Performance of the 1-1 Data Pump

Tobias Mayr (mayr@cs.cornell.edu),
Jim Gray (gray@microsoft.com)

10/24/2000

Abstract: This document describes the implementation and performance of a 1-1 data pump, i.e., a program transferring data between disks on one node or on two different nodes connected by a network. Section 1 outlines the design, Section 2 describes the experimental setup, and Section 3 discusses the performance measurements.

1 Design of the Algorithm

The data pump moves data from a *source* to a *sink*[1]. The source and the sink, called the *endpoints* of the pump, can each be a file, a network connection, or a null terminator. The transfer from a disk on a first site to a disk on a second site happens through two data pumps: the *sender* pump on the first site and the *receiver* pump on the second site. The sender pump moves data from the file source to a network sink, which is connected to a network source on the target site. The receiver pump moves data from this network source to a local file sink. Null sources and sinks simulate the behavior of an actual endpoint without incurring significant costs. They are used to isolate the resource usage of network and file endpoints in the experiments.

The next section describes the algorithm that moves data between a source and a sink. It makes no difference to the algorithm if the involved sources and sinks are files, network connections, or null terminators, since the same interfaces are used in all cases. Section 1.3 will examine a few differences between files and network connections.

1.1 The Copy Loop

To allow pipeline parallelism (and hence maximum throughput), the source and sink should be active concurrently. They operate in parallel by making asynchronous IO requests that do not block the caller to wait for request completions. Several requests are pipelined on source and sink, to allow immediate processing of the next request after the previous one is completed. The number of posted requests is called the *request depth*.

The main loop of the algorithm looks like this (we omitted error handling):

```
While ( !Source->IsEndOfFile() || 0 < Source->NumberOfPendingIOs() )
{       // post read requests up to the maximal request depth
        while( m_Source->NumberOfPendingIOs() < MaxSourceRequestDepth )
               m_Source->IoStartRead();
        // wait for oldest source request to complete
        Buffer = Source->WaitForCompletion();
        // if necessary, wait for a sink request to complete
        if(Sink->NumberOfPendingIOs() == MaxSinkRequestDepth)
               Sink->WaitForCompletion();
        // write the newly read buffer to the sink
        Sink->IoStartWrite(SourceBuffer);
}       // in the end, wait for the sink to complete its work
while ( 0 < Sink-> NumberOfPendingIOs() )
        Sink->WaitForCompletion();
```

As long as the source has not reached the end of its data, the algorithm asynchronously posts as many read requests as possible. The algorithm then waits for the first read request to complete and writes the result to the sink. If necessary, it waits for an older sink request to complete before posting the write (the stream must be processed in order). The final *while* loop simply waits for all write requests to the sink to finish. The only time this algorithm blocks is during calls to WaitForCompletion on either the source, to get data for the sink, or on the sink, to post new write requests.

[1] The program is based on earlier versions by John Vert, Joe Barrera, and Josh Coates.

1.2 Parameters

Request size and depth are the two main parameters influencing the execution speed.

1.2.1 Request Size

The *request size* is the size of buffers used for source and sink IO requests. It determines the granularity of data transfer. Request size affects three factors:

?? *Memory usage*: Larger requests consume more memory during the transfer. A buffer cannot be reused until its request completes.

?? *Overhead*: Each data transfer has a fixed cost independent of the amount of data. Larger buffers have less fixed costs per byte moved.

?? *Latency*: Larger requests increase the time the sink will be idle during the first read request and also the time the source will be idle during the last write request. This becomes relevant when the request size is a large fraction of the overall data.

The performance impact of request size is examined in the experiments. Based on earlier studies of Windows disk IO behavior [1,2], we expect 64KB to be an acceptable disk request size.

1.2.2 Request Depth

The *request depth* determines the number of pending parallel requests. The request depth affects two factors:

?? *Concurrency:* In some cases the latency of an IO request delays execution beyond the time needed due to bandwidth limitations, and it makes sense to hide this latency by executing multiple requests concurrently.

?? *Memory usage*: Each asynchronous request consumes a buffer until the request is completed. The number of buffers times the buffer sizes dominates the data pump memory usage.

?? *Flexibility*: Multiple outstanding requests allow continuous processing even if requests complete at varying rates, e.g., in bursts. Also, more requests allow the source or the sink more liberty in executing them (e.g., scatter/gather IO).

