TECHNICAL RESEARCH REPORT

Cues: File Subsystem Data Streams

by S. Gupta, J.S. Baras, S. Kelley,
N. Roussopoulos

T.R. 96-53

Sponsored by
the National Science Foundation
Engineering Research Center Program,
the University of Maryland,
Harvard University,
and Industry

Cues: File Subsystem Data Streams*

Sandeep Gupta† John S. Baras§ Stephen Kelley‡ Nick Roussopoulos†‡
Institute of Systems Research, University of Maryland, College Park, MD 20742
{*sandeep,nick,skelley*}*@cs.umd.edu, baras@isr.umd.edu*

July 2, 1996

Abstract

We present a system call which enables directing high performance data transfers with in-kernel streams. The streams are defined and run using abstract data types called Cues, formed over the Unix file subsystem. The system call, named `cue()` , returns a descriptor, which can be used to write requests to the newly created stream. These requests define the flow of the stream. The abstraction simplifies the design of applications that transfer large amounts of data from files or devices. It also enables high throughput when multiple transfers are in progress. Cue code is compact, modular, and portable. This model also results in a simple mechanism for remotely cueing data flow using standard connections with peer processes. The implementation and tests are also described in this paper.

1 Introduction

In-kernel data streaming [1] [2] is an OS service to enable high performance transfers of data. This is useful when the data being moved need not be modified by the user process transferring it from one file or device to another. We describe a new Unix system call `cue()` , for in-kernel streaming of data.

Tests using the call show improved throughput on the system for concurrent file transfers. It turns out that scheduling of such transfers can be done efficiently in the kernel within a process's time share. The performance measures reported show up to 50% improvement over the transfers with the standard I/O calls. Our abstraction is over the file subsystem. This makes the construct modular, and simplifies the code.

A general discussion on data streaming, including the concept of in-kernel streaming appears in [1]. The case for such a primitive was also made earlier, in [6]. Following this, an implementation of a Unix system call, `splice()`, was described in [2]. It's implementation included a rewrite of the file and network subsystem routines. It also has a scheduling mechanism for I/O from the disk for maintaining the throughput. Data streaming through the Unix kernel can be done at several levels, and in as many ways. Other than the approach used in [2], kernel abstractions such as device drivers, System V streams modules [7] can be, and have been used to stream data through the kernel at

†Also with Department of Computer Science.
‡Also with Institute of Advanced Computer Studies.
§Also with Department of Electrical Engineering.
*This material is based upon work supported in part by the National Science Foundation under Grant No. NSF EEC 94-02384, and by the Center for Satellite and Hybrid Communication Networks under NASA contract NAGW-2777, by the University of Maryland Institute of Advanced Computing Studies, by ARPA under Grant No. F30602–93–C–0177, Maryland Industrial Partnerships and Loral Corporation. Part of the design for this work was also done by the first author as a summer intern at AT&T Bell Laboratories, Murray Hill in Summer '94.

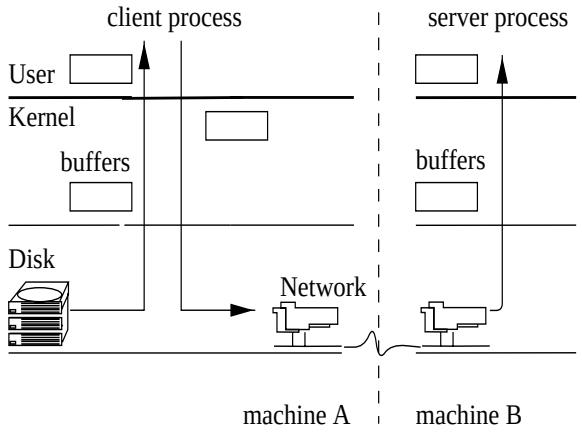

Figure 1: Data path for user driven transfers.

