Clockwise: A Mixed-Media File System

Peter Bosch, Sape J. Mullender, Pierre G. Jansen
University of Twente, Netherlands
peterb@huygens.org, sape@huygens.org, jansen@cs.utwente.nl

Abstract

This (short) paper presents the Clockwise, a mixed-media file system. The primary goal of the Clockwise is to provide a storage architecture that supports the storage and retrieval of best-effort and real-time file system data. Clockwise provides an abstraction called a dynamic partition that groups lists of related (large) blocks on one or more disks. Dynamic partition can grow and shrink in size and reading or writing of dynamic partitions can be scheduled explicitly. With respect to scheduling, Clockwise uses a novel strategy to pre-calculate schedule slack time and it schedules best-effort requests before queued real-time requests in this slack time.

1 Introduction

Clockwise is a mixed-media file server that is developed as part of the ESPRIT Pegasus[1] Project [14] and demonstrates:

(1) It is possible to build a high-throughput server for bulk data cost-effectively, using off-the-shelf (PC) hardware;
(2) It is possible to mix real-time (audio and video) data and conventional best-effort data in a single storage architecture in such a way that best-effort latencies are minimized without missing any of the real-time traffic deadlines;
(3) It is possible to build a file system that prevents Quality of Service crosstalk; that is, misbehaving applications cannot disrupt the performance and deadline guarantees of other applications.

Clockwise achieves these goals through a number of techniques. First, high data rates for bulk data transport can only be achieved by carefully tailoring data transfer mechanisms to the underlying hardware. We learnt *how* to do this by measuring and understanding the performance opportunities and limitations of current hardware before committing to a system architecture; by doing this meticulously, we could obtain server performance close to the raw hardware performance.

Clockwise schedules disk requests both on deadline and on latency. Deadlines are associated with real-time data requests, latency has relevance to conventional best-effort data requests. Clockwise has incorporated a scheduler that meets the deadlines of real-time traffic and minimizes the latency of best-effort traffic.

Disk bandwidth, CPU bandwidth, network bandwidth, and buffer space are critical resources that require explicit allocation and scheduling by the system. Clockwise schedules disk bandwidth and buffer space explicitly, and it relies on the host operating system, Nemesis [7], to schedule the CPU and network explicitly [10, 2] and thus gives performance guarantees to its clients. Clients, in this case, are storage applications that run on the server and provide real-time or best-effort data storage. Such storage applications usually communicate with remote clients over the network. The storage applications are not part of Clockwise *per se*: they run as independent processes or *domains* on Nemesis. Should a storage application crash, other storage applications and Clockwise itself are not affected.

The reason for developing Clockwise is because we know of no other system that addresses the three issues in a single solution. There exist many projects and systems that have solved parts of what Clockwise offers. Where possible we have re-used ideas from other projects and combined them in Clockwise. In particular, Clockwise resembles Symphony [13] and Barham's *User-Safe Disk* (USD) [1]. Symphony implements an integrated best-effort and continuous-media file server with support for multi-traffic-class caching, data layout policies and failure recovery. USD partitions the disk and allows client applications to directly access the disk resources.

[1]The Pegasus Project is a project, initially of the Universities of Twente and Cambridge, now also of the University of Glasgow, the Swedish Institute of Computer Science, and APM Ltd., supported by the European Communities' ESPRIT Program-me through BRA project 6586 (1992 – 1995) and LTR project *21917* (1996 – 1999).

Clockwise has three layers, the host operating system Nemesis, the Clockwise layer and a storage application layer. The storage applications implement the semantics of the data stored in dynamic partitions, the access control and the policies concerning timing, buffering and caching.

Clockwise accounts, schedules and polices the storage resources (disk and physical memory) and performs the admission test for new tasks. Without this test, Clockwise cannot give any performance guarantees. Nemesis controls and polices the CPU and, in some cases, the network resource.

The remainder of this paper describes Clockise in detail. In Section 2, we present Clockwise by topic. A short overview of simulated and measured performance results from Clockwise are presented in Section 3. Finally, Section 4 summarizes this paper.

