Scheduling Real-Time Disk Transfers for Continuous Media Applications

Darrell D. E. Long, Madhukar N. Thakur
University of California
Santa Cruz, California

Abstract

We study how continuous media data can be stored and accessed in the Swift distributed I/O architecture. We provide a scheme for scheduling real-time data transfers that satisfies the strict requirements of continuous-media applications. Our scheme allows large data objects to be stored and retrieved concurrently from multiple disks to satisfy the high data rate requirements which are typical of real-time video and audio data. To do this, data transfer requests are split into smaller requests which are then handled by the various components of Swift.

We study on-line algorithms that respond to a data request by promising to either satisfy or reject it. Each response must be made before the next request is seen by the algorithm. We discuss two different performance measures to evaluate such algorithms and show that no on-line algorithm can optimize these criteria to less than a constant fraction of the optimal. Finally, we propose an algorithm for handling such requests on-line and the related data structures.

Introduction

Advances in high speed networking and storage technology will soon make it possible to use data in the form of continuous media (CM), such as real-time digital audio and video, in computing applications. The characteristics of CM data are vastly different from those of the I/O streams that the current generation of distributed systems are capable of supporting.

As the term continuous media indicates, the storage and retrieval of such data must be continuous in real-time. This requires the file system, along with the storage media, to be fast enough to guarantee the data transfer rates that the application demands. Typically, continuous media applications require large data transfer rates, which may vary from 1.2 megabytes/second for DVI compressed video to 90 megabytes/second for uncompressed, full-frame color video. Architectures like Swift [1, 2] and RAID [3] stripe files over several disks, and drive the disks in parallel to achieve high data rates. Continuous media data also have large file sizes. A file system dealing with such files must provide mechanisms for manipulating large data objects. For example, ten minutes of video at 30 frames per second and one megabyte per frame requires a file size of 18 gigabytes.

We study how an array of disks and associated I/O agents can guarantee to read or write data at the transfer rates required by an application. In the past, researchers have investigated other aspects of the design of an operating system to handle CM applications. Govindan and Anderson [4] investigated CPU scheduling and IPC mechanisms for operating systems for CM applications. Little and Ghafoor [5] studied formal specification and modeling of multimedia objects using a logic based on temporal intervals and Petri Nets.

Our aim is to study how the Swift [1, 2] architecture can be used efficiently to read and write CM data objects. Swift is designed to support high data rates in a general-purpose, distributed system. It is built on the notion of striping data over multiple storage agents and driving them in parallel. It assumes that data objects are produced and consumed by clients and that the objects are managed by the several components of Swift. In particular, the *distribution agents, storage mediators*, and *storage agents* are involved in planning and actual data transfer operations between the client and an array of disks, which are the principal storage media. We refer the reader to references [1] and [2] for details of the functionality of these components of Swift.

A client application, when reading or recording CM objects, decides on its data demands in advance and makes a request to pre-allocate I/O resources. This request is called a *client-job*. The client-job is successively broken down into smaller tasks called *braids*, *ropes*, and *strands*.

An implementation of Swift operates as follows: when a client issues a client-job, the storage mediator responds to the request by promising to either satisfy or reject it. In the case of acceptance, the storage mediator creates a *transfer plan*, which is executed by the distribution agent at the appropriate time.

A transfer plan is a sequence of braids with some pertinent timing information. There is one braid per storage agent. A braid, in turn, consists of smaller data transfer specifications, called ropes, one rope per disk. A rope is further split into strands, a strand is the smallest unit of specification in our model. It contains the details for the actual transfer of one block of data. A rope is simply a collection of strands requiring data transfer from the same disk.

1051-9173/93 $3.00 ©1993 IEEE

A strand is satisfied if the request can be honored by the I/O system subject to all its constraints. We say that a rope is satisfied if all the strands in it are satisfied. Similarly, a braid is satisfied if all ropes in it are satisfied. The I/O system commits to satisfy all the braids, or refuses, in which case the application must pursue a different course of action. It is essential that the I/O system keeps its commitments.

