A case for in-kernel data streaming over the file subsystem *

Sandeep Gupta
Department of Computer Science †
John S. Baras ‡
Institute of Systems Research
Stephen Kelley †
Institute of Advanced Computer Studies
Nick Roussopoulos †
Institute of Advanced Computer Studies and Department of Computer Science

University of Maryland, College Park, MD 20742
{*sandeep,nick,skelley*}*@cs.umd.edu, baras@isr.umd.edu*

May 30, 1996

Abstract

This paper maeks a case for a new implementation of in-kernel data streaming in Unix. In particular, we show we can use it to improve the throughput during an important activity: file transfer. The performance improvements are not limited to the aggregate throughput of the system, and we see that it makes lesser demands on the system. This note describes the performance measurements of our implementation of in-kernel data streaming in Unix for large file transfers, and presents a comparison with standard user driven transfers.

1 Introduction

We make a case for a new implementation of in-kernel data streaming in Unix. In particular, we show we can use it to improve the throughput during an important activity: file transfer. This can be done by inexpensively scheduling the transfers in the kernel within a process's kernel share, without returning to the user process. We implemented and tested data streams over the file subsystem. Our interface turns out to be simple, compact and flexible to implement in-kernel data streams as well as has promise for a more general system service. The performance improvements that we report here are not limited to the aggregate throughput of the system, and we see that it makes lesser demands on the system.

*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 internt at AT&T Bell Laboratories, Murray Hill in Summer '94.

†Also with the Institute of Systems Research.

‡Also with the Department of Electrical Engineering.

The throughput gain for multiple transfers has been found to be upto 50% to 80% for four transfers in parallel. This gain has also been verified for a standard ftp server (wuftpd). In addition to raw throughput performance, since we eliminate copies in and out of the kernel and avoid involuntary context switches, we also expected savings in processor cycles. This was found to be true too. Consequently this technique should be very useful for popular applications that use file servers viz., ftp, www servers, and distributed databases.

A general discussion on in-kernel streaming appears in [1]. An earlier implementation [2] shows an alternative way of splicing for Unix. Data streaming through the kernel can be done in several ways. System V streams [6] modules can be used for the same concept. Our implementation uses the file subsystem abstraction [4], and in contrast to these differs in the applicability and at the user interface. Our implementation has potential for a more general OS service and is extremely compact. On the other hand, in it's current version we consider it early to evaluate it for video data where [2] is well tested [3], [5].

Most of this note describes the performance measurements of our implementation of in-kernel data streaming in Unix for large file transfers. Before we could compare the this technique to standard user driven transfers, we needed some meaningful comparison. In the next section we describe the measurement environment. Section 3.1 details the selection of our experiments, measurements and key observations that make a case for this technique. We give an outline of the implementation in section 4, and following the conclusion, a view of the future work.

2 The measurement environment

These measurements are on a DEC-station 3100, running Ultrix 4.2 with the additional system call (Cue) implemented for in-kernel streaming. This machine is used for transmitting data from the disk to the network, and the data is being received on a DEC 3000 Alpha running OSF1 V3.2. Two types of setups were used, in one the data was simply read off the socket, and in the second, an ftp client copied data to the disk. The machines being asymmetric ensures that the Alpha is capable of receiving data at least as fast as the 3100 can transmit.

Our programs to measure the performance (other than the ftp server, mentioned next) do a number of transfers and report the total transfer time, system time and other system activity by directly reading the appropriate kernel variables. We print out several details to ensure the readings are not affected by other system activity.

Most of the initial sets reported are several sets of 60 transfers each, and the later run on performance of several transfers in parallel are two to three sets of 3 transfers. The numbers measured from these parallel transfers were verified to correspond to performance on a ftp server from Washington University (wuftpd-2.4) that we modified to use Cue. The modifications on the ftp server involved replacing the two lines from the data transfer loop that read and write data.

2.1 Details of measurement routines

In each individual run we vary the scheduling of transfers, by having the Cue relinquish the processor for some duration after a number of transfers. By varying the number of transfers 'k', and the duration 't' for which it relinquishes, we can study the performance and design schedules. Three kind of measurements are reported in this section.