2 Clockwise by topic

A good way to view Clockwise is as a disk scheduler. Applications can request real-time (continuous-media) or best-effort service from Clockwise. For real-time service, an application must reserve bandwidth through Clockwise. Clockwise admits reservations as long as it can guarantee the requested bandwidth — that is, as long as it can meet the applications' deadlines.

Since we wanted to cater at least for high bandwidth applications, such as continuous-media applications, Clockwise specializes in reading and writing large blocks of data (avoiding delays due to seek operations and rotational delays) in such a way that the CPU is never in the data path.

As a consequence, we differentiate between in-band and out-of-band data transfers. In-band (and bulky) data flows through the server machine by using DMA chips without requiring the CPU to alter, read or copy the data for data that travels between the network and disk. Out-of-band data is used for signaling purposes, such as setting up of scheduling parameters, calculating placement of disk blocks on disk or any other *meta* operation.

Clockwise also manages memory buffers between the various storage applications. Applications allocate and free data buffers through Clockwise. These memory buffers can be used as "DMA" buffers — when a memory address is presented to Clockwise for a read or write operation, Clockwise only needs to verify that the address is a Clockwise buffer before inserting the pointer into a DMA device. To prevent denial of service attacks, Clockwise has the ability the take away buffers when applications have allocated excessive quantities of memory buffers.

Clockwise currently hosts two types of storage applications: continuous-media storage applications and the NetBSD *Virtual File System* (VFS) file system which we ported to Nemesis/Clockwise. Continuous-media applications use per-file dynamic partitions to store and retrieve digitized audio and video. VFS file systems implement complete file systems (using one of several types) within a single dynamic partition.

In the remainder of this section describes dynamic partitions and the disk scheduling techniques.

2.1 Dynamic partitions

Clockwise organizes data in *dynamic partitions*. From the application's viewpoint, a dynamic partition presents sequential storage capacity. It is made up of large disk blocks (currently 1 MB per block) and it can grow or shrink dynamically by adding or removing blocks. In this, it resembles Loge [4] and Logical Disk [3]. Dynamic partitions can span multiple disks. Internally, Clockwise maintains a dynamic-partition table. For each dynamic partition, the table lists the name and number of the partition and a list of the megabyte blocks allocated to it.

Before a dynamic partition can be used, an application must open it. If the application requests isochronous service, at open, Clockwise carries out a schedulability test which determines whether sufficient resources to service the application are available. The application presents the requested throughput b and the block size B for transfers. These numbers, together with the partition's layout on disk are run through a schedulability analysis. The analysis finds out whether the disks serving the partition have sufficient unallocated resources to accommodate the new task.

The service time is calculated from the block size B, and a disk-performance table. For each zone on each disk, and for a range of block sizes, Clockwise maintains a table of 95-percentile service times and a list of seek service times. The service rate p is the quotient of throughput and block size $p = \frac{B}{b}$. For the schedulability analysis we assume that each request is preceded by a full seek.

When a request on a dynamic partition spans multiple disk zones, Clockwise makes the schedulability analysis for the worst case (the innermost zone). Although this strategy under-utilizes the disk, we do not expect that a dynamic partition frequently spans multiple zones: when disk space is reserved, or when the dynamic partitions are reorganized, related blocks are allocated close to each other and preferably in the same zone.

When a new dynamic partition is created, an application can either use the default strategy in assigning blocks to dynamic partitions, or it can devise its own al-

location policy. In the former case, Clockwise rotates the blocks through a user-supplied maximum number of disks. The load is equally distributed over these disks and the maximum throughput of a dynamic partition corresponds to the aggregate disk performance of the parallel disks.

An application can also decide to allocate blocks explicitly to certain locations on disk(s) much like Symphony's disk location hints. McKusick's FFS, for example, tries to coalesce related blocks in the same cylinder group on a physical disk. The equivalent of a cylinder group in Clockwise are a number of dynamic partition blocks in a dynamic partition that are located consecutively on a disk. An application can instruct Clockwise to allocate a consecutive range of blocks in a dynamic partition on one disk.