A good algorithm will also try to satisfy as many client-jobs as possible. Since the client-jobs, and hence the braids and ropes, arrive in an on-line fashion, the algorithm is required to make a commitment on a rope, before the next rope arrives. Later, we discuss the measures of performance of on-line algorithms in general, and the reason why every on-line algorithm for this problem can be far away from the optimal in the worst case.

Data Transfer in Swift

A *client-job* is a request by a client to initiate a data transfer. Each client-job contains the name of the file, a starting position, the number of bytes to be transferred, and whether the request is to read or write. It also contains three other parameters, T_{client}, the time at which to start the data transfer, R_{client}, the average transfer rate, and b, the block size. The client will read data one block at a time; the block size could reflect the natural granularity in CM data.

Time is assumed to be discrete and measured in steps of appropriately small units such as microseconds. The I/O subsystem can consume or produce data at rates vastly different from the desired rate when measured over small intervals of time. This is due to the jitter in the data transfer rate. But at the end of a period equal to the time required to transfer a block, the client expects one block of data.

Depending on the block size and the amount of memory available to the distribution agent, the storage mediator decides the number of buffers, each holding one block. More buffers allow for better scheduling the strands by the disk controller. These buffers, denoted $B_0, \cdots, B_{k-1}$, are managed by the distribution agent, and are co-resident with the client. The goal of the transfer plan is to keep B_i full with the appropriate block of data when the client expects to read it, or to store B_i when the client is finished writing to it. Data is transferred between the buffers and the storage agents over a fast network.

A transfer plan is simply a sequence of braids, and a braid is broken down into finer specifications called ropes and strands. A strand is a specification to transfer a specific block of data to given buffer. Each strand contains the block size, the disk address of the block, a buffer identifier, and whether the request is to read or write. It also contains two time specifications: T_{start}, the earliest time that the strand can be serviced, and T_{end}, the time by which the service must be complete.

Given a client-job, we describe how a storage mediator computes the braids necessary to satisfy it. Let n be the number of blocks of data to be read or written by the client. The storage mediator first constructs a sequence of n strands, one for each block. Using the layout of the data object, the storage agents and the disks they manage, the storage mediator computes the start address of each block. The ropes and the braids are then composed from these strands.

The buffers are used in a circular manner. These buffers allow the client to read or write at its own pace without being affected by the variations in the transfer rate provided by the disk. They also allow the disk the flexibility to schedule data transfer in a wider time window.

We assume that the time taken to transfer one byte of data over the network is bounded by a constant D. The underlying network protocols are assumed to provide for a uniform transfer time without significant variance [6].

We describe next how to compute T_{start} and T_{end} for each strand. We consider the read and write cases separately for this computation. In the following discussion, strand i relates to buffer B_j, with $j = i \bmod k$ and the block i of data. Let a be the time taken by the client to process one buffer of data.

In the case where the client is reading data, it is essential that the buffers are available to the I/O subsystem for a period before the client reads the first block of data, that is, before T_{client}. This period is used to fill the buffers before the client reads them. To allow for a general framework, we introduce an initial delay, T_{init}, which is at least as long as the time required to fill the first buffer. This is decided by the storage mediator when it starts computing the specifications of a transfer plan. Hence the time to start execution of the plan is $T_{plan} = T_{client} - T_{init} - bD$, where bD is the time taken to transfer a block over the network.

The client expects a buffer B_j to be read after processing all the other k-1 buffers once. Since, it takes a time units to process one buffer of data, the client will access the buffer B_j after every ka time units. When the client is reading data from the buffer for the first time (that is for block i, $0 \leq i \leq k-1$), the system should load the buffer after the time T_{plan} and before the client reads it. For subsequent reads of the buffer B_j, the system should load data after the client has read it once and before it reads it again after ka time units. These constraints allow us to compute the times T_{start} and T_{end} for the strand i.

In the case when the client is writing data, $T_{plan} = T_{client}$, the system can store the data from a particular buffer, say B_j, to the disk after the client has written to it and before it starts writing into B_j, again after ka time units. In case the client is writing to B_j for the last time, the system should store B_j to the disk before it is no longer available to the client. To allow for time to copy the buffers, we introduce T_{clean}, a clean up time which is at least as long as the time required to copy a buffer B_i to the disk. Again, using these constraints, we can compute the times T_{start}, T_{end} during which the system has to start and finish writing to the disk the block i of the data.

