Extent–like Performance from a UNIX File System

L. W. McVoy & S. R. Kleiman – Sun Microsystems, Inc.

ABSTRACT

In an effort to meet the increasing throughput demands on the SunOS file system made both by applications and higher performance hardware, several optimization paths were examined. The principal constraints were that the on–disk file system format remain the same and that whatever changes were necessary not be user–visible. The solution arrived at was to approximate the behavior of extent based file systems by grouping I/O operations into clusters instead of dealing in individual blocks. A single clustered I/O may take the place of 15–30 block I/Os, resulting in a factor of two increased sequential performance increase. The changes described were restricted to a small portion of the file system code; no user–visible changes were necessary and the on-disk format was not altered.

Introduction

File systems are a common place to find performance problems. The original UNIX file system [Thompson] is elegant in its simplicity: it has a single block size and a simple list based allocation policy. [McKusick] describes the drawbacks of this design and also describes Berkeley's fast file system (FFS). The fast file system solves many performance problems found in the original UNIX file system. The fast file system is the basis for UFS, Sun's UNIX File System.[1]

UFS has served us well for several years. However, both applications and disk subsystems are demanding higher and higher transfer rates through the file system. Applications such as video and sound require much higher data rates than are available today through UFS. Disk subsystems, such as disk arrays [Patterson], are being developed to deliver the desired I/O rates. Measuring the existing UFS showed that about half of a 12MIPS CPU was used to get half of the disk bandwidth of a 1.5MB/second disk.

Goals and constraints

It was clear that the current implementation of UFS did not scale to the desired I/O rates, so we set out to improve the system. We wanted a UFS that used less CPU to run the disks at their full bandwidth. An additional goal was that *all* users of the file system should benefit from the enhancements; the primary constraint was that the on–disk format of the file system could not change, The ‘‘dusty–deck’’ approach insured that no application would need to be aware of the enhancements.

This paper describes an enhancement to UFS that met all our goals. The remainder of the paper is divided into seven sections. The first section reviews the relevant background material. The second section discusses several possible solutions to the performance problems found in UFS. The third section describes the implementation of the solution we chose: file system I/O clustering. The fourth section discusses problems found in the interaction between the file system and the VM systems. The next section presents performance measurements of the modified file system. The sixth section compares this work to other work in this area. The final section discusses possible future enhancements.

Background

To understand our UFS enhancements, it is necessary to understand the basics of the SunOS Virtual Memory (VM) and Virtual File System (VFS) architectures[2]. A brief review is presented here. More details on the VM system may be found in [Gingell] and [Moran]. Readers familiar with the interaction between the VM system and a file system, in particular the `rdwr`, `getpage`, and `putpage` VFS interfaces, may wish to skip forward to the section on UFS performance problems. Readers familiar with either FFS or UFS, in particular the reasons for its rotational delay, may skip past the section on UFS performance problems. Readers are expected to understand the original UNIX I/O system (the buffer cache) explained in [Bach] and [Ritchie].

[1] UFS has been modified to fit into Sun's virtual file system architecture [Kleiman]. Other than that, it has been tracking the fast file system very closely.

[2] The VM and VFS architectures are similar to those in System V release 4. Virtually all references to SunOS are also applicable to SVR4.

Virtual file system interfaces

The SunOS virtual file system (VFS) interface [Kleiman] allows the kernel to support many different types of file systems simultaneously. Each file system type implements two object classes: *vfs* and *vnode*. A VFS object represents a particular instance of a file system. A vnode object represents a particular file within a VFS. These objects export interface routines that the main body of the kernel uses to manipulate a file system without knowing the details of how it is implemented. A file system type may be thought of as a driver that provides a set of file system abstractions without exposing the details of the implementation.

There are many entry points into a VFS, but we need concern ourselves only with the read/write (`rdwr`), read a page (`getpage`), and the write a page (`putpage`) interfaces. These are the interfaces used by the `read`, `write`, and `mmap` system calls that the programmer sees.

The `getpage` interface returns a page filled with data from the vnode at the file offset specified by the caller. The file system may use a page cache supplied by the VM system to store active page data. The entries in this cache are named by the vnode and file offset of the data in the page. The `putpage` interface is used to return a page to secondary storage.

In most SunOS file systems, the `getpage` and `putpage` routines are where the I/O actually occurs. It is important to understand that `getpage` and `putpage` are used asymmetrically. `getpage` is usually called first both for reading and writing. In the read case it is called to retrieve that data from the disk. In the write case it is called to get a copy of the data to be modified. `putpage` is only called when the page is to be written to the backing storage.

When a process uses the `read` or `write` system call, the kernel redirects the call to the `rdwr` entry point of the appropriate VFS. `rdwr` copies the appropriate file data to or from a buffer supplied by the caller. Usually this is the buffer specified by the process in the read or write system call. Many file systems implement `rdwr` by mapping a portion of the file into the kernel's address space and then copying to or from the user's buffer.

SunOS virtual memory system

The SunOS VM model is similar to that of Multics [Organick] and TENEX [Bobrow]. The VM system works in concert with the file systems to manage a cache of vnode pages. To illustrate the caching mechanism, we describe the VM system's management of a simple address space. The address space, associated with a process, is made up of a collection of segments each of which refers to a portion of a file (vnode).