2.2 Disk QoS

Clockwise uses the storage applications' periods and service times as input for a schedulability test. The schedulability test is based on earlier work by Jeffay *et al.* [5] that proves that if a task set satisfies two conditions, the task set can be scheduled by a non-preemptive *Earliest Deadline First* (EDF) [8] algorithm. The two conditions place restrictions on the total load. The first condition is identical to the preemptive EDF condition. The latter condition makes sure that there is always enough computational power available to meet all of the deadline regardless of their phasing (relative start time) without preempting an earlier started request.

Furthermore, Clockwise pre-calculates the minimal slack time in a schedule. Clockwise uses this minimal slack time to schedule best-effort requests before real-time requests is not all of the slack time has been used up already. This way Clockwise can prioritize unadmitted requests without endangering the deadlines of real-time requests[2].

Symphony [13] is also a system that combines best-effort files with media files. The Symphony scheduler maintains an EDF queue and calculates the *Latest-Start-Time* (LST) of requests in the real-time queue and when there is enough time before a real-time request, Symphony schedules a best-effort request before a real-time request.

The difference between our approach and Symphony's or Cello's is that we compute on beforehand the minimum slack time that can be used freely in the on-line schedule. Symphony calculates the LST of the real-time queue whenever a request is queued. When a scheduling decision needs to be made for a stream that uses too many resources or when a best-effort task needs to be scheduled, Clockwise can decide on the spot whether or not executing the request violates other real-time guarantees. The amount of free scheduling space is a result from the schedulability analysis, and it turns out that the minimal slack-time calculations fundamental: without knowledge of the minimal slack-time prioritizing best-effort traffic over real-time traffic may lead to deadline misses.

[2]The exact details of this approach are beyond the scope of this paper and will be published separately.

2.3 Memory QoS

The second important task for Clockwise is memory management. When a number of storage applications run on Clockwise, each one of them can have a different need for the memory resources. A media application, for example, requires only a few buffers to provide disk buffering. In some cases, a media application caches data, and requires extra buffers. The VFS file systems, on the other hand, require as much memory as they can get to improve their cache hit rate. Other applications, such as database or web servers may require yet another usage for the memory.

Instead of forcing a single memory-allocation policy onto all of the storage applications, Clockwise allows storage applications to manage their own data buffers. For this, Clockwise has defined two types of memory: fixed and variable buffers. *Fixed buffers* are allocated as part of the quality of service negotiation and remain in the possession of the storage application for as long as the storage application wants to keep the buffers. Typically, these buffers are used in a double-buffering scheme when playing or recording audio or video, or as a VFS dirty-block cache.

If an application temporarily requires more buffers, it can request *variable buffers* from Clockwise. A VFS uses the variable buffers to enlarge the cache space. A media application uses them to move peak load to less demanding periods. Clockwise services these variable buffers through a *'fair-share policy'*: each application that requires variable buffers can get a fair part of the available memory based on its fixed buffer usage. We view the fixed buffer usage as a measure for the average buffer usage. Each application is entitled to the same ratio of variable-to-fixed buffers.

To prevent applications from hogging memory buffers, Clockwise is able to revoke the rights to variable buffers. For this, Clockwise first tells the storage application to return some memory buffers. The application is given some time to react (*e.g.* to flush data to disk) and if the application does not respond within a certain time, Clockwise simply deallocates some buffers for them.

2.4 Media storage

Clockwise dynamic partitions were initially designed for the storage and retrieval of media data to and from disk. A media application receives the digitized, and usually compressed, data from a network and stores the data in a newly allocated and private dynamic partition.

Clockwise media applications are applications that run as independent processes on the Clockwise server. The reason for running the media applications as user processes on the server is threefold. First, Nemesis, Clockwise's host operating system, can start applications in their own domain with their own CPU quality-of-service parameters. A Nemesis process receives its allocated quality of service, no matter how many resources other processes attempt to consume. Secondly, by running the media storage applications in separate processes, an error in one such application does not bring down the service as a whole. Lastly, when new types of media are incorporated, only a new media application needs to be installed at the server; Clockwise need not be changed.