In our experiments, just a few parallel asynchronous requests are sufficient for 64KB buffers because the sources and sinks have relatively short latency between request and completion.

1.3 Other Issues

The algorithm's presentation in Section 1.1 omitted some interesting issues for the sake of clarity. This section presents some of them.

1.3.1 Incomplete Returns

The data pump algorithm presented above only deals with full blocks (except for the final one). An asynchronous read request to a network connection does not always return all the requested bytes (nor does the read at the end of a file). The read returns as soon as some number of bytes is available. This makes it necessary to copy the partially filled source buffers and incrementally fill an output buffer. To provide a simple source interface, we encapsulated this mechanism as part of the source. As an alternative, the algorithm could write a buffer to the sink as soon as it is returned from the source, even if only partially full. This would avoid an extra copy and eliminate the delay of waiting for a buffer to fill up. The disadvantage of this choice is that the granularity with which the source returns data determines the granularity of requests for the sink. Another, more decisive argument for our choice were the technical constraints on unbuffered file IO in Windows – the addresses must be sector aligned and the lengths must be multiples of sectors.

1.3.2 Completion Order

Sources and sinks differ in the way they wait for request completion. For sinks, the completion order is irrelevant – whatever buffer becomes available can be used for further requests. However, the source completion order is crucial: If the algorithm forwards data in the order in which the read requests complete it might permute their order in the stream. A source's WaitForCompletion must block until the *oldest* request completes. This implies that if more recent requests complete first, they will wait without being processed until it is their turn.

1.3.3 Shared Request Depth

Sources and sinks in the same process use a common buffer pool but they each have an individual maximum request depth. Earlier implementations used dynamic request depth limitations: Using a dynamic heuristic, the endpoint requiring more parallelism could increase its throughput by hogging buffers, limiting the parallelism of the competing endpoint. Theoretically, this sounds good, but we observed that the request depths would not 'self-optimize' but somtimes oscillate between maximal and minimal depth. We picked independent request depths for greater simplicity and better control of our experiments.

1.3.4 Blocking Mechanisms

Windows provides several mechanisms to wait for request completions. The data pump uses waiting for multiple events, where each event is signaled for the completion of an individual request. As an alternative, IO completion ports would have advantageous thread scheduling; however, the single-threaded data pump code is simpler using blocking on events. Alternatively, a single event per endpoint could have been used in combination with explicit polling for completion of each request.

1.3.5 Asynchronous Disk Writes

Asynchronous IO requests let the requesting thread perform other tasks while the asynchronous request is being processed and let multiple requests complete in parallel. Unfortunately, an asynchronous write request at the end of a file is executed synchronously in Windows (as a security feature). This ensures that initial writes and later reads of the new part of the file are serialized. One way to avoid this behavior was to preallocate a file of adequate length, which is not a very likely scenario. To avoid blocking the whole process, the file sink uses a separate thread to post disk write requests. This thread blocks on each request until it completes, while the main thread can execute in parallel. Still, for file sinks a request depth larger than one cannot be achieved because even with the extra thread the requests are serialized.

2 Experimental Setup

2.1 Platform

In all experiments the sender is a dual processor 731MHz Pentium III with 256MB memory, reading from a Quantum Atlas 10k 18WLS SCSI disk with a Adaptec AIC-7899 Ultra 160/m PCI SCSI controller. The receiver is a dual processor 746MHz Pentium III with 256MB memory, writing to a 3Ware 5400 SCSI controller.
The machines are connected through 100Mbps Ethernet using 3Com FastEthernet Controllers and a Netgear DS 108 Hub.

2.2 Experiments

2.2.1 Variables

As explained in Section 1.2, the possible independent variables in the experiments are the *request size* and the *request depth*.
We measured the following dependent variables:

?? *Elapsed time*. The *overall elapsed time T* together with the amount of data moved A allows us to determine the overall bandwidth of the data pump pipeline as A/T.

?? *Thread times*. The times that a thread was actually scheduled to execute, either in user or in kernel mode, give us a part of the incurred CPU costs.

?? *CPU usage*. For asynchronous IO, the thread times are only part of the CPU usage because the IO handling is done through deferred procedure calls and interrupts by system threads once the IO completes. The user thread only posts the IO. We measure the actual overall CPU usage using a soaker, as explained in Section 2.2.2.