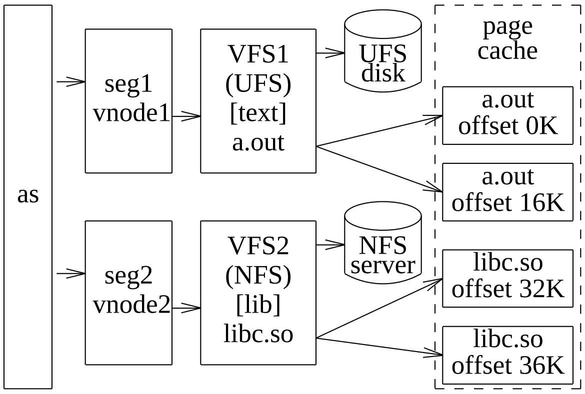

Figure 1: The VM system.

Figure 1 shows a simple address space made up of two files: `a.out`, a file from a local UFS file system, and `libc.so`, a dynamically linked shared library from a remote NFS file system.

Page faults

When a process references an address for the first time, a page fault occurs. The fault is resolved by traversing the object hierarchy and invoking the fault handlers for each object type. Specifically, the kernel finds the address space associated with the process and calls the address fault handler, passing it the faulting address. The address fault handler uses the address to find the enclosing segment and calls that segment's fault handler. The segment's fault handler converts the address into a <vnode, offset> pair and calls `getpage` of the associated file system. The `getpage` routine first requests the VM system to find the page denoted by the <vnode, offset> argument. If the page is found in the page cache, it is returned. Otherwise, the page is not in memory and the file system has to retrieve the page from secondary store. After the data has been retrieved, the file system puts the page in the page cache for future reference.

An important point is that there is no longer a distinction between process pages and I/O pages. Pages are brought into the system for different reasons but they are all labeled in the same way. This unified naming scheme allows all of memory to be used for any purpose, based on demand. All of memory may be an I/O cache if the system is acting primarily as an I/O server, or all of memory may be used up for a single large active process. Older UNIX variants confined I/O pages to a small ''buffer cache.''

UFS details

The UFS implementation uses several internal concepts, such as inode, dinode, logical block, and physical block. These are explained in [Leffler] but we briefly review them here.

UFS represents each active file with an *inode*. An inode is an in–memory version of the control information associated with a file; the inode is initialized when the file is first read from disk from an on–disk structure called the *dinode*. The inode contains information such as file size, the location of the first few data blocks on disk, date created, etc. Each inode is directly associated with a vnode. Inodes also contain meta information that the file system uses to help tune performance. We discuss this information in the `ufs_getpage` section below.

UFS breaks up each file into *logical blocks*. A logical block is the main unit of allocation in UFS[3]. Logical block numbers, or lbns, are numbered from zero and denote a particular block of a particular file. Logical blocks are used for two reasons: to decouple the file system block size from the disk block (or sector) size, and to decouple the location of a block in a file from the location of the block on the disk.

ufs_rdwr

`ufs_rdwr` performs a read by breaking the request into block sized pieces, mapping each file block in turn to an unused portion of the kernel's address space, copying the data to the requesting process, and unmapping the block.

If the page representing the block is not already in memory with an active MMU translation, the copy will fault. The kernel handles the fault by calling `ufs_getpage` to find the page. After the page is retrieved, the MMU translation to the page is set up, the fault returns, and `ufs_rdwr` finishes the copy unaware that the fault ever occurred.

Repeated accesses to the same page will find the page still in memory with an active translation and will avoid multiple page faults.

The work done for a write is similar. The main difference is that when the block is unmapped from the kernel's address space after each block is copied, `ufs_putpage` will be called to start the I/O to the disk. `ufs_rdwr` can also request that `ufs_putpage` wait until the I/O is complete (synchronous write) or that it return after the I/O has been started.

ufs_getpage

When `ufs_getpage` is called, it first checks to see whether the page is actually already in the page cache and returns the page if it is. Otherwise, it converts the vnode and offset into the equivalent inode and logical block number and calls `bmap`, which is responsible for mapping logical blocks of an inode to physical blocks on the disk as well as the allocation of physical blocks on disk. It uses the block pointers in the inode to perform the translation, unless the file is large, in which case the inode contains a pointer to a disk block of pointers; this block is called an indirect block. For large files, `bmap` needs to fetch the indirect block to perform the translation. The physical block number returned by `bmap` is used to start up the I/O.

[3]For the purposes of this discussion, we will assume that the size of a block is always greater than or equal to the size of a page.

The `ufs_getpage` routine is complicated by the heuristics for optimizing read performance. The algorithm is shown in figure 2.

```
bmap() to find disk location
if (requested page not in cache) {
      start I/O for requested
}
if (sequential I/O) {
      do another bmap() if necessary
      start I/O for next page
}
if (first page was not in cache) {
      wait for I/O to finish
}
predict next I/O location
```

Figure 2: UFS getpage algorithm.

In the absence of other information, `ufs_getpage` uses the pattern of logical block requests it sees to predict the file access pattern in the near future. If the pattern of requests is such that the current request is one page greater than the last request, it is assumed that the file is being accessed sequentially. If sequential access is detected, `ufs_getpage` predicts that the next access will be to the page following the requested page. In this event, `ufs_getpage` will *read ahead*, i.e., will start the I/O for the page following the one requested.

page 0	page 1	page 2
sync read page 0 async read page 1 nextr = 1	async read page 2 nextr = 2	async read page 3 nextr = 3

Figure 3: access pattern showing read ahead.