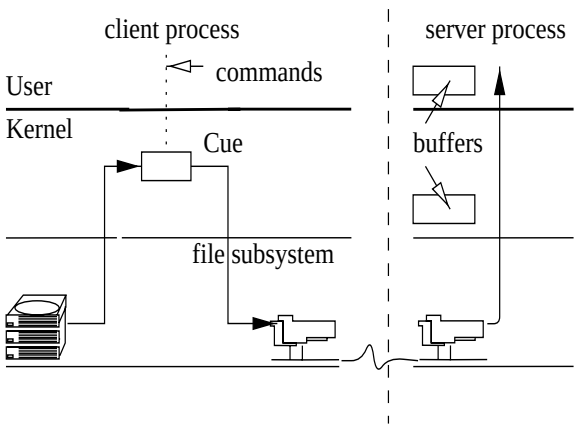

Figure 2: The model for Cue.

interfaces and over protocol modules.

Cues, and splice [2] operate on file descriptors, instead of lower level kernel data structures or protocol modules.

A Cue forms a stream *over* the file subsystem abstraction [5]. The transfer routines use the kernel file objects. This places it just above the *vnode* layer [4] of VFS (Virtual File System) or it's counterpart. Splice, on the other hand cuts across the lower layers. The objective there was to eliminate data copies and to take advantage of protocol implementation specific details. This makes it very efficient but requires an implementation for each protocol and the file subsystem on which it is to be used.

Cue is portable across more versions of Unix as the vnode layer is defined over different protocols and file-systems. As a consequence of using the standard abstraction it is extremely compact. It performs very well for bulk data transfers.

We have not tried applications such as continuous media to compare the approach with customized rewrites of the file/network subsystem. Splice has been well evaluated for continuous media applications [3] [6] Cues are general purpose mechanisms that operate within the standard Unix scheduling and file buffers framework. This makes it easier to retain the standard accounting and file buffering strategy.

In the next section, we present the model of service provided by our system call. The performance measurements are presented in section 4. Section 3 describes the environment of the experiment and in section 5 we describe the implementation of the system call. In the concluding section, we also outline future work.

2 The Model

Figures 1 and 2 outline the kernel streaming construct. In the first figure, the path taken by data in standard transfers is shown, and in the second, the basic construct is outlined.

A `cue()` call creates a data structure that defines a type of stream called a Cue. These streams are formed over the vnode layer [4] (or it's counterpart) in the file subsystem, and are private to the process. The call also returns a standard descriptor which is used to write requests to define and initiate the data transfer.

A write to this descriptor is used to send requests from the user process to this service, as shown by the dotted line in figure 2. In the current implementation, requests consist of integer codes which initiate different actions. One of these actions is to perform a data transfer. The requests used in these tests are: changing the offset in the source file, setting the size of individual read/write, and setting the length of the total transfer.

Since file descriptors define a Cue's end-points, `cue()` can be used to connect a socket and a Cue descriptor. This means another process across the network can interact with the Cue without having to go via a user process. This is useful for popular applications that use file servers, viz., ftp, www servers, and distributed databases. In particular it provides high performance and simplifies user code. The abstraction in the kernel is at a sufficiently high level to maintain the access rights desired by the local process.

At invocation, `cue()` initializes the default parameters and context for the transfer, including per-process kernel buffers to be used for moving data between the network and the file-system routines. Subsequent writes to the control descriptor either modify the parameters or initiate the transfer.

No changes are required to `cue()` for the second invocation for remote cuing of the transfer. This simplifies the code structure. Consider the the alternative to this in the standard case. It would require running another connection in parallel to the transfer, and either need to interrupt the main transfer to check for requests, or have routines equivalent of RPC for servicing the request. Both involve scheduling by the user processes. Requests as simple as file or block reads can be serviced by the kernel directly. For testing this concept, the only additional line of code required was to set the read size of service on the second invocation (to avoid blocking), using commands described above.