Given the "DMA" interface of Clockwise, media applications only orchestrate the DMA transfers from network to memory and further onto the disk or *vice versa*. This implies that media applications do not require much CPU attention, and to deliver a motion-JPEG data stream to a remote client over an ATM network, only 2% of the CPU is required.

2.5 Conventional File Systems

We ported the NetBSD *Virtual File System* (VFS) [6] to Clockwise and run it as a best-effort conventional file system. The VFS contains many file-system types such as McKusick's FFS [9], BSD LFS [11, 12] and a NetBSD version of EXT2FS. The VFS interface combines all file-system types into a common file-system tree, and can be accessed similarly.

There were three reasons for choosing the NetBSD VFS tree. First, by porting the VFS structure, Clockwise is able to use many different storage algorithms. Secondly, NetBSD's implementation of VFS is a clean implementation: most of the file-system code compiled with only slight modifications as a Nemesis user application even though Nemesis is completely different from UNIX. Finally, by porting stable and widespread file-system code we immediately had a reliable file system which, because of this, is used more and thus gave us more performance feedback.

Each VFS running on Clockwise has access to a private dynamic partition. Before a VFS can run, a dynamic partition and disk space is allocated and an initial file-system structure is written to the dynamic partition. Disk-space reservation can be based on the characteristics of the file-system structure. Disk-space allocation for an FFS, for example, makes use of cylinder groups: each disk in the Clockwise array hold groups of consecutive megabyte blocks on one disk to store FFS cylinder groups.

Block caching is implemented through Clockwise memory buffers instead of the Unix memory allocator. When a VFS boots, it requests a fixed buffer set for caching purposes. Next, VFS periodically allocates more memory buffers until the memory space is exhausted. Each VFS also installs a handler with Clockwise for call-back purposes: when a VFS needs to release some buffers, the handler is called with the amount of buffer space that needs to be released.

3 Performance

Performance measurements on Clockwise are ongoing work. We are measuring in two areas in particular: how well can best-effort and real-time loads be integrated and what is the maximum bulk-data transfer rate on Clockwise machines.

To measure how well Clockwise can schedule a mixed-media load, we are using disk traces from HP Laboratories [15] and real-time block timings from our own server in a disk simulator. The disk simulator simulates the behaviour of a (200MHz) Pentium-Pro based machine, which is equipped with 3 Quantum Atlas-II disks. The simulator executes best-effort and real-time requests and the latencies of the best-effort requests are measured. To validate the simulated and measured latencies, short parts of the HP traces are re-executed on a real Clockwise. Only when the latencies match, conclusions are drawn from the simulated results.

By performing combined real-time and best-effort file system experiments we learnt that Clockwise can schedule up to 17MB/s of real-time traffic on 3 parallel Quantum Atlas-II disks, while at the same time executing up to 4,000 best-effort I/Os requests per minute. We learnt that, compared to Symphony scheduling, Clockwise's disk scheduler performs better when real-time loads are high. Also, by performing the measurements we learnt that Symphony cannot guarantee deadlines simply because its scheduler has not notion of minimal slack time. This minimal slack time turns out to be a fundamental property when scheduling a combination of best-effort and real-time traffic.

As said, the measurements were performed on three parallel Quantum Atlas-II disks. From a performance perspective these disks are not interesting: they operate at a maximum measured speed of approximately 9.5MB/s each. A second experiment has been performed

with a set of 6 parallel Seagate Cheetah disks. A quick measurement showed that we can schedule 6 parallel Cheetahs at a rate of 78.5MB/s for sequentual 1MB transfers. The mean transfer rate that we measured is approximately 80MB/s. For this measurement we did not include seek overhead. When seek overhead is also accounted for the guaranteed performance is approximately 25% less.

4 Summary and lessons

We have presented Clockwise, a mixed-media file system that is able to deliver best-effort and real-time load by the same server. The contribution of this work is that disks and memory can be scheduled dynamically and explicitly without sacrificing raw disk bandwidth, and that cheap hardware can be used for high performance file service.