The series of events that will cause read ahead is illustrated in figure 3. Each box represents a page and shows what happens when a fault is taken for that page. The first fault (for page 0) will start an I/O read for page 0 and also start up an I/O read ahead on page 1. The next fault (for page 1) will find page 1 in memory and will start up a read on page 2 and so on.

In figure 3, the first page fault caused both the primary read and the read ahead. Since the fault was for the beginning of the file, it may seem that the read ahead heuristic should not have been enabled. The file system uses an inode field, `nextr`, to predict the location of the next read. When the inode is initialized, `nextr` is set to zero, predicting that the first read will be the first block of the file. Starting read ahead at the beginning of the file turns out to be a beneficial heuristic.

ufs_putpage

When the kernel wishes to free some pages that contain modified data, it calls the appropriate file system's `putpage` routine. `putpage` simply writes out the page data to the correct location on secondary storage.

UFS performance problems

This section considers the reasons that file system operations in UFS are so expensive. The answer comes in two parts: computational overhead and placement policy. There is little that can be done that will reduce the computational overhead. The computational cost can be amortized by moving more data for each traversal of the file system code. This idea was a basic motivation for the FFS changes to the original UNIX file system. Placement policy is more interesting. Even if we reduced the computational overhead to zero, the file system could not deliver the data faster than half the disk transfer rate.

Placement policy

While UFS has many tuning parameters, including ones that affect the placement policy, it is almost always tuned in the same way.

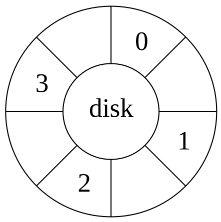

Figure 4: Interleaved blocks.

Blocks from a single file are placed as shown in figure 4 in which you are looking down on one track of a disk platter. (The unlabeled blocks will be used by a different file.) The file system is responsible for placing the logical blocks on the disk in a pattern that is optimal for sequential access. Each block is separated by a gap called the rotational delay or `rotdelay` by the file system code[4]. `rotdelay` is specified in milliseconds and the minimum non–zero value is the rotational delay of one block time. For a file system with a block size of 8KB this is 4 milliseconds on typical disks. The number of blocks placed contiguously between each rotational delay is known as `maxcontig`. `maxcontig` is typically set to 1 as shown in figure 4.

[4]Note that UFS does this differently than file systems in other operating systems in that the gap is maintained by software. Other systems format the disk to have this gap and call it the disk *interleave*.

Rotational delay

Why is the rotational delay necessary? We already know that the file system does read ahead to avoid delays in sequential access. The rotational delays allow the file system enough time to deliver the current block to the requesting process, for the process to compute using the new data then generate a request for the next block, and for the file system to check that the requested block is in memory (due to read ahead) and generate the disk I/O for the next read ahead block. If the file system is properly tuned, the I/O request will get to the disk as the appropriate block is moving under the head. If there were no rotational delay, the next block would already have started under the disk head by the time the disk saw the request. The disk would have to wait almost a full rotation (about 16 milliseconds on today's disks) before starting that request.

This explains why the rotational delay is necessary but we can see that it comes at a cost: having those holes reduces the maximum transfer rate to half that of the disk rate. To solve this performance problem, the rotational delays must be eliminated and the computational overhead of the system must be reduced.

Possible Improvements

In this section we explore the full range of improvements, from hacks to completely new file system implementations. We reject them all except clustering; the discussion of the extent based file system solution is of special interest.

Raw disk

Get rid of the file system altogether by using the raw disk. Some users, mostly those running database applications, actually do this. There is no question of file system overhead; the raw disk is a direct interface plus a few permission checks.

This solution is an act of desperation. There is no file system, no file abstraction, no read ahead, no caching, in short, none of the features that are expected of a file system. The fact that users resort to the raw disk is usually an indication that the file system is too slow.

File system tuning

Tune the file system to take advantage of track buffers. A track buffer is a memory cache the size of one track commonly found on newer disks, such as SCSI disks, that have on board controllers. When a read request for a block is sent to the disk, the entire track is read into the buffer. If successive blocks are on the same track, they are serviced immediately from the track buffer. Therefore, there is no need for rotational delay between successive file blocks. UFS can be tuned to attempt to place successive blocks contiguously on the disk by setting `rotdelay` to zero (see figure 5). This increases read performance substantially, since an entire track's worth of file data can be read in one rotation.

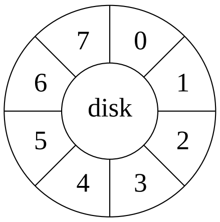

Figure 5: Non–interleaved blocks.

At first glance this looks like a win. If we had no rotational delays then a track would contain twice as much relevant data and the effective disk bandwidth would be twice as great. However, not all drives have track buffers. Drives without track buffers would suffer substantial performance penalties on both reads and writes. Still, many of the drives sold today do have track buffers, so why not take the easy way out? The answer is write performance; it suffers horribly when the file system has no rotational delay. The reason for this is that the track buffer acts as write through cache, each write goes through the track buffer to the disk[5]. Since the writes go directly to the disk, we need the rotational delay between each block or each write will wait a full rotation before beginning. Given that writes will degrade and only some reads will improve, we rejected this approach.

Driver clustering

First tune UFS to allocate sequential logical file blocks contiguously by setting `rotdelay` to zero. Then have the disk driver combine (cluster) any contiguous requests in its queue into one large request. This is relatively simple to implement, since many SunOS disk drivers call a routine, `disksort`, that orders the disk queue for optimal seek performance. These drivers call `disksort` each time a new block request is received. `disksort` could coalesce multiple adjacent blocks into one I/O request.