We have specified above, how a client-job is broken down into smaller and more specific tasks by the storage mediator and how to compute the attributes of a strand, which is the smallest unit of data transfer. The storage mediator rejects the client-job if it is unable to fulfill all the requirements specified by the client. In case of acceptance, the storage mediator presents a transfer plan to the distribution agent. In order to accept the client-job, the storage agents must promise to satisfy the requirements of the braids presented to them. The storage agents can make such a promise if they can obtain a promise from all their disks to satisfy the ropes presented to them. The decision of the disk controller is the key to acceptance or rejection of the client-job. It has to be consistent with the promises made in the past: any promise to accept a rope must not invalidate previous promises.

The disk controller needs an on-line algorithm to reply to a rope. The algorithm is on-line because the reply must be made without knowing the future ropes that it may receive. A reply once made cannot be countermanded.

Guaranteeing Good Service from the Disk

To check if an arriving strand is satisfiable, an algorithm must know the time required for the disk to transfer one block of data. For this, it needs information about the disk layout and other disk parameters. Depending on the disk model used, we can derive estimates of time required to read m bytes from a disk. We denote by $T(m)$ the time taken to transfer m bytes of data to or from a disk. It may be difficult to accurately model disks analytically, and so simulation studies may provide the best method to obtain an estimate of $T(m)$ [7].

Circular buffering at the disk controller

If the disk controller commits to satisfy a strand and actually starts data transfer at some time, say T_0, the controller must guarantee to transfer a block of b bytes starting at T_0, ($T_{start} \leq T_0$) and ending before $T_0 + T(b)$. This data transfer has to be achieved in spite of disk jitter. The actual time taken to transfer b blocks of data could vary from the computed value because the actual disk parameters could be slightly different from the ones used to compute $T(m)$ above.

One way to practically deal with this problem is by buffering the data at the disk controller and have synchronization during the data transfer process. This will ensure that the appropriate buffer of the distribution agent experiences the data flow at the proper time and without large variations in the data rate. In our opinion, this is the best strategy to deal with the variance in the disk transfer rate because we are interested in providing guaranteed service to the client. Once a transfer plan is presented to the distribution agent, all its strands must meet the demands specified for them. Buffering data at the disk controller and presenting it to the storage agent will provide the required guarantees unless the variation in the disk transfer rate is high.

We use buffers D_0, D_1, $\cdots$ each of the size c, where c is the size of a cylinder in bytes. The number of buffers is chosen depending on the size of memory available and the amount of variation in the disk transfer rate. The storage agent transfers data across the network, from the buffers D_i in a circular fashion to the buffers B_j managed by the distribution agent.

As a concrete example, we consider how two buffers D_0 and D_1 are used and the generalization to more buffers is straightforward. For the case, when the strand is a request to read data from the disk, the disk controller fills up first buffer D_0, in $c/T(b)$ time duration, starting at $T_0 - c/T(b)$. It then transfers the data to the distribution agent across the network. The storage agent waits for $c/T(b)$ time units and then it starts reading the data from D_0. If the disk is transferring data faster than expected, it could fill up buffer D_0 in less than $T(b)$ time units and then it could go ahead and fill buffer D_1. But it should not fill up the buffer D_0 until the storage agent has finished reading the data from there, which it will, at time $T_0 + c/T(b)$. Similarly, the data is read into D_1, and then into D_0 again in a circular way.

The disk controller manages synchronization with the storage agent at the appropriate points in time. Initially if the disk takes less than $T(b)$ time units to read data into the buffer D_0, it could take more time to read the next c bytes of data, as long as the total time taken to read $2c$ bytes is $2T(b)$. This allows for the small variation in the disk transfer rate.

Writing is similar to the process of reading. In this case, data is copied from the distribution agent's buffers B_i to the buffers D_j maintained by the disk controller and eventually onto the disk. The buffers D_j, in this case too, serve to avoid data loss in case of jitter in the disk transfer rate.