?? *Partial IO completions*. Network read requests complete with partial results, introducing overheads for additional requests and the assembly of partial results into full buffers. The data pump keeps track of the number of partial results and the average amount of data returned.

2.2.2 Soaking

The thread times measured by Windows do not show much of the time a process spends doing IO. To solve this problem we used a soaker that measures the system idle time. A soaker determines the direct CPU usage and also the kernel thread CPU costs of handling asynchronous IO requests (deferred procedure calls (DPCs) and interrupts). A soaker has one low-priority thread per CPU, running a busy wait. The thread is only scheduled when no other thread is running. It 'soaks up' all CPU time that is left over by all other threads, especially the data pump's work threads. Running at a higher priority, the data pump's work threads and the kernel threads that execute their deferred procedure calls preempt the soaker threads. The actual CPU time of threads performing asynchronous IO is the elapsed time minus the time consumed by the soaker threads and the background system load. In a calibration phase before each experiment, the background system CPU load is determined as the time *not* consumed by the soaker threads while they are running without the worker threads.

While performing experiments with soakers we discovered an interesting effect: Soakers running on multi-processor machines can, in certain configurations, decrease the bandwidth of network transfers. This effect appeared to different degrees on various systems that we tested, varying from 2% to 20%. The reason for this effect appears to be the way in which DPC and interrupt handling is distributed among multiple CPUs. Soaker threads, running with the lowest priority, affect this distribution. The system rather interrupts a CPU running a thread with the lowest priority then an idle CPU[2]. Running the soaker only on a subset of the CPUs directs most DPCs and interrupts to those CPUs. Even soaking all CPUs slightly affects the DPC distribution and the achievable network bandwidth (up to 10%). Consequently, in our experiments we determined the bandwidth without using soakers, while all shown networking CPU costs are determined in separate experiments, using a soaker.

2.3 Scenarios

The data pump experiments measure the *bandwidth* and the *CPU cost* of transferring data. Costs are incurred by each pipeline component: the source disk, the sender CPU, the network, the receiver CPU, and the sink disk. Each component has a maximum bandwidth. A pipeline has the bandwidth of its *bottleneck component* – the component with the smallest bandwidth. The component bandwidths and costs are measured in isolation by using *null terminators*. A null source produces data and a null sink consumes them without incurring significant costs.

This allows experiments in the following scenarios:

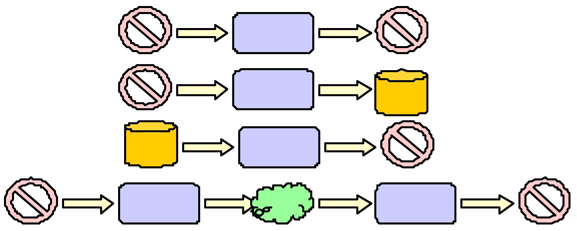

Figure 1: The four isolated experiments measure the CPU cost and the bandwidth of a file source, a file sink, a network source, a network sink, and a network transport.

?? *Isolated CPU*: Pump data from a null source to a null sink. The pipeline components are the null source, the CPU, and the null sink. The CPU bandwidth is measured for this experiment. We assume the load generated by the null terminators is insignificant.

?? *Isolated disk source*: Pump data from a disk file to a null sink. The pipeline components are the disk source, the CPU, and the null sink. The disk bandwidth and CPU cost are measured.

?? *Isolated disk sink*: Pump data from a null source into a disk sink. The disk bandwidth and CPU cost are measured.

?? *Isolated network*: A sender on one node pumps data from a null source to the network, while a receiver on another node pumps data from the network to a null sink. The source CPU time, sink CPU time and, the network bandwidth are measured.

These four scenarios measure CPU usage and bandwidth of each component.

[2] We received information that Intel designed the interrupt mechanism to consider an idle CPU as having a higher priority (IRQL 2) than an idle priority thread (IRQL 0).

3 Experimental Results

3.1 *Isolated CPU Cost*

The CPU costs for the generation of null source and sink requests and for the necessary synchronization are measured by a data pump "moving" one billion bytes from a null source to a null sink. No data are actually generated or moved in memory, but buffers are handed from source to sink the necessary number of times (10^9 / request size), all while using the event-based synchronization mechanism. Because there is no IO involved the CPU is fully utilized. For various buffer sizes, the CPU is busy for *20 microseconds per request* with a standard error of 7% for 64KB buffers when each experiment is run 10 times. The processor time is about half in user mode and half in kernel mode. Experiments with varying request sizes indicate that this per-buffer cost is nearly constant. The "throughput" for 64KB buffers is 3 GBps (no bytes are actually moved).