The first type starts a process that transfers a large file (nearly 12 MB) from the disk sixty times using user driven as well as in-kernel data transfers. In parallel it forks a process that makes a system call (`gettimeofday()`) in a loop, and maintains a count of how many calls it makes. At the beginning and end of each of the sixty transfers, the first process sends a signal to the second process. The second process, on receipt of 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 signal. It is harder to otherwise meter the progress of the parallel process synchronized with the duration of the transfer in the first one. By the numbers of syscalls reported by the parallel process during the durations of the Cue and user driven transfers, we can compare how much of the processor was available for other system activity during each of these transfers.

The second type of measurements do not fork the parallel process. They simply transfer the files using the two mechanisms and report the measurements across the transfers. The third type of measurements use ftp. The observations are taken with a modified ftp server on the DEC 3100 as mentioned above, and one or more standard ftp clients running on the DEC 3000 Alpha. The measurements of throughput are those reported at the client.

The first type of transfers with metering process in parallel, maintain a full utilization of the processor. For all three types of measurements, the number of 8 KB blocks read from the disk for the transfer are approximately 90,000. Over the course of the sixty runs of the 12 MB transfers each, the user driven transfer consistently uses 25 more block inputs. This is not completely understood, but since the number is small, we ignored the difference, while we account for it in the calculations. We also monitored other activity on the system and there is little page reclaim and page fault activity during these runs - only a couple in the hour long runs. There was no other difference in the two transfers. Most of the initial transfers run over an hour each. Some of the later experiments with multiple streams running in parallel, have shorter runs.

3 Schedule selection and evaluation

First we describe the heuristics for equivalent schedule computation for in-kernel streaming of large file transfers. An discussion on the measurements with the selected schedules follows. An appropriate scheduling for the transfers was needed since the first attempts at in-kernel streaming using Cue did not show any gain in performance. In this implementation, a schedule refers to two parameters *(k, t)*: *k* is the number of blocks it fetches from the disk and sends out without pause every time the Cue call goes active, and *t* is the time it pauses between two such sets of transfers.

3.1 Schedule selection heuristics

To decide on the initial values of *(k, t)* , we used the following heuristic: We measured the average number of user driven block transfers per 100ms (the time slice on the Unix systems) and yielded the processor for the *right* amount the time. The notion of *right* was in terms of the usage of the kernel for the transfer, both in terms of share, and the duration of individual requests. For the first example, there are two dominant processes: the transfer process and the metering process. We do not want either process to hold the kernel for a time greater than the time quantum that Unix tries to limit for each process. Second, to begin with we do not want to keep either process active for an unfair share, though in the later experiments we evaluate the results of slightly aggressive scheduling of the transfers.

We started with a simple heuristic of scheduling as many transfers as possible without violating the time slice the OS provides, and then yielding the processor for the amount of time equal to what it used. An approximation of disk seek time/block transfer time was used to guess the time to yield the processor.

We approximate fair behavior in the first schedule by starting with three transfers (i.e., k=3), not wanting to routinely overshoot the time quantum as the average blocks read during user driven transfers was less than four. Cue transfers could manage four transfers within the same quantum so we experimented with k=4 as well. The delay (i.e., t) chosen for the first test is 70ms, was approximated with the average seek time of the disk being considered as the dominant factor.

Comments on the heuristic In retrospect, our starting choices were close to the fraction of the system time used during the transfer. User driven transfers maintain transfer rates of around three block transfers per 100 ms unix quantum on an average. System time usage was approximately 50%. If we used the system time rationale we would be yielding the CPU after using the same ratio of system time as for the user driven transfers used, i.e., around 50ms after three transfers. It turns out that the exact choice of the delay is somewhere close, and for multiple transfers, it does not matter, as the disk transfer times dominate the actual delay.