For transfers to proceed in the kernel, a mechanism is required to schedule the transfers. It is important to ensure that transfers for a process do not exceed it's quantum. A very simple method was found to work well. We specify the schedule as a repeating sequence of a fixed number of blocks 'k' transferred followed by a fixed delay 't' between them. The two parameters *(k,t)*, are specific to each invocation of the service. During the transfer, 'k' read/write operations are executed, and then the processor is relinquished for time 't' (in addition to the time

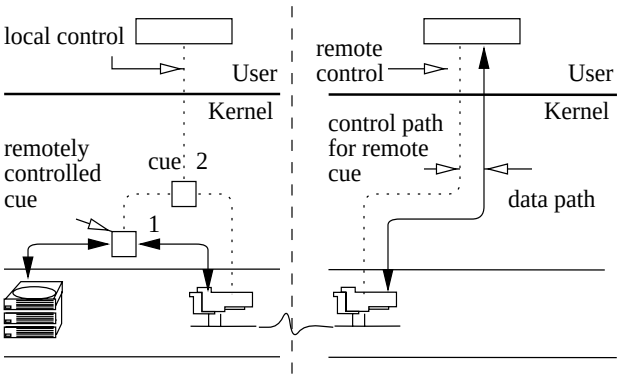

Figure 3: Remote control with two invocations.

while waiting for the disk block to be read from the disk). The transfers are executed only during the process's share of CPU time. Overshoots are currently avoided by ensuring that the value of k is set no larger than the per process time slice, i.e., 100 mS in Unix. At CPU yield time, Cue lowers the priority such that the Unix scheduler can preempt it to schedule other processes.

3 Experiment Environment

We ran extensive experiments and measured the performance obtained by the standard user driven read/write and using Cues. These measurements are for reading files from a disk on a DEC-station 3100, 12 MB RAM running Ultrix 4.2. The Ultrix kernel was modified to add the `cue()` system call. This machine is used for transmitting data read from a file on the disk to the network, on a TCP/IP socket. The data was received on a DEC 3000 Alpha running OSF1 V3.2. On the receiver, two types of setups were used. In one the data was simply read off the socket, and in the second, all data was copied to the disk. Keeping the machines asymmetric helped ensure data was received at least as fast as the 3100 can transmit.

The programs to measure the throughput perform a number of transfers of large files and report the total transfer time, system time and other system activity by directly reading the ap-

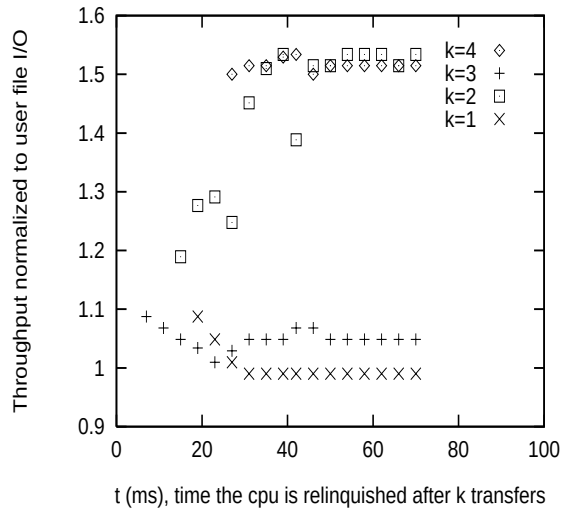

Figure 4: Performance of four parallel transfers.

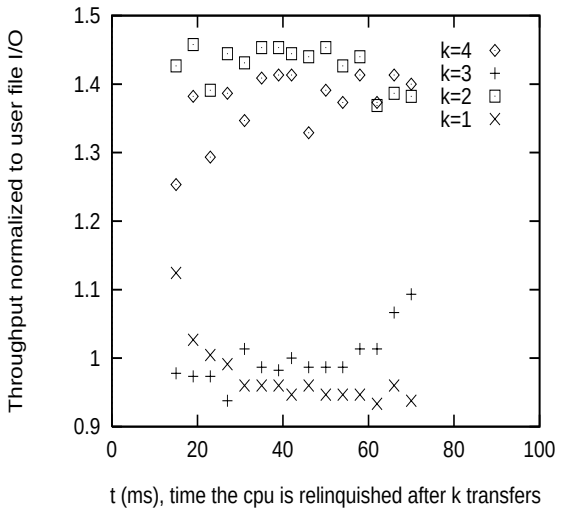

Figure 5: Three parallel transfers.

propriate kernel variables. These routines also print out several details, such as current page fault activity, signals received, etc., to ensure that any readings affected by other system activity could be rejected. The number of transfers and the size of the file were chosen to ensure that file reads were always from the disk and not from the file-system buffer cache, (by keeping the file size 12MB, approximately ten times the buffer cache size), and that there was a reasonably large transfer (Hundreds of seconds) to eliminate transient effects.