On-Line Scheduling Algorithms

On-line or real-time algorithms have been studied theoretically, with an aim of proving performance bounds. We briefly discuss on-line algorithms in general, different measures of performance of such algorithms, and then propose one such algorithm to satisfy a rope.

Abbott and Garcia-Molina [8] have studied real-time scheduling of transactions with deadlines on a single processor memory resident database system. Shih, Liu and Liu [9] worked on the problem of real-time scheduling periodic jobs which have deferred deadlines. Dertouzos and Mok [10] have studied the problem of on-line scheduling of real-time tasks in a multiprocessor environment. They also show that optimal scheduling without a priori knowledge of the input is impossible.

On-line algorithms have also been studied analytically in other contexts [11, 12]. Many data structure problems are on-line, including scheduling problems, caching problems, and others. Karlin, Manasse, Rudolph, and Sleator [13] studied on-line algorithms for caching problems. They also coined the term *c-competitive* algorithm, to refer to an on-line algorithm which always performs within a constant multiplicative factor, c, of the optimum on any sequence of requests.

Informally, an algorithm A is c-competitive with respect to some performance measure, if for any input sequence, A always achieves performance that is within a constant (multiplicative) factor c, of that achieved by a best off line algorithm. Stated another way, if B is an off-line algorithm, the ratio of the performance of A to the performance of B is always bounded by a constant. This definition does not specify the actual performance measure, but gives us a way of comparing the performance of two algorithms in general. The performance measure that is chosen depends on the criteria deemed important to the problem under consideration.

It is our interest to study on-line algorithms for the rope satisfiability problem, which is to respond to a rope. The algorithm must check whether the individual strands, in the rope presented to it, are satisfiable given the current set of commitments made by the algorithm. It should commit to satisfy the rope if every strand can be satisfied. While, seemingly, there has been some related work [14,9] on real-time scheduling of tasks, we cannot use their techniques because we have to schedule ropes which not only have a deadline, but also an earliest time before which they cannot be scheduled. In short, the abstract scheduling problem that arises from the rope satisfiability problem, is to schedule tasks within a time window.

We discuss next two performance measures for the rope satisfiability problem and show, using adversary arguments, that no algorithm can be c-competitive, for any constant c, with respect to either of these performance measures. For the worst case examples required in our adversary arguments, we need only ropes with a single strand.

An interesting performance measure is the number of ropes that can be satisfied by a given on-line algorithm ^{C}. We argue that ^{C} is not a c-competitive algorithm, for any constant c. Let B be the best off-line algorithm. To show this, assume that there is an adversary generating the sequence of ropes. The first rope that the adversary presents is such that it must start at time 1, requires time $c+2$ to satisfy, and, therefore, must be completed by time $c+3$. If algorithm ^{C} commits to satisfy this rope, then the adversary will present a sequence of $c+2$ ropes, all of which require unit time to service, and which follow one another sequentially starting at time 1. Since ^{C} is busy in the interval $[1, c+3]$ it must refuse these ropes. Then the ratio of the performances of A and B is less than $1/c$.

On the other hand, if ^{C} refuses to satisfy this first rope, then the adversary ends the sequence immediately with this rope. While B satisfies the sequence, ^{C} does not, and so the ratio of the performances of A and B is 0. This argument shows that any on-line algorithm ^{C} cannot be a c-competitive algorithm for any constant c.

As a result, there is no good on-line algorithm for this problem, as long as the performance is measured as the number of ropes satisfied. To investigate if this pessimistic scenario is just due to the objective function, or is partly due to some deeper nature of the on-line setting of this problem, we have studied on-line algorithms with respect to another performance measure.

We let the performance measure be the total time for which the disk is busy when the algorithm ^{C} satisfies the sequence of ropes. As before, let ^{C} be an algorithm that accepts a sequence of ropes on-line and commits or refuses to satisfy each rope. Using similar arguments as above, we can prove that ^{C} is not a c-competitive algorithm, for any constant c. Results such as these give strong evidence that this problem of satisfying ropes is inherently intractable in the on-line setting, as ignorance about the future leads to on-line algorithms that are not c-competitive for any constant c.