One disadvantage of this approach is that the file system code must be traversed for each block. We felt this to be excessively expensive in CPU cycles. Another problem is that driver clustering helps only writes. The reason for this is that there can be many related writes in the disk queue at once, since writes are asynchronous in nature. Reads, on the other hand, are synchronous, so there can be at most two, the primary block and the read ahead block, in the queue at once. Finally, not all drivers call `disksort`. Instead, those drivers depend on intelligent controllers to do the ordering of requests.

[5]If the block went into the buffer, but not on the disk, the system and/or user may believe that the data is safely on stable storage. If the system crashes the data is lost, even though a promise was made that the data was safe.

Extent based file system

Replace UFS with a new file system type, an extent based file system. This is a popular answer to file system performance issues. The basic idea is to allocate file data in large, physically contiguous chunks, called extents. Most I/O is done in units of an extent. This improves performance in both I/O rate and CPU utilization, since the I/O is done contiguously, and file system CPU overhead is amortized over larger I/Os. Typically, the user can control the size of these extents on a per-file basis. In most cases the on–disk file system represents the mapping of logical file blocks to physical blocks as a tuple of <logical block number, physical block number, length>. In addition, the on–disk inode is usually expanded to maintain the user's requested extent size(s).

The disadvantage of exposing extents to the user is that it is unlikely that a user will be able to choose the ''right'' extent size. Even if a good extent size can be determined for a particular file, the size will vary between machines with different configurations, between file systems on the same machine, or even between different locations on the same file system. For example, consider a variable geometry drive (a drive that has more blocks on the outer tracks than on the inner tracks). Such a drive may have different values for the optimal extent size at different locations. The same sort of problem exists when considering a single drive versus a disk array [Patterson]. Trying to write portable code that knows about extents is close to impossible.

Exposing this sort of information to the application is rarely helpful and is frequently confusing. Users rarely want to manage extents. Usually, they really want some sort of performance promise. If the file system performed satisfactorily, the user would never consider telling the file system what to do. We believe that the file system is capable of the required performance with no assistance from the user.

Another disadvantage of this approach is that a change in on–disk file system format would require changes to many system utilities, such as dump, restore, and fsck.

File system clustering

Modify UFS to combine blocks adjacent to the requested blocks into a larger I/O request. This produces most, if not all, of the advantages of an extent–based file system without requiring changes to the on–disk format of UFS.

Clustering Implementation in UFS

This section presents the implementation of the solution we chose, clustering in the file system. The goal of our solution is to realize the full potential of the disk but to incur less CPU cost per byte doing so.

To reach our goal we made two basic changes: we tuned the file system to allocate files contiguously and we changed the file system to transfer sequential I/O in units of clusters. A *cluster* is simply a number of blocks, usually about 56KB worth[6]. This approach solves both of the problems in the old system: the rotational delays are removed, which potentially allows a single file to be read or written at the disk speed, and clusters are used in place of blocks which causes the file system code (and the driver code below it) to be traversed far less frequently than in the old system. The details of our implementation follow.

Allocator details

There were no changes to the allocator. The UFS allocator has always been able to allocate files contiguously. This is almost true; in reality the allocator *tries* to allocate files as requested, but it may not be able to do so if the disk is fragmented. Since our work depends heavily on contiguous allocation, it is important to have confidence in the allocator's ability to allocate contiguously.

Most extent based file systems have the ability to preallocate extents to insure maximum transfer rates. We had originally considered preallocation as well but experience showed that this was largely unnecessary. We tried several tests, ranging from filling up an entire partition with one file to filling up the last 15% of a heavily fragmented `/home` (users' home directories) partition. In the best case, the average extent[7] size was 1.5MB in a 13MB file. In the worst case, the average extent size was 62KB in a 16MB file. We expected the allocator to do well when there were no other competing files, but were worried about the fragmented file system case. The results showed us that the allocator thinks ahead enough that it has a good chance of being able to allocate blocks in the desired location. The reason that the allocator is able to do so well is that it keeps a percentage of the disk (usually 10%) free at all times. The free space is not in a fixed location; the allocator may use any free block at any time as long as it keeps a certain percentage free. It uses this flexibility to do better allocation, good enough that we decided not to ''fix'' the system by adding preallocation code.

[6] 56KB is used because there are still drivers out there with 16 bit limitations.

[7] Extent is used here to indicate a span of contiguous blocks followed by a gap (unrelated block). An extent may contain any number of clusters.

Sizing clusters

We use `maxcontig` to indicate the desired cluster size[8]. Although we ask the allocator to create clusters of size `maxcontig` blocks, the actual cluster size may be less than that. For example, we may want to transfer a 40KB cluster but the portion of the file that we want may be in two 20KB extents on the disk. Somehow, the file system needs to be told that 20KB is the best that can be done at the moment.

The `bmap` routine is able to give us this information since its job is to know about the location of the file on disk. `bmap` used to take a logical block number and return a physical block number. We modified it to return a length as well as the physical block number. The portion of the file starting at the logical block given to `bmap` is located at the physical block returned and continues for at least the length returned. The length returned is at most `maxcontig` blocks long and is used as the effective cluster size by the caller (`ufs_getpage` or `ufs_putpage`).