3.2 *Disk Source Cost*

The CPU costs and bandwidth of a disk source are measured for a data pump moving 100 million bytes from a disk source to a null sink. The disk is read sequentially and the null sink simply frees each buffer. The request depths varied from one to four and request sizes were 16KB, 32KB, 64KB, 128KB, and 256KB. For all but the 16KB buffers, a request depth of one was adequate. Consequently, all other disk source results are reported for a request depth of one. For each parameter setting the experiment was run ten times. The standard error for the elapsed times is 10% or less, that for the CPU times is 25% or less.

Figure 2 shows the disk bandwidth and CPU costs. Buffer size has no effect on bandwidth: Doubling the buffer size from 16KB to 32KB increases the overall bandwidth by 0.4% and further increase has no effect. The top right graph shows the CPU time for different request sizes in their linear dependency on the amount of data moved. The CPU cost per request, shown in the lower left, remains almost constant for buffer sizes up to 128KB. This

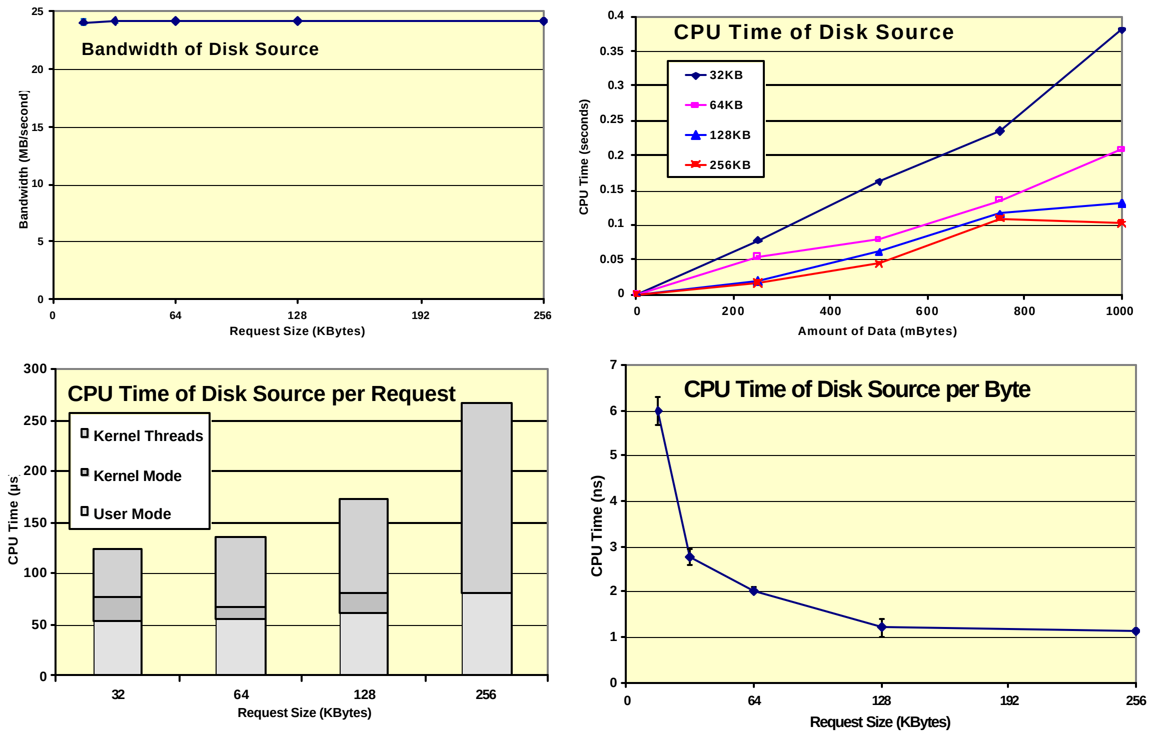

Figure 2: The costs of a disk source. Bandwidth reaches a plateau at request sizes above 16KB but CPU cost continues to decline for larger request sizes. The CPU time per byte approaches a nanosecond. As requests get larger, the user and kernel cost of an IO stays about constant, but the kernel thread cost rises as the memory copy cost of the DMA increases. Still, the per-byte cost is the smallest at large request sizes.