An on-line algorithm for the rope satisfiability problem

Below is a simple on-line algorithm to respond to a rope which is a sequence of strands $[s_1, s_2, \cdots, s_k]$. The algorithm checks if the individual strands are satisfiable given the current set of commitments made by the algorithm. It commits to satisfy a rope if every strand in it is satisfiable. So, we need only describe the algorithm to satisfy a single strand.

We say a disk is *busy* during a time interval for a strand if it has to be involved with data transfer at any point during that interval to satisfy another strand that the storage agent has committed.

As no algorithm can be c-competitive, we take a simple minded approach and use the algorithm a described below. This is a first-fit algorithm that commits or refuses to satisfy a strand s with block size b and start and end times denoted by T_{start} and T_{end} respectively.

Algorithm *A*.
Input: A strand s.

1. Compute $T(b)$ for the strand s.
2. Find the earliest sub-interval of $[T_{start}, T_{end}]$ which is not busy and of duration $T(b)$.
3. If such an interval is found, commit to satisfy s, else refuse.

If we let the performance measure be the number of strands accepted by an algorithm, then we can show, using simple arguments, that the worst case (over all sequences of strands) competitive factor of our algorithm is a function of the smallest data transfer time required by any strand in the sequence and the total time the disk is kept busy by the algorithm. This is not bounded by any constant independent of the sequence of strands.

The algorithm a needs efficient data structures to store and access information about the time intervals, when the disk will be busy. We store the set of time intervals when the disk is busy in a height balanced 2-3 tree and call it the *busy tree*. The leaves of this tree store the time intervals and are joined in a doubly linked list. The time intervals are ordered according to their start times, that is, $[a_1, b_1] \leq [a_2, b_2]$ if and only if $a_1 \leq a_2$. On receiving a strand s requiring block size b and start and end times given by T_{start} and T_{end} respectively, the First Fit algorithm a computes $T(b)$ using an appropriate disk model or obtains its value from simulation studies. To find the first interval of duration $T(b)$ fully contained in the interval $[T_{start}, T_{end}]$, algorithm a accesses the *busy-tree* using the following procedure b.

Periodically, we also clean up the *busy-tree*. The clean up operation deletes all intervals $[a,a']$, such that a' is before the current time. This prunes the tree of unnecessary information from the past.

Procedure *b*.

1. In the *busy-tree* find the first interval $[a,a']$, such that $a \geq T_{start}$.
2. if $a \geq T_{start} + T(b)$ then
   Commit to satisfy strand s starting at time T_{start}.
   Insert $[T_{start}, T_{start} + T(b)]$
   in the *busy-tree*.
   return.
   fi
3. Try to find the next available time interval at which strand s can be committed. Starting at $[a,a']$, scan the doubly linked list of leaves of the *busy-tree* till one of the following occurs:

- There are two neighboring (in the linked list) intervals $[c_1,d_1]$, $[c_2,d_2]$, such that $c_2 - d_1 \geq T(b)$ and $d_1 \leq T_{end} - T(b)$.
  In this case, commit to satisfy strand s starting at time d_1.
  Insert $[d_1, d_1 + T(b)]$ in the *busy-tree*.
- The end of the list is reached or we find $[c,d]$, such that $c > T_{end} - T(b)$.
  In this case, refuse to satisfy the strand s.

Concluding Remarks and Future Work

We have studied scheduling time requests to access data from disk. However, in any system there will also be data requests generated by applications that are not dealing with continuous media data. Such requests may not have a time duration during which they have to be scheduled. It is easy to incorporate such requests in our scheme. Such a request to transfer b bytes of data from a disk is handled by the First Fit algorithm A as a special case of satisfying a strand with T_{start} being the current time and T_{end} being unbounded.

We have studied the problem of on-line scheduling of CM application ropes on a disk. We modeled the problem and found that if we try to maximize the number of ropes scheduled, or the disk utilization, then there is no c-competitive algorithm possible. Hence, we decided to work with a simple minded approach and proposed a First Fit algorithm. Though we have studied this in the context of the Swift architecture, the work is general and could be used in any distributed system.