Read clustering implementation

The implementation of read clustering is in `ufs_getpage`, no changes were required anywhere else (but see the section on page thrashing below). The `ufs_getpage` code still implements the same ideas: do a transfer, predict the location of the next transfer, and if the prediction comes true start the read ahead. The changes in `ufs_getpage` all stem from the switch to clusters from blocks: the rest of the code did not need to be changed. The read ahead implementation, shown in figure 6, is a little different, since we don't do a read ahead on each page, just on each cluster.

page 0	page 1	page 2	page 3	page 4	page 5	page 6
sync 0,1,2 async 3,4,5 nextrio 3			async 6,7,8 nextrio 6			async 9,10,11 nextrio 9
1st cluster			2nd cluster			...

Figure 6: Clustered reads when `maxcontig = 3`.

As before, each box represents a page and contains the actions that occur as a result of the call to `ufs_getpage` for that page. The first box shows the synchronous read of the first cluster, and the asynchronous read of the second cluster. It remembers where to start the next read ahead by setting the `nextrio` inode field to the current location plus the size of the current cluster. The next two calls do nothing except return the page. Even the call for page 3 finds the data in memory because this data was prefetched.

[8] Previously, when `rotdelay` was zero, `maxcontig` had no meaning, but now it always indicates cluster size.

But we notice that this is the start of a new cluster and we start up the prefetch of 6, 7, and 8. The pattern repeats indefinitely, every third fault will start a prefetch three pages ahead.

Earlier, we said that, although the allocator tries to place a file contiguously on disk, it may not be able to do so because of fragmentation. This means that the cluster sizes sent back from `bmap` may vary at any point. In fact, an old file system will always send back a cluster of one block because of the rotational delays between each block. To insure that the read ahead code works regardless of cluster size, the code that sets up the next read bases its calculations on the returned rather than desired cluster size.

Write clustering implementation

The implementation of write clustering is contained in `ufs_putpage`. We handle writes by assuming sequential I/O and pretending that the I/O completed immediately (in other words, do nothing). If the sequentiality assumption is found to be wrong at the next call, we write the previous page out and then start over with the current page. If the assumption is correct, we keep stalling until a cluster is built up and then write out the whole cluster. The implementation relies on the page cache to hold dirty pages that `ufs_putpage` pretended to flush. The sequence of events is shown in figure 7.

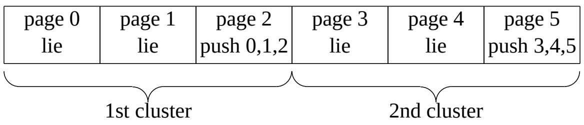

Figure 7: Clustered writes with `maxcontig = 3`.

To implement write clustering, we added two more inode fields: `delayoff` and `delaylen`, as seen in figure 8. These new fields indicate the offset of the first page that was delayed and the number of pages delayed (in bytes), respectively.

```
if (delaylen < maxcontig &&
    delayoff + delaylen == off) {
        delaylen += PAGESIZE
        return
}
find all pages from delayoff
    to delayoff + delaylen
while (more pages) {
        bmap()
        start I/O for this cluster
        subtract that many pages
}
```

Figure 8: Clustered write algorithm.

We use these variables to detect sequential vs. random write patterns. If we do detect random writes, we write out the old pages between `delayoff` and `delayoff + delaylen` before restarting the algorithm with the current page; this is not shown in figure 8.

The fact that the allocator may not be able to allocate contiguously is reflected in the addition of the while loop. Note that this means we do not know if the file is allocated contiguously until we try to write out the cluster.

Unanticipated Problems

The implementation of clustering uncovered other problems in the system which are described here. Many of these can be traced to the interaction of the file and VM subsystems.

Page thrashing.

We thought that the file system was the only major bottleneck in I/O throughput, but in fixing it another problem area appeared: the paging part of the VM system. After reducing the file system overhead by clustering, we expected to be able to see throughput rates equivalent to the disk bandwidth. The throughput was lower than expected and we found that the VM system was the culprit. Pages were entering the system at a higher rate than they could be freed.

The unified VM system has only two ways of freeing pages: removing the backing store (unlinking the file) or running the pageout daemon. The pageout daemon implements (or tries to implement) a least recently used page replacement algorithm. The algorithm is the basic two handed clock and is explained in [Leffler]. The first hand of the clock clears reference bits and the second hand frees the page if the reference bit is still clear. The hands move, in unison, only when the amount of free memory drops below a low water mark.

Considering large sequential I/O, we can see that the pages just brought in are recently touched and as such will not be candidates for page replacement. This has the side effect of using all of memory as a buffer cache for I/O pages. For limited I/O, this is generally a good policy, but for large (greater than memory size) I/O this is a poor policy since it will replace all, potentially useful, pages with I/O pages that are unlikely to be reused. The VM system implements a least recently used (LRU) page replacement algorithm but for large I/O it should implement most recently used (MRU).

Suppose we were to move an infinite amount of data through the system. If we have other users on the system, we don't want to disturb their pages or they won't be able to do any work. In this case, the best thing to do is to use and reuse a small number of pages, say the current cluster's worth. Unfortunately, this is not always the best thing to do or it would be the default in the system. If we used MRU for every file, we would effectively turn off caching, which is as bad as the original problem of destroying the cache.

We needed a compromise that would allow large I/O to go through the system with little impact but still leave in place the caching effects for smaller files. The compromise is inelegant and eventually the paging subsystem will be improved to address these issues properly. For now, we turn on *free behind* if the file is in sequential read mode, at a large enough offset, and free memory is close to the low water mark that turns on the pager.

