of metadata blocks in future. The code using which we conducted the experiments can be found at http://agni.csa.iisc.ernet.in/˜dharma/CCACHE/.

Bibliography

[GFS] “http://www.redhat.com/software/rha/gfs/”

[VCP93] Roy G. Davis, “VAX Cluster Principles,” Digital Press, 1993.

[Dahlin94] M. Dahlin, R. Wang, T. E. Anderson, and D. A. Patterson, “Cooperative caching: Using remote client memory to improve file system performance,” In the 1^{st} OSDI, pages 267-280, Nov 1994.

[Feeley95] M. J. Feeley, W. E. Morgan, E. P. Pighin, A. R. Karlin, H. M. Levy, and C. A. Thekkath. “Implementing global memory management in a workstation cluster,” In the 15^{th} SOSP, pages 201-212, Dec 1995.

[Fan98] L. Fan, P. Cao, J. Almeida, and A. Z. Broder. “Summary cache: a scalable wide-area web cache sharing protocol,” IEEE Transactions on Networking, 8 (3):281-293, 2000.

[Dabek01] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and Ion Stoica, “Wide-area cooperative storage with CFS,” In SOSP, Banff, Canada, October 2001.

[Hagersten92] E. Hagersten, A. Landin, and S. Haridi, “DDM: A cache-only memory architecture,” IEEE Computer, 25 (9):44-54, 1992.

[Mann94] T. Mann, A. D. Birrell, A. Hisgen, C. Jerian, and G. Swart, “A coherent distributed file cache with directory write-behind,” ACM Trans. on Computer Systems, 12 (2):123-164, May 1994.

[Muthitacharoen02] A. Muthitacharoen, R. Morris, T. Gil, and B. Chen, “Ivy: A read/write peer-to-peer file system,” In OSDI, Boston, MA, December 2002.

[Rowstron01] A. Rowstron and P. Druschel, “Storage management and caching in PAST: a large-scale, persistent peer-to-peer storage utility,” In SOSP, Banff, Canada, October 2001.

[Voelker98] G. M. Voelker, E. J. Anderson, T. Kimbrel, M. J. Feeley, J. S. Chase, A. R. Karlin, H. M. Levy, “Implementing cooperative prefetching and caching in a globally-managed memory system,” Proceedings of the 1998 ACM SIGMETRICS joint international conference on Measurement and modeling of the computer systems, Madison, Wisconsin, United States, Pages: 33-43, 1998.

[Annapureddy05] S. Annapureddy, M. J. Freedman, and D. Mazieres, “Shark: Scaling File Servers via Cooperative Caching,” Proceedings of the 2^{nd} USENIX Symposium on Networked Systems Design and Implementation (NSDI 05). Boston, MA, USA. May 2005.

[Chang06] J. Chang, G. S. Sohi, "Cooperative Caching for Chip Multiprocessors," Proceedings of the 33^{rd} annual international symposium on Computer Architecture, Pages: 264-276, 2006.

[Lenoski90] D. Lenoski, J. Laudon, K. Charachorloo, A. Gupta, J. Hennessy, "The directory-based cache coherence protocol for the DASH multiprocessor," Proceedings of the 17^{th} annual international symposium on Computer Architecture, Seattle, Washington, 148-159, 1990.

[Rodeh03] Ohad Rodeh, Avi Teperman, "zFA " A Scalable Distributed File System Using Object Disks," Proceedings of the 20^{th} IEEE/11^{th} NASA Goddard Conference on Mass Storage Systems and Technologies (MSS'03), 207,2003.

[shah] Gautam Shah (IBM research), "Personal Communication".