The tests for schedules reported in section 4 last up to several minutes each. Several of these schedules have been run for hour long transfers separately, during initial testing.

4 Performance of Transfers

In this section we present two types of performance measurements. First we show the performance of the system during concurrent transfers. Next, we estimate the load offered to the system using a Cue transfer.

The first two subsections describe the performance of reading data from the disk. The tests in the first subsection are with reads from the socket and in the second subsection we corroborate the results with performance of file transfers to a remote system's from a standard file server. In both cases we report measurements obtained with the standard user driven I/O and `cue()` .

4.1 Concurrent Transfer Throughput

Figures 4-5 show two performance measurements made on this implementation. As mentioned, it is possible to fix the number of blocks (k) transferred during one time slice given to the user process and the delay (t) for which the process yields. The value of d is actually specified in terms of system's `1/HZ` tick, and in this paper, it is converted to ms, as in the graphs (x-axis). The throughput reported is the aggregate of all transfers, normalized to that achieved with standard read/write on the system. A value of 1 on the y-axis denotes throughput equal to that achieved without in-kernel streaming with the same number of transfers. The actual values of throughput for three and four transfers can be read off of table 1. The results shown in these transfers (cf. section 3) are for reading a file off of the disk and sending it to a remote machine over a TCP socket. Each of the transfers is done by a different processes and it uses a different 12 MB file from the disk.

There are three observations worth noting in

these graphs. First, there is jump in the performance during schedules with 't' less than 30-40 mS. The four transfers case, fig. 4 stabilizes around 40mS, and the three transfers case stabilizes around 30mS. This suggests it is a direct consequence of being too close to the actual seek time of the disk, which has been measured close to 10mS.

Second, the case with k=3 shows a very poor performance in both these cases. In a subsequent experiment with only two transfers in parallel, (not shown here) it performs better, on an average between the k=2, and k=4 case, with t greater than 60mS. This may be due to disk seek times. A guess is that read-aheads initiated with an odd number of transfers add to the delay, whereas the case of k=2 and 4 the block read-ahead is taken from the buffer. At this point, this is at best a conjecture.

Finally, with these measurement, while it is clear that this technique maintains the systems throughput, still, with our measurements nothing concrete can be said as yet about what determines an optimal schedule.

Plausible explanations include the role of the delays introduced by the disk, including details such as placement of the actual files, and the fact that in the kernel the scheduling of delays is not fine grain or synchronized with disk transfers delays. In this case, the $\approx$ 4ms granularity of kernel schedules is very close to the disk seek time, $\approx$10ms.

4.2 File server tests

One of the most popular applications utilizing parallel transfers like the ones tested above is an ftp server. Cues were successfully tested to improve the performance of a Unix ftp server. The changes required to the ftp server were minimal. The modifications involve replacing the two lines from the data transfer loop that read and write data.

The `cue()` call connect the two file descriptors for reading and writing and a `write()` call initiates the transfer. No changes were made on the client.

No. of clients in parallel	(KB/s)		
	user driven transfer	using `cue()` (4,16)	(4,70)
1	410	370	200
2	300	360	350
3	240	330	300
4	240	360	317

Table 1: File server performance. Peak performance numbers reported for standard as well as `cue()` transfers. Please see text for the choices (k=4,t=70), and (4,16).