Free behind is triggered in `rdwr` when the kernel unmaps the page. If the free behind conditions specified above are met, then the unmap will cause a call to `ufs_putpage` that will free the page. Free behind has the desired attribute that the process that is causing the problem is the process finding the solution. The pageout daemon no longer wakes up to free pages when the system is heavily I/O bound, since the I/O bound processes are doing it themselves. Having a process do the free behind in the I/O code path eliminates the overhead associated with switching to and running the pageout daemon.

Write limits or fairness

There is a fairness problem with `write` in the VM system. A single process can lock down all of memory by writing a large file (remember that write I/O is asynchronous; the kernel copies it and allows the user process to continue). In old UNIX systems, the buffer cache imposed a natural limit on the amount of memory that could be consumed for I/O. In the SunOS VM implementation, where all of memory is used as a cache, there is nothing to prevent a single process from dirtying every page. For example, a large process dumping core can cause the system to be temporarily unusable, since all the pages are essentially locked (they are dirty and in the disk queue which is the same as being locked down).

This is a basic fairness problem – the asynchronous nature of writes may be used to the advantage of one process, but it may be at the expense of other processes in the system.

Our solution to this problem is to limit the amount of data that can be in the write queue on a per file basis. We do this by adding what is essentially a counting semaphore in the inode. Each process decrements the semaphore when writing and increments it when the write is complete. If the semaphore falls below zero, the writing process is put to sleep until one of the other writes completes.

The initial value of the semaphore has to be chosen carefully. If it is too large we return to the old problem; if it is too small, we will degrade both sequential and random performance. The sequential problem is exposed when we consider the I/O path as a pipeline. We need to feed the pipe at a fast enough rate that we never have any bubbles. For example, suppose we allowed only one write at a time in the queue. The first write would go down to the driver and the second would block, waiting for the first to complete. When the first completes, the second starts down, but this is too late. By the time the second request makes it out to the drive, there is a good chance that the drive will have rotated past the desired block.

The pipeline problem can be solved by allowing two or three outstanding writes, but this is still not good enough. There is another problem with random access. Consider a process that seeks to the beginning of the disk, writes a block, seeks to the end, writes a block, back to the beginning, writes a block, and so on until N blocks have been written. If we allow the disk queue to be infinitely large, then `disksort` will get a chance to sort the requests such that the system will seek to the beginning, write N/2 blocks, seek to the end, and write N/2 blocks. The effective I/O rate will be much higher in the case without a write limit than the case with a write limit of one. For this reason, we allow a fairly large (currently 240KB) amount of I/O per file in the disk queue.

The limit is currently set on a global basis for all processes. This is not as flexible as it could be. The write limit may be better implemented as a resource limit on a per process basis (see `getrlimit(2)`).

Performance Measurements

We ran several benchmarks, from pure I/O to multi–user time–sharing, to test out our work. The I/O benchmarks, as shown below, showed substantial improvements, but the time–sharing benchmarks improved only slightly.

We were a little disappointed with the time–sharing numbers until we examined the benchmark in detail. The benchmark, MusBus, was spending most of its time sleeping and the rest of the time running small programs such as `date(1)` and `ls(1)`. The largest I/O transfer done by Musbus was around 8KB which is the file system block size. In other words, MusBus didn't move any substantial amount of data.

	cluster size	rot delay	UFS version	free behind	write limit
A	120KB	0	SunOS 4.1.1	Yes	Yes
B	8KB	4	SunOS 4.1	Yes	Yes
C	8KB	4	SunOS 4.1	No	Yes
D	8KB	4	SunOS 4.1	No	No

Figure 9: IObench run descriptions.

We use an internal program called `IObench` to show transfer rates. Figure 9 explains the configuration of each of four I/O benchmark runs. The hardware configuration is the same in each run, an 8MB, 20MHz Sparcstation 1, with one 400MB 3.5"

IBM SCSI drive. We used a kernel that has variables that enable and disable the old and new code in an attempt to get an apples to apples comparison. The ‘‘A’’ configuration is almost identical to that shipped with SunOS 4.1.1; the difference is that the file system has been tuned to use 120KB clusters instead of 56KB clusters. The last configuration, ‘‘D,’’ is a close approximation of a SunOS 4.1 installation; the file system has been tuned to make 1 block clusters with the standard 4ms rotational delay. The ‘‘B’’ and ‘‘C’’ configurations are similar to ‘‘D’’ but add some of the paging and fairness heuristics described in the section on unanticipated problems.

In the results shown below, the columns are headed by a three letter name indicating the type of I/O. The first letter means **F**ile system, the second letter indicates **S**equential or **R**andom, and the third letter indicates **R**ead, **W**rite, or **U**pdate. The difference between write and update is that in the update case the file's blocks have already been allocated.

	FSR	FSU	FSW	FRR	FRU
A	1610	1364	1359	383	452
B	805	799	790	369	431
C	749	783	784	366	428
D	749	722	718	370	545

Figure 10: IObench transfer rates in KB/second.

Figure 10 shows the transfer rates, for the various I/O types, for four different software configurations. Since the numbers are hardware specific, we show and discuss the ratios below.

	FSR	FSU	FSW	FRR	FRU
A/B	2.00	1.71	1.72	1.04	1.05
A/C	2.15	1.74	1.73	1.05	1.06
A/D	2.15	1.89	1.89	1.04	0.83

Figure 11: IObench transfer rate ratios.