As mentioned above, the value of d was chosen based on disk seek times and they being 15-20 ms. This was not looked at rigorously as the initial tests started to lead us to interesting answers. Also, at this point we wanted to establish if this concept had any utility. In section 5 we mention how this scheduling is also dependent on the disk's access time, and the need to remove this dependency. One experiment also showed us that the reasoning behind the heuristic doesn't always work, and the merit in the concept makes a case for study of the schedules. Once we got interesting answers we looked at other schedules by dividing the delay across blocks, and choosing schedules that differ in aggressiveness from the first one.

3.2 System performance with the first schedule and variations

The most important observation from set 1 is that it does not seem to cost too much in terms of time to do even 50% more context switches if they are voluntary. This fact can be used to expect that the cost of in-kernel scheduling for large file transfers will be justifiable, if there are performance benefits. Second observation is that the variance in the number of context switches is much more with user driven transfers. Finally, this schedule shows it is *possible* to get more work done by a parallel process (the metering process in this case) by a less aggressive transfer. We emphasize 'possible' because later experiments also showed that all schedules that appear less aggressive by the above heuristic are not necessarily less expensive.

The important thing to note is that it is possible to pace the transfers entirely using in-kernel voluntary context switches within the time quantum of the process, at a lower cost. The answer on gain in overall performance is not conclusive in *this* run as the user driven transfers provide better throughput than Cue. Even so, with this schedule taking 15% more time, it is a good starting point to guess test schedules based on the observations on system behavior.

The average *system* time spent servicing each block in the Cue transfer is 12% less, and the number of system calls made by the metering process are 18.5% more than in the user driven transfer. Note that the time taken by Cue transfer in this case is also 16% more. There is indication of improvement in system performance, interpreting it as follows. The number of calls the metering process makes, were it left to run on it's own is roughly 32,000. Then, the Cue transfer and the

k = 3 (8192 KB block transfers), d= 70ms			
S. No.	Context Switches per MB transferred		
	User driven transfer		Cue transfer
	voluntary	involuntary	(all voluntary)
1.	6	23	48
2.	6	23	47
3.	5	23	47
4.	19	18	50
5.	8	22	49

Table 1: Set 1: The number of context switches during user driven transfer, though three times less, are involuntary over 70% of the time. This shows the possibility of inexpensive scheduling of transfers in the kernel.

S.No.	User Driven			Cue (k=3, d=70ms)		
	Time (S) per MB		System calls per	Time (S) per MB		System calls per
	wall clock	system	second in parallel	wall clock	system	second in parallel
1.	4.6	2.3	12574	5.3	2.1	14740
2.	4.6	2.3	12605	5.2	2.1	14653
3.	4.6	2.3	12619	5.2	2.1	14640
4.	4.5	2.2	12420	5.4	2.1	14913
5.	4.5	2.3	12564	5.3	2.1	14928

Table 2: System performance during user driven and Cue part in Set 1. This hints towards the utility of the call with schedules that perform slower than user driven transfer. Please see text for interpretation.

user driven transfers take off 47% and 60% respectively from the standalone metering rate. The next experiment uses a schedule which does not use as many context switches but gives us firm evidence of increased CPU availability during these transfers.

3.2.1 Set 2: Evidence of improved cpu availability with comparable transfers.

In the second set we have a schedule which takes roughly equal time as the user driven transfer (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 amounting to about two million extra calls over the transfer. This amounts to an approximate gain of 4%, in spite of this marginally extra time spent, as shown next.

As noted in the previous section, the metering process, left to itself without the transfers in parallel makes less than 32,000 calls a second. At this rate, 16 seconds of extra time only gives it a chance to make less than half a million more calls. Even if we subtract this number of calls from the surplus calls that we see in the Cue transfer, we see cpu cycles for 4% more system calls available to the other process, for a *single* transfer.

k = 4 (8192 KB block transfers), t= 70ms				
S. No.	Total time for Cue part		System calls during transfer	
	(seconds)	% over user driven part	per block	% over user driven
1.	3183	1.7	463	4.8
2.	3246	3.4	478	7.9
3.	3170	0.5	462	3.6
4.	3194	1.3	468	4.2
Average	3198	1.7	468	5.1

Table 3: Set 2: Performance with a schedule that takes about as much time the user driven transfers. The 1.7% extra time amounts to only a fourth of the 5.1% gain seen by the metering process, for just one transfer in parallel. Please see text.