In conclusion, we suggest that practical considerations may be more important than theoretical worst case bounds for this problem. An average case analysis of the algorithms and data structures involved, with proper probabilistic assumptions should be attempted. It is our opinion that this will be quite difficult and if simplifying assumptions are made, it may be too far from reality to be

useful. Simulation studies, with good data, may be a better approach. We would need traces of the client job activity and the disk accesses made for the simulation model to be realistic. With such information, we feel it is possible to provide probabilistic guarantees that a client job once accepted will be honored.

In our current work, we haven't allowed for the possibility that a strand once scheduled, could be rescheduled within the appropriate time bounds. Such rescheduling could satisfy another strand that arrived later in time that would otherwise go unsatisfied and could improve the performance of the First Fit algorithm. We leave it to future work to study how rescheduling will affect the performance of the algorithm.

Other open problems are to study ways to handle changes in the specifications of a client-job, once it has been scheduled. The client could change the rate R_{client} after the transfer plan has been made by the storage mediator, or the client could change the required data rate during processing of data. This could happen when in an interactive session, the viewer of a real-time video segment decides to view parts of the video in slow-motion or uses the fast-forward mode. Such actions will cause drastic change in the data rate requirements. It is left to future work to at least provide degraded performance to such an application.

Acknowledgments

We wish to thank Luis-Felipe Cabrera, Jeffrey Keller, David Levy, Vikram Sahai, and K. B. Sriram for useful discussions and comments on the earlier drafts of the paper.

This research was supported in part by the National Science Foundation under Grant NSF CCR-9111220 and by the Institute for Scientific Computing Research at Lawrence Livermore National Laboratory.

References

[1] L. F. Cabrera and D. D. E. Long. Using disk striping to provide multiple high I/O data rates. Computing Systems, 4(4):407-438, December 1991.

[2] L. F. Cabrera and D. D. E. Long. Swift: A storage architecture for large objects. In Proceedings of the 11th Symposium on Mass Storage Systems, pages 123-128, Monterey, California, October 1991. IEEE.

[3] D. Patterson, G. Gibson, and R. Katz. A case for redundant arrays of inexpensive disks (RAID). In Proceedings of the ACM SIGMOD Conference, pages 109-116, Chicago, June 1988. ACM.

[4] R. Govindan and D. P. Anderson. Scheduling and IPC mechanisms for continuous media. In Proceedings of the 13th ACM Symposium on Operating Systems Principles, pages 68-80. Association for Computing Machinery SIGOPS, October 1991.

[5] T. D. C. Little and A. Ghafoor. Synchronization and storage models for multimedia objects. IEEE Journal on Selected Areas in Communications, 8(3):413-427, April 1990.

[6] C. Osterbrock, D. D. E. Long, and L. F. Cabrera. Providing performance guarantees in an FDDI network. Submitted for publication, 1992.

[7] C. Ruemmler and J. Wilkes. Disk shuffling. Technical Report HPL-CSP-91-30, Concurrent Systems Project, Hewlett Packard Laboratories, October 1991.

[8] R. Abbott and H. Garcia-Molina. Scheduling real-time transactions: A performance evaluation. In Proceedings of 14th VLDB Conference, 1988.

[9] W. K. Shih, J. W. S. Liu, and C. L. Liu. Modified rate monotone algorithm for scheduling periodic jobs with deferred deadlines. Real-time Systems Newsletter, 7(1-2):17-23, Winter-Spring 1991.

[10] M. L. Dertouzos and A. K. Mok. Multiprocessor on-line scheduling of hard-real-time tasks. IEEE Transactions on Software Engineering, 15(12):1497-1506, December 1989.

[11] M. S. Manasse, L. A. McGeoch, and D. D. Sleator. Competitive algorithms for server problems. Journal of Algorithms, 11:208-230, 1990.

[12] D. D. Sleator and R. E. Tarjan. Amortized efficiency of list update and paging rules. Communications of the ACM, 28(2):202-208, 1985.

[13] A. R. Karlin, M. S. Manasse, L. Rudolph, and D. D. Sleator. Competitive snoopy caching. Algorithmica, 3:79-119, 1988.

[14] S. K. Dhall and C. L. Liu. On a real-time scheduling problem. Operations Research, 26(1):127-140, 1978.