The numbers in this section are reported from four to five tests of the 12MB file. Since FTP is layered over a TCP socket application, we used just enough tests to reconfirm the numbers obtained in the above tests which are already rigorous. These numbers were very all close to the above averages. In table 1 we have reported the peak numbers. We do that because these numbers are taken off of the client reports, and they are not as consistent as the numbers we report in section 4. In those tests, we closely controlled the measurements, and these numbers are very close to the above tests.

It is alright to measure them this way without being very rigorous because the numbers are close, and are a reconfirmation. The difference in the readings at the client versus the server has to be negligible. This is so because the window used by tcp is orders of magnitude smaller than the individual transfer sizes tested ($\approx$ 10K versus $\approx$ 10 MB), and the client reports measurements after writing the retrieved file to the disk. The two schedules chosen are (4,16), which showed one of the best performances, and (4,70). Note that (4,70) is a conservative schedule for four transfers.

4.3 System Overhead

An important evaluation of a new read/write primitive such as `cue()` , is to compare it's performance with the standard read/write calls, not just in terms of time but also with respect to the load it offers to the system. It is possible using several techniques, including Cues to use the processor very aggressively. One intuitive measure is the costs of this transfer while delivering the baseline performance (provided by the existing primitives). On the other hand it is difficult to make an exact comparison of the load offered to the system. The transfers are scheduled differently and the comparison is especially complicated as one of the two methods involves a user level scheduling of the process. To keep the comparison meaningful, we want a schedule with an acceptable *equivalence* with the standard transfers. We describe the results id measurements obtained by using one such equivalent schedule. A description of the heuristics used for arriving at this schedule as a comparable one to standard transfers is described following these measurements. The test designed to compare the load offered to the system by the in-kernel streaming primitive starts a process that transfers a large file (nearly 12 MB) from the disk sixty times using user driven as well as in-kernel transfers. In parallel, it forks a process that calls (`gettimeofday()`) in a loop, and maintains a count of the calls. At the beginning and end of each of the sixty transfers, the first process sends a signal to the second process. The process receiving the signal prints out the current count and resets the count to zero before resuming. The system call loop was chosen to ensure most timely delivery of the signals. It is hard to otherwise meter the progress of the parallel process synchronized with the duration of the transfer in the first one. The numbers of syscalls reported by the parallel process during the durations of the in-kernel and user driven transfers, help us compare how much of the processor was available for other system activity during each of these transfers. This type of transfer, with a metering process in parallel, maintains a full utilization of the processor. The total time for this set of tests in averaged 3200 seconds for each run.

The schedule chosen for this comparison (k=4,t=70) takes almost equal time as user driven transfers (on an average 1.7% more). This amounts to around 16 seconds more on an average. We notice an average of 5% increase in the number of calls made by the metering process showing two million extra calls over the transfer. This corresponds to an approximate gain of 4%, in spite of this marginally extra time spent, reasoning as follows: The number of calls the metering process makes, were it left to run on it's own is roughly 32,000 per second. At this rate, 16 seconds of extra time gives it a chance to make less than half a million more calls. After subtracting this number from the surplus calls that we see in the in-kernel transfer, we see cpu cycles amounting to 4% more system calls available to the other process, with only a *single* transfer in progress.

4.4 Pacing of transfers

An informal but interesting made during the above tests is that Cue transfers also seem to pace themselves very well relative to each other. This is true of user read/write driven transfers even with their performance drops. Such pacing is a good sign, indicating fair use of the system's resources. This can be taken as a hint that as far as bulk data are concerned, it is possible to rely on Unix kernel scheduling with the simple mechanism of this implementation (explained in section 5). The schedule can easily be made user-customizable in the current implementation of the user interface. In the current version we do not have tests for different connections pacing at rates predicted by a simple use of schedules. If this tests out well, it could be provided to the user or to the system user to prioritize transfers. Safely providing such a flexibility to the user only

entails very simple checks for transfer requests to be no more aggressive than a preset (k, t).