Setup	Time[1] (S)		System call rate in parallel
	Wall clock	System	
User Driven[2]	3147	1562	12471
Set 3:Cue (k=1, t= 16ms)	2678 (85% of std)	1214	9890 (80% of std)
Set 4:Cue (k=1, t= 20ms)	3022 (96% of std)	1213	10422 (82% of std)

Table 4: Sets 3,4: Effect of distributing the delay after each transfer. Adding extra yield time doesn't help linearly. [1] The times reported are from aggregated times of 60 transfers. [2] The user driven transfer part numbers are averaged over eleven such measurements from Sets 1-4.

3.2.2 Sets 3,4:Distributing 't' over individual transfers

Next we measured the effect of distributing the delay t after each transfer instead of after every three or four transfers, with the same heuristic. It is not possible to get exact division of delay because of the scheduling quantum in the kernel, so we use two sets. The CPU yield time per transfer averages to around ten percent less in set 3 and ten percent more in set 4.

Both these cases result in a nearly identical saving of the time the process spends in the system mode (22%). By itself, this is not as conclusive a measure of performance, but for the fact that nearly this amount is seen saved from the overall time as well, in set 3. In set 4, even though the process yields for 10% extra time, it does not seem to help the process in parallel, which tells us that by yielding the processor for an extra time, it doesn't always help. It is also important to note that in all the experiments, the transfer size (8K) is the optimal size for this machine. Increasing the buffer size for user driven transfers doesn't help the throughput significantly.

The more useful of these two schedules in this case is greedier than the user driven. Since the quantum for timing is large, (4ms), it is not possible to get exactly comparable schedules. Still, if we have not yet found equivalent schedules, i.e., an exact criteria for comparing or finding the best schedule it is not entirely discouraging. This schedule, e.g., is indirectly an evidence of more capacity, as it gives a totally free processor for 15% of the time than the user driven transfer case. Using the number of metering calls standalone (32,000 a second), this is equivalent to about 37% more of system calls that would be made by the metering process.

Parameter	User Driven transfer Sets 1-4		Cue transfer					
			Set 1		Set 2		Set 3	Set 4
			k=3, t=70		k=4, t=70		k=1	k=1
	Mean	σ	Mean	σ	Mean	σ	t=16	t=20
Wall Clock time/MB	4.53	0.04	5.27	0.05	4.60	0.05	3.85	4.34
System time/MB	2.25	0.06	2.13	0.02	2.17	0.01	1.75	1.75
Context switches/MB	30[1]	2.4	48	1.4	39	1.2	132	131
Syscalls in parallel (s^{-1})	12471	390.8	14775	138.6	13017	74.7	9890	10422

Table 5: Summary observations from Sets 1-4 show there is less strain on the machine for Cue transfers. [1]73.2% of context switches during user driven transfer tests are involuntary in this set up. Some later experiments show that large number of involuntary context switches show up especially for one process doing the transfers (32%). The rest are due to the metering process.

Setup on both transfers	Time(S)/MB Wall clock	Time(S)/MB System	Context switches/MB	System call rate in parallel (s^{-1})
User driven	4.05	1.00	59.6	10463
Set5:Cue (k=3, t=16ms)	4.46	0.64	157.0	11571
Set5:Cue (k=4, t=16ms)	3.17	0.89	97.2	6032
Set6:Cue (k=4, t=16ms)	3.13	0.88	96.8	No metering

Table 6: Sets 5,6: Two Cue transfers in parallel show better machine utilization scaling, as well as an improvement in system's aggregate data throughput.