corresponds to our expectation that fixed CPU cost per request dominates until one gets to large (256KB) buffers. The disk source CPU cost can be approximated as a constant CPU cost per byte *Cb* and a constant CPU cost per request *Cr* (independent of the request size). The overall CPU cost, *CPU(B,RS)* would be *B*Cb + B/RS*Cr*, where *B* is the number of bytes and *RS* is the request size. The presented measurements can be approximated using *Cb = 0.5ns* and *Cr=86µs*. A more complex model would use individual per byte costs for each request size: The slope of each curve in the upper right graph is the cost per byte for its request size. Table 1 compares the actual per-byte costs observed for different request sizes and compares them to the costs derived from our simple model. Considering that the measured numbers contain the *20 µs* per-request cost of the pump mechanism itself (see Section 3.1), we can isolate the disk source costs as ***Cb = 0.5 ns*** and ***Cr = 66 µs***.

Table 1. CPU Cost of a Disk Source: Actual and as modeled by Cb= 0.5 ns and Cr = 86µs

Request Size:	Observed per-Byte Cost:	Model Prediction: *Cb + Cr/RS:*	Relative Error:
32KB	3.2 ns	3.2 ns	0 %
64KB	1.9 ns	1.9 ns	0 %
128KB	1.3 ns	1.3 ns	0 %
256KB	0.95 ns	.93 ns	2 %

3.3 Disk Sink Cost

The disk sink cost was measured with a data pump transferring 100 million bytes from a null source to a disk sink. Because writes to the end of a new file are synchronous, the disk sink data pump operator has a separate thread that posts the write requests sequentially. Hence, request depths greater than one have little effect at request sizes of 16KB or more. For each parameter setting, the experiment was repeated 20 times, with a standard error of less than 3% for the elapsed time and bandwidth. The standard errors for the CPU times were up to 100%, due to the very short CPU times involved and the rather coarse time measurements that the OS allows.

Figure 3 shows the results. The first graph shows the bandwidth as the request size increases from 16KB to 256KB: Larger request sizes increase the bandwidth, asymptotically approaching the disk write rate. Doubling from 32KB to 64KB increases the bandwidth by 8%, while doubling from 64KB to 128KB only brings a 3% increase.