5 Implementation Details

The syntax of the call for creating the service is:

```
c = cue (s, d);
```

where `s` and `d` are descriptors pointing to the source and destination file (or socket), respectively. The return value `c` is the descriptor that is used to direct the transfer. Simple integer codes are written to this descriptor using the standard unix `write()` call to start the transfer, set the offset and transfer size, or to set the size of the read/write unit. The following example summarizes the code for a typical application.

```
   int sd, fd, c, commands[CSIZE];
/* Open connections, etc.          */
   sd = socket(AF_INET,SOCK_STREAM,0);
   connect(sd, &addr, sizeof(addr));
   fd = open ("file", O_RDONLY);
   c  = cue(fd,sd);
/* At this point the call is ready */
/* for transfers.                  */
/* Commands to transfer 4000 KB    */
   commands[0] = WRITE;
   commands[1] = 4096000;
/* The following statement starts  */
/* the transfer                    */
   write (c, commands, 2*sizeof(int));
/* Dismantle the service           */
   close (c);
```

Actual user code includes the error checks for return values from the system calls, i.e., socket, connect, open, cue, write and close. The code for initiating read and write sits directly on top of the vnode layer of VFS.

On invocation with appropriate descriptors for the file and the socket, the call instantiates a Cue to maintain context for this transfer and creates an entry in the descriptor table for the control descriptor. It also allocates a file table entry. The `fileops` structure points to the routines for handling Cue operations instead of the inode/socket fileops structure. The control descriptor is returned to the user. On a subsequent write call to this control descriptor, the routines that schedule the transfer are invoked The write request on the control descriptor may ask for as large a transfer as it needs. The call transfers a preset number of blocks and then goes to sleep for the preset interval before resuming. The priority is raised during this time such that other processes ready to run may get the processor.

The code for the basic implementation, including the code for interpreting writes to the control descriptor and scheduling transfers is around 400 lines of C code. This does not require any modification to the file subsystem below the the VFS wrapper. Most of the implementation is free of proprietary system details and can be made available with appropriate substitutions of the proprietary code.

The return value from the system call is a descriptor just like any other descriptor. It is simple to connect this descriptor to a remote socket using the call itself. This allows sending commands from a remote system to control the file transfer. The transfers operate with descriptors over the vnode layer and are executed only in the context of the process. As a consequence of this abstraction, the flags to check the read/write permission on the file are available to the Cue operations.

6 Conclusion

The implementation and tests show use of this concept for improving the performance as well as for a simplifying the interface for an important application. The implementation uses the standard abstractions. This makes it simple and portable and it delivers good performance. The model also allows a modular composition of this service for simplifying distributed control of transfers.

The schedule values chosen right now (k,t), are heuristic or at best empirically derived magic numbers. Determining the optimal schedule, especially on the fly would be on the immediate to-do wish list.

Future extensions to this work may focus on

use of additional buffering, particularly if we want to evaluate it for some time sensitive transfers. Pacing transfers at different rates would be a very useful. We are also looking at extending the Cue ADT to do more than single input single output streams.

References

[1] Peter Druschel, Mark B. Abbott, Michael Pagels, and Larry L. Peterson. Network Subsystem Design: A Case for an Integrated Data Path. *IEEE Network*, July 1993.

[2] Kevin Fall and Joseph Pasquale. Exploiting In-kernel Data Paths to Improve I/O Throughput and CPU Availability. In *Winter Conference*, pages 327–333. USENIX, January 1993.

[3] Kevin Fall and Joseph Pasquale. Improving Continuous-media Playback Performance with In-kernel Data Paths. In *Multimedia Conference*. IEEE, March 1995.

[4] S. R. Kleiman. Vnodes: An Architecture for Multiple File System Types in Sun Unix. In *Summer Conference*. USENIX, June 1986.

[5] Leffler, et al. *The Design and Implementation of the 4.3 BSD UNIX Operating System.* Addison-Wesley Publishing Company, 1989.

[6] Joseph Pasquale. I/O System Design for Intensive Multimedia I/O. In *Workshop on Workstation Operating Systems*, Key Biscayne, FL, 1992. IEEE.

[7] Dennis M. Ritchie. A Stream Input-Output System. *AT&T Bell Laboratories Technical Journal*, 63(8):1897–1910, 1984.