In figure 11, we can see that almost all I/O rates improved, some slightly and some substantially. Predictably, the sequential I/O rates improved about a factor of two. Reads are better than writes because the track buffer helps only reads. We made a tradeoff in favor of reads in not adding rotational delays between clusters. If the delays are present, the writes will improve slightly, but the reads will degrade slightly.

The random update (or write) numbers went down when compared to the generic 4.1 UFS. We made a tradeoff between performance and fairness in favor of fairness, which is explained in the section on unanticipated problems.

CPU	Notes
2.6s	4.1.1 UFS, no rotdelays, 16MB mmap read
3.4s	4.1 UFS, rotdelays, 16MB mmap read

Figure 12: System CPU comparison.

We used yet another internal benchmark for comparing CPU time. The benchmark is similar to IObench, in fact it shows identical I/O rates, but uses the `mmap` interface to avoid the copying of data from the kernel to the user. The IObench CPU times are dominated by the copy time and hence are approximately the same. Since we want to show the overhead of the new system versus the old, we used `mmap`. The cpu times in figure 12 show the seconds used by the CPU to read a 16MB file. The new UFS is approximately 25% more efficient in terms of CPU cycles. We believe that we can do even better; we explain how in the section on further work.

Comparison to Related Work

Peacock's System V clustering [Peacock] is the most similar work we've found. The reasoning of reducing per byte overhead by doing larger requests is the same. Both designs try to improve performance by turning sequential I/O requests into larger sequential I/O requests. We believe that most of the following differences can be traced to starting with one base or the other, UFS versus the System V file system (S5FS).

- We depend on the FFS allocator to lay out the files contiguously. Originally we had planned to preallocate blocks, but we found that the allocator does such a good job that there was little to be gained by preallocation. The same is not true of the S5FS allocator. As Peacock pointed out, it is based on a free list that gets scrambled as the file system ages. Peacock was forced to rewrite the allocator to make use of the new bitmap free list. The rewrite caused on–disk format changes which were reflected in the file system utilities such as fsck, mkfs, etc.
- The UFS interfaces (`ufs_getpage`, `ufs_putpage`) are general enough that no changes were needed for clustering. Unfortunately, the same is not true of the S5FS interfaces (`bread`, `bwrite`). Peacock added `mbread` and `mbwrite` to cluster the I/O while we were able to hide the clustering beneath the `ufs_getpage` and `ufs_putpage` interfaces.
- Our write algorithm is different, it starts a write each time a cluster boundary is crossed. Peacock's waits until the buffer cache fills up. The problem with waiting is that the system periodically flushes the cache to avoid file system inconsistencies in the event of a system crash or power failure. If the machine has a large buffer cache (large memory) then the flush may cause a proportionally large I/O burst. If the I/O were flushed to disk at each cluster boundary, the disks are kept uniformly busy, instead developing large disk queues. Smoothing out the disk queue will improve perceived performance since new requests will be serviced quickly.
- As described above, the SunOS VM system had no I/O heuristics. Peacock was able to use the buffer

cache heuristics where we had to add them in order to prevent the pageout daemon from hogging the machine.

Further Work

Performance work is never finished; there is always one more refinement. In this section, we sketch out further work that could be applied to the file system. Some of these ideas have to do with clustering but others look at other ways of improving other aspects of file system performance.

Random clustering. Clustering is currently enabled only when sequential access is detected in the `ufs_getpage` routine. Certain access patterns, such as random reads of 20KB segments of a file, will not receive the full benefits of clustering. If the request is a `read` of a large amount of data, it is possible that the request size could be passed down to the `ufs_getpage` routine, which could use the request size as a hint to turn on clustering for what is apparently random access.

Bmap cache. The translation from logical location to physical location is done frequently and gets more expensive for large files because of indirect blocks. A small cache in the inode could reduce the cost of `bmap` substantially.

UFS_HOLE. Since UFS allows files to have holes, it is possible for `bmap` to return a hole. If we look back at the `ufs_getpage` algorithm (figure 2), we see that `bmap` is called even when the requested page is in memory. The reason for this call is that `ufs_getpage` needs to know if the requested page has backing store (i.e., is not a page of zeros from a hole in a UFS file). If the page has no backing store, then `ufs_getpage` must change the page protection bits to be read only. A read only page will fault when written, allowing UFS the chance to allocate the block to back the page. If the system did not enforce these rules, a `write` may appear to succeed but later will find that there is no more space in the file system.

If UFS did not allow holes in files, we could bypass the `bmap` in all the cases that the page was in memory. One possible solution is to remember whether the file has holes and do the `bmap` only if the page is not in memory or if the file has holes.

Data in the inode. Many files are small, less than 2KB. Caching small files in the system causes fragmentation since the cache is made up of pages which are typically larger that the average file. We would like the caching effect without the fragmentation effect. This could be achieved by increasing the size of the inode in memory and caching small files in the extra space. This is already done for symbolic links if the link is small enough (the space normally used for block pointers is filled with the symlink data on the first access). Inodes are already cached in the system separately from pages which means that the system could satisfy many requests directly from the inode instead of the page cache. This would not work for `mmap()` since the data would not be page aligned.

Extents vs blocks. UFS maintains a physical block number for each logical block number. Given that UFS now allocates mostly contiguous files, there is a potential for substantial space savings by storing extent tuples of <logical, physical, length> instead of a long list of physical blocks. Unfortunately, this would mean an on–disk format change which is not acceptable for UFS. However, if this idea were coupled with the inode cache, large files could use the extra space as a `bmap` cache. To maximize the benefit of the space, the cache could be a cache of extent tuples.