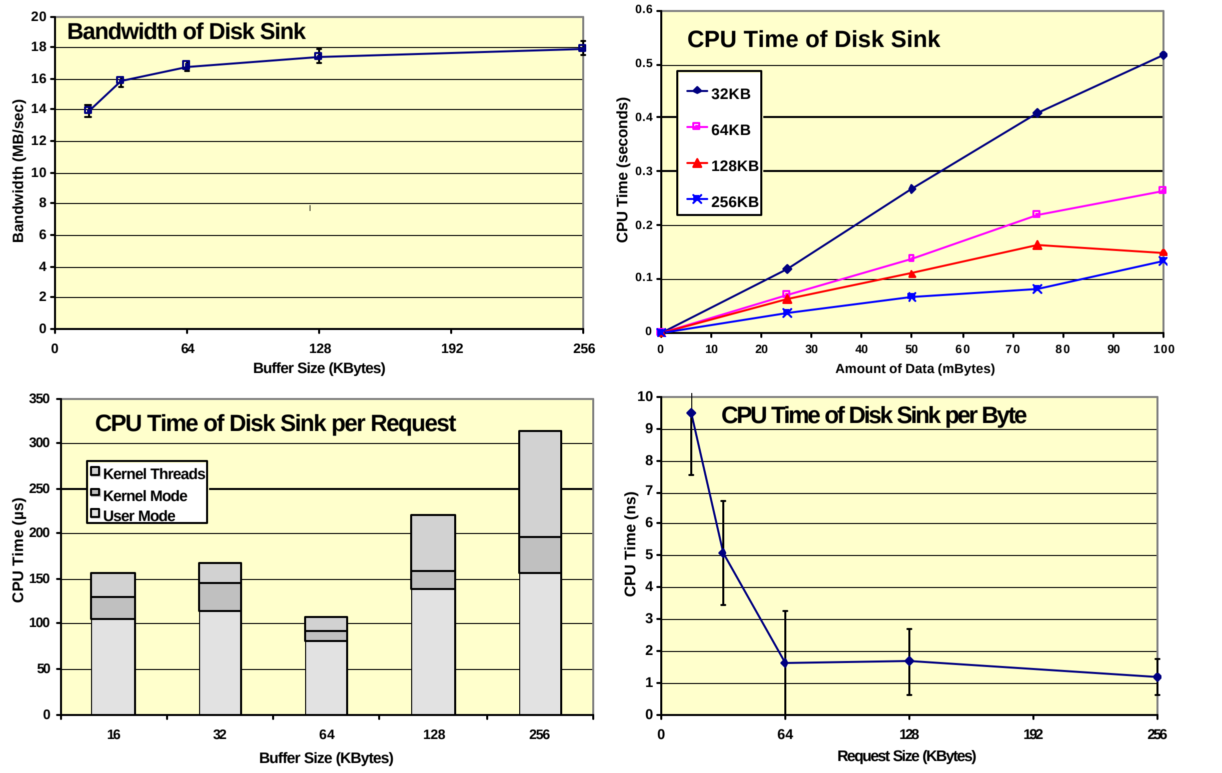

Figure 3: The bandwidth and CPU cost of a disk sink.

Table 2. CPU Cost of Disk Sink:
Actual and as modeled by Cb = 1.6ns and Cr = 73µs

Request Size:	Observed per-Byte Cost:	Model Prediction: *Cb + Cr/RS:*	*Relative Error:*
32KB	5.3 ns	3.8 ns	39 %
64KB	2.8 ns	2.7 ns	4 %
128KB	2.2 ns	2.2 ns	0 %
256KB	1.3 ns	1.9 ns	46 %

The lower left graph shows the CPU time per request. The CPU costs are approximately constant up to 128KB. This matches our expectation of a fixed per-request CPU cost between 100 and 300 microseconds. The presented measurements can be approximated using *Cb = 1.6 ns* and *Cr=73 µs*. Similar to the last section, Table 2 shows in how far we are able to match the slopes in the upper right graph. Compared to Table 1, the mo del of Table 2 approximates the four graphs only poorly. Considering that the measured numbers contain the *20 µs* per-request cost of the pump mechanism itself (see Section 3.1), we will isolate the disk source costs as ***Cb = 1.6 ns*** and ***Cr = 53 µs***.

3.4 Network Transfer Cost

The network throughput was measured by sending data from a null source via a data pump to a null sink on another node. The request depth varied from two to five and request sizes varied from 2KB to 128KB. The soaker mechanism degraded performance, so we executed the experiments twice, measuring the CPU times with the soaker and elapsed time without it, and . The experiments were run 10 times with a standard error of about 15%.
Figure 4 shows the results. The first graph shows that neither request depth nor request size has much impact on throughput – the wire speed is the limiting resource for requests large than 8KB.
The lower left graph shows the sender and receiver per-request CPU costs – the three different parts are: the time that the pump's thread spends in user mode, the time it spends in kernel mode, and finally the time used by kernel threads while processing IO interrupts and deferred procedure calls. Time spent by kernel threads was determined as the time unused by the soaker threads minus the thread times of the data pump. The CPU time per byte is nearly independent of the request size, around 20 ns for senders and 40 ns for receivers – this implies that for this