3.2.3 Other observations from Sets 1-4:demands from the system during transfer

In table 5 we summarize the observations from sets 1-4. In additions to performance averages, we also computed the variance. The number for the total transfer time are roughly the same, but in terms of the demand it makes from the system time, the context switches it puts the system through, and the amount of CPU available to the other process, Cue transfer seems to behave more consistently than user process driven transfers. The number of observations for Cue transfers in sets 1-4 are 4-5 each, and for the user driven transfers, the number is around 10. With the caveat that this is a small number of observations, we take it as a hint to the possibility that user level transfers strain the system's performance more than Cue.

3.3 Sets 5-9: Utilization and throughput scaling with parallel transfers

Cue transfers seem to scale well, both in performance and in use of the system's resources. Sets 5,6* show the performance of two schedules with two transfers in parallel. In Set 1, we had the Cue transfer and the metering process run in parallel, and we relinquished the processor for as much time as Cue was occupying it. With two transfers and two metering processes, there are four processes in the experiment. These measurements were without our earlier heuristic, and intuitively we were relying on the Unix scheduler to do a good job even though individual Cue contend more aggressively. It does do well.

*There is no set 7 in these measurements.

No. of transfers in parallel	Time(S)/MB	
	Cue	User driven
2 (set 6)	3.13	7.40
3 (set 8)	3.42	13.32
4 (set 9)	3.23	18.97

Table 7: Sets 6,8,9: Parallel Cue (k=4, t=16ms) transfers scale throughput much better than user driven transfers. The value of t has little impact, as the kernel cannot return control to another transfer before completing k blocks, which take more than 16ms. The next subsection reports this.

In this set, we have reduced the time for which the processor is relinquished by each of the Cue. The idea was to see if there were schedules in the same neighborhood that yielded comparable performance. Set 5 performs 10% slower than user driven transfers, yielding this time to the metering process, which makes 10% more progress. Following the observation that the average system time per MB transferred was only 64% of the user driven case, Set 6 schedules the transfers a bit more aggressively, while still keeping the system time per MB transferred 10% lower. The transfers turn out 20% faster than the user driven case, however, it taxes the metering process 42%. This still is very attractive, as the total time spent on the transfer is much less. It also turned out, as observed in the next set with ftp, that the 15% slower schedule from the first set (k=4, t=70ms) shows a good scaling of throughput.

Finally, a new observation was made in Set 6 by dropping the metering process. It seems to make very little difference to the system time and context switches incurred by the Cue transfer. We take it, that in addition to creating less contention with other Cue transfers, this may be a hint that a Cue transfer also spends less CPU resources contending with non I/O intensive processes.

Sets 8,9 show the dramatic difference in performance of the transfers when two or more of them (the same type only) are scheduled in parallel. This is not the only interesting observation in these experiments. The number of context switches begins competing with Cue, and the number of involuntary context switches drops. There are several aspects to comparing the number of context switches in user driven transfers to Cue transfers, so we are not attempting to describe them in this report. These numbers are important. In the next subsection we show several Cue schedules which show comparable scaling of throughput with a standard application.

3.3.1 Wuftpd FTP server tests

The numbers in this section are reported from four to five tests of the 12MB file. Eventually, these need to be automated and made as rigorous as the earlier sets. Because we use smaller samples, and the variability in the results is large, we have not computed averages. Instead we have listed the range of throughput reported by the FTP clients. We have not looked into the ftp client sources about how they measure the throughput, hence we are not certain about the variability in the throughput across different experiments.

The measurements in the first two types of sets were at the server side, in contrast to this set. Since we are measuring the throughput only in this case, then the reading at the client versus the server will be negligible, as the window used by tcp is orders of magnitude smaller than the individual transfer sizes ($\approx$ 10K versus $\approx$ 10 MB).