B_ORDER. We would like to improve performance of UFS for the average user, not just the users who want high sequential I/O rates. One approach is to discard UFS in favor of a log based file system [Rosenblum]; this approach has merit. However, there are improvements that can be made to UFS today, and the installed base of UFS disks makes them worth considering.

A long standing problem with UFS is that it does many operations, such as directory updates, synchronously to maintain file system consistency on the disk. The file system uses synchronous writes to insure an absolute ordering when necessary. If there was a way to insure the order of critical writes, the file system would be able to do many operations asynchronously. The performance of commands like `rm *` would improve substantially.

We are considering adding a new flag, B_ORDER, that would be passed down to the various disk drivers. Requests in the disk queue with the B_ORDER flag may not be reordered by the driver, by `disksort`, or by the controller.

Summary

We have shown an enhancement that doubles the potential I/O rate of any UFS based file system. We described our implementation and the results of our implementation. The results show that the disk potential can be realized and also show that our method is less costly in CPU cycles than the old method.

Our approach was similar to that taken by extent based file systems, but differs in important ways: the extent size is variable, maintained by the file system, and is not exposed to the user. We believe that the user is rarely able to choose a correct extent size because there rarely exists a ‘‘correct’’ extent size. The optimal extent size varies based on many factors that may change during the life of an application. Even given that an extent based file system may be able to provide guaranteed throughput for the application that chose the optimal extent size, we believe that the enhanced UFS will provide better average throughput, since UFS is trying to allocate extents for all applications, not just the ''smart'' applications.

Acknowledgements

Many people contributed to this project. We would like to thank the following: Anil Shivalingiah, who explained the VM implementation over and over, Matt Jacob, for SCSI knowledge and the driver clustering implementation, Glenn Skinner, Bill Shannon, John Pope, Mark Smith, and David Rosenthal, for their helpful comments on this paper, Rich Clewett and Pat Townsend, for providing the hardware resources without which this project would have never completed, and the systems group environment at Sun Microsystems that made this work possible.

References

[Bach]
M. Bach, *The Design of The Unix Operating System, Prentice–Hall,* 1986.

[Bobrow]
D. Bobrow, J. Burchfiel, D. Murphy, and R. Tomlinson, ''TENEX, a Paged Time Sharing System for the PDP–10,'' *Communications of the ACM,* 15(3) March 1972.

[Gingell]
R. Gingell, J. Moran, and W. Shannon, ''Virtual Memory Architecture in SunOS,'' *Proceedings of the Usenix Conference,* Summer 1987.

[Kleiman]
S. Kleiman, ''Vnodes: An Architecture for Multiple File Systems in Sun UNIX,'' *Proceedings of the Usenix Conference,* Summer 1986.

[Leffler]
S. Leffler, M. McKusick, M. Karels, and J. Quarterman, *The Design and Implementation of the 4.3BSD UNIX Operating System, Addison–Wesley,* 1989.

[McKusick]
M. McKusick, W. Joy, S. Leffler, and R. Fabry, ''A Fast File System for UNIX,'' *ACM Transactions on Computer Systems,* 2(3) August 1984.

[Moran]
J. Moran, ''SunOS Virtual Memory Implementation,'' *Proceedings of the European UNIX User's Group,* April 1988.

[Organick]
E. Organick, ''The Multics System – An Examination of Its Structure'' *M.I.T. Press,* 1972.

[Rosenblum]
M. Rosenblum and J. Ousterhout, ''The LFS Storage Manager,'' *Proceedings of the Usenix Conference,* Summer 1990.

[Patterson]
D. Patterson, G. Gibson, and R. Katz, ''A Case for Redundant Arrays of Inexpensive Disks (RAID),'' *Report No. UCB/CSD 87/391,* December 1987.

[Peacock]
K. Peacock, ''The CounterPoint Fast File System'' *Proceedings of the Usenix Conference,* Winter 1988.

[Ritchie]
D. Ritchie and K. Thompson, ''The Unix Time–Sharing System,'' *Bell System Technical Journal, 57(6),* July–August 1978.

[Thompson]
K. Thompson, ''Unix Implementation,'' *Bell System Technical Journal,* 57(6), July–August 1978.

Larry McVoy is currently a Member of Technical Staff in the Operating Systems Technology Department at Sun Microsystems. He received M.S. in 1987 and B.S. in 1985 in Computer Science from the University of Wisconsin at Madison. Since then, he has ported Unix to a super computer, brought up TCP/IP on machines ranging from 80386 to a super computer, added POSIX conformance to SunOS, and lectured at Stanford University on Operating Systems. Since joining Sun, he has been improving the performance of the VM and file subsystems of SunOS. He may be reached by electronic mail at *lm@Eng.Sun.COM*, by phone at (415) 336-7627, or by mail at MS 5-44, 2550 Garcia Ave., Mountain View, CA, 94043.

Steve Kleiman is currently a Distinguished Engineer in the Operating Systems Technology Department of Sun Microsystems. He received an M.S. in Electrical Engineering from Stanford University in 1978 and a B.S. in Electrical Engineering and Computer Science from M.I.T in 1977. He has been involved with the design and development UNIX and workstation architecture since 1977; first at Bell Telephone Laboratories and then at Sun. He was one of the developers of NFS, Vnodes, and the original port of SunOS to SPARC. His electronic mail address is *srk@Eng.Sun.COM*.