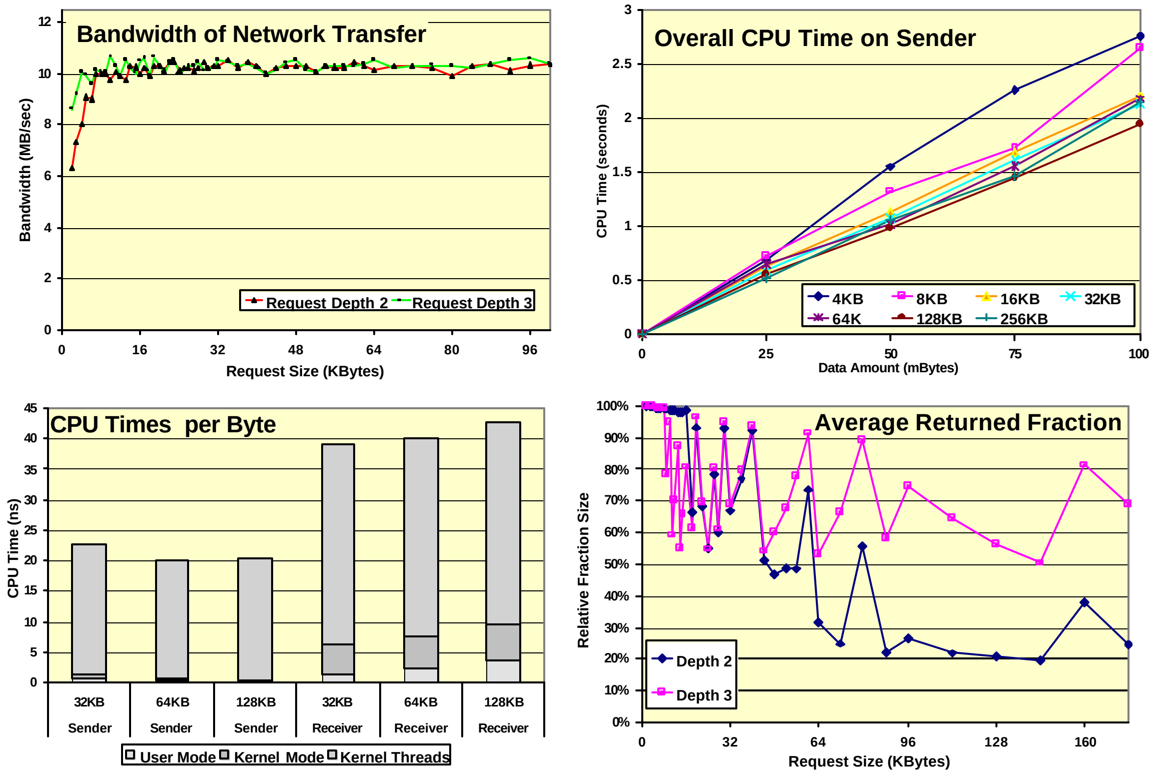

Figure 4. Network transfer costs on sender and receiver. CPU times for different request sizes are only given for the sender. The lower right graph shows the average fraction of the requested data that was returned for a read request from the network.

configuration, the CPU would be limited to a throughput of about 25MBps per CPU. The majority of the CPU time is spent running kernel threads: Asynchronous network IO involves deferred procedure calls and interrupt handling, which is not done by the requesting thread but by the kernel. The larger CPU costs on the receiver are partially due to the iteration of requests that were not fully completed and to the copying of incomplete buffers.

Request size has little effect on the CPU costs of a network transfer (Figure 4 lower left graph). This could have two explanations: a) The CPU times largely reflect the amount of data received on the network, not the number of requests, and b) The amount of actual requests does not decrease with the size of a request due to incomplete returns that have to be iterated. The graph on the lower right shows the average size of the return of a request for different request sizes and request depths. The network transports smaller units than the used buffers and imposes its granularity on the data pump.

The cost model for the sender has a low per request cost: *Cr = 40µs*, but a high cost per byte: *Cb = 20 ns.* Table 3 compares the slopes of the curves from the upper right graph – the per-byte costs for different request sizes, with our model.

For the receiver (the linear cost functions are not shown in Figure 4), we would have to reflect the fact that the per-byte cost is greater for larger requests. We could only do this by using a negative per request cost across all request sizes. In this way smaller requests, resulting in more requests, are modeled as advantageous. But even this model would only apply for the larger request sizes beyond 16KB. A more complex model would be appropriate. In our uniform model, we pick *Cb = 40ns* and *Cr = 20 µs*. The chosen request cost reflects the cost of the pump itself. Table 4 shows how these parameters help model our observations.

Table 3: CPU cost of Network Sender:
Actual and as modeled by Cb = 20 ns and Cr = 40 µs

Request Size:	Observed per Byte Cost:	Model Prediction: *CB + CR/RS:*	Relative Error:
32KB	23 ns	21 ns	6 %
64KB	20 ns	21 ns	3 %
128KB	20 ns	20 ns	0 %

Table 4: CPU cost of Network Receiver:
Actual and as modeled by Cb = 40 ns and Cr = 20 µs

Request Size:	Observed per-Byte Cost:	Model Prediction: *CB