Dynamic partitions as storage abstractions are a good way of organizing disk partitions. Storage applications use dynamic partitions as if they are reading and writing to the raw disk, while the disk blocks may be scattered over an entire array of disks. Blocks can be re-arranged for performance optimization without the storage application's knowledge. On the other hand, Clockwise does not hide the location of data on disks; an application can inform Clockwise how to organize data on disks.

Clockwise's scheduler guarantees that it meets all real-time deadline while it optimizes best-effort traffic. The scheduler calculates in advance how much slack time is available for best-effort jobs and it uses this slack time to prioritize best-effort traffic.

Clockwise now exists and is used by the authors for experiments and demonstrations. Future work to Clockwise involves a thorough analysis of the scheduler, a full performance analysis, the integration of a CD-ROM/DVD jukebox into Clockwise and a better integration of client machinery and client caching/buffering. Also, Clockwise is currently being ported to Linux and will be made available.

References

[1] Paul Barham. A Fresh Approach to Filesystem Quality of Service. Pages 119–128.

[2] Richard Black. *Explicit Network Scheduling*. PhD thesis, published as Technical report TR361. University of Cambidge, April 1995.

[3] Wiebren de Jonge, M. Frans Kaashoek, and Wilson C. Hsieh. Logical disk: a simple new approach to improving file system performance. Technical report IR-325. Dept. of Mathematics and Computer Science,Vrije Universiteit, Amsterdam, 1993. Also MIT/LCS/TR-566 at MIT.

[4] Robert M. English and Alexander A. Stepanov. Loge: A Self-Organizing Disk Controller. *USENIX Conference Proceedings* (San Francisco, CA), pages 237–52. USENIX, Winter 1992.

[5] Kevin Jeffay, Donald F. Stanat, and Charles U. Martel. On Non-Preemptive Scheduling on Periodic and Sporadic Tasks. *Real-Time Systems Symposium*, IEEE TC Real-Time Systems, pages 129–139, 1991.

[6] S. R. Kleiman. Vnodes: An architecture for multiple file system types in Sun UNIX. *1986 Summer USENIX Technical Conference* (Atlanta, GA, June 1986), pages 238–47. USENIX, June 1986.

[7] Ian Leslie, Derek McAuley, Richard Black, Timothy Roscoe, Paul Barham, David Evers, Robin Fairbairns, and Eoin Hyden. The Design and Implementation of an Operating System to Support Distributed Multimedia Applications. *IEEE Journal on Selected Areas in Communication*, **14**(7):1280–97, 1996.

[8] C.L. Liu and James W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM*, **20**(1):46–61, January 1973.

[9] Marshall K. McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. A fast file system for UNIX. *ACM Transactions on Computer Systems*, **2**(3):181–97, August 1984.

[10] Timothy Roscoe. *The Structure of a Multi-Service Operating System*. PhD thesis, published as TR-376. University of Cambridge, April 1995.

[11] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *Proceedings of the 13th ACM Symposium on Operating Systems Principles* (Pacific Grove CA (USA)), volume 25, number 5 of Operating Systems Review, pages 1–15, October 1991.

[12] Margo Ilene Seltzer. *File system performance and transaction support*. PhD thesis. University of California, 1992.

[13] Prashant J. Shenoy, Pawan Goyal, Sriram S. Rao, and Harrick M. Vin. Symphony: An Integrated Multimedia File System. http://www.cs.utexas.edu/users/dmcl. University of Texas at Austin, 1996.

[14] Sape J. Mullender, Ian M. Leslie, and Derek McAuley, *Operating System Support for Distributed Multimedia, Usenix 1994 Summer Conference* (Boston). Usenix, June 1994.

[15] John Wilkes, Richard Golding, Carl Staelin, and Tim Sullivan. The HP AutoRAID Hierarchical Storage System. *ACM Transactions on Computer Systems*, **14**(1):108–136, February 1996.