The gains in performance seen by the multiple transfers seem to scale well with appropriate

No. of ftp clients in parallel	wuftpd performance at clients (Kbytes/S)				
	user driven transfer	using Cue			
		k=1, t=16ms	k=3, t=16ms	k=4, t=16ms	k=4, t=70ms
1	400-410	260-280	340-360	350-370	200
2	220-300	220-240	240-280	320-360	350
3	202-240	239-257	221-250	277-330	300
4	205-240	216-237	204-225	300-360	317

Table 8: FTP daemon performance.

schedules. In table 8, the performance with use of Cue in a user driven FTP daemon. The most conservative schedule tested above scaled 50% better with four transfers, though they start off with a much slower performance. These observations seem to suggest that the optimal (k,t) for multiple transfers may not be the same. Even the aggressive schedule e.g., (k=4, t=16) from Set 3, shows a dip with three transfers even while achieving upto 80% improvement in performance clearly indicating there is more to be understood here.

3.3.2 Pacing of the transfers

Another interesting observation made informally during the ftpd tests and the multiple transfer tests reported above, is that Cue transfers also seem to pace themselves very well relative to other transfers. In ongoing experiments as well, this seems to be true, indicating that this simple scheduling mechanism is good.

4 The `Cue()` Implementation

In this section we describe the semantics and implementation of the Cue call for file transfers to a remote machine through the network, and the change made to wuftpd. The filesystem splice concept is more general, and the same primitive can be useful for upgrading another system function (descriptor passing) within the system as well. That part is not implemented yet, and not described here. The inputs to the current implementation of the call are file descriptors for the disk file to be transmitted, and the socket on which it is transmitted.

4.1 Concept

The inputs to the call are used to locate the appropriate file table entries from the kernel data structures, and the kernel routines for data transfer area used to drive the transfer loop over the file subsystem. The first implementation transferred data in a kernel loop, and returned when the transfer was over. It would yield the processor only when it needed to wait for disk blocks. This did not yield the desired performance even at the expense of fairness in scheduling. This is because it naively holds the cpu at a high priority all through the transfer. Subsequently we tested simple cpu yielding schedules, relinquishing the priority to the minimum the kernel would admit. and measured the performance. This shows improvements in performance, and with parallel transfers, upto 80% additional throughput.

4.1.1 Syntax and semantics

The syntax of the first version of the call is `Cue (int sdes, int ddes);`, where sdes and ddes are file descriptors to a disk file and a socket, respectively. At this point, the call does not return a value, and can only do transfer in this direction. † The current implementation of the system call blocks the user process until the transfer is complete. Multiple processes can make the call and their transfers are executed concurrently. The blocking and sequential transfer semantics are going to change very soon, to facilitate random access experiments, and the syntax will remain backward compatible with this call.

4.1.2 Changes to wuftpd

The change to wuftpd is minimal. For these tests, we have simply replaced the couple of lines used for data transfer using user driven I/O with our calls. Specifically, the `read()` and `write()` call loop is replaced by a `Cue()` call with the appropriate descriptors, referring to the file being transferred and the socket to the connection for sending data.

5 Conclusion and future work

We have described the set of experiments that establish the feasibility and usefulness of implementing in-kernel data streams over the file subsystem, and in-kernel scheduling of such data streams for high volume file I/O. We measured the performance of the system with our implementation during such data transfers and compared them with user driven transfers. Finally, we claim the utility of this concept using ftp, one of the most common data transfer applications.

The most important observation from these measurements is that in-kernel streams using these primitives scale much better than user driven transfers when we try multiple transfers at the same time. We also found that our scheduling method paces multiple transfers evenly, i.e., transfers get a fair share of the throughput.

Immediate next goad on Cue is to scan for optimal schedules, and test it's suitability for databases. For further work, we want to understand the scheduling, and work on how optimal schedules could be arranged dynamically for applications such as database servers, ftp and web servers. Some of these schedules do not scale as well as the others, even though they yield better performance in some cases. Understanding the mechanics of scheduling may help us select the best or design adaptive schedules. Once a good basis for the scheduling is established, we can try simple enhancements like prefetching. Finally, the Cue call design has potential to be a very general interface to I/O services and IPC related functions such as transfer configuration, descriptor passing across processes. Some of these features will become apparent in the tests for database servers.

References

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

†It is missing the code to close the file after the transfer is complete, for the receive transfers to be correctly reflected in the filesystem. This is *not* a technical problem.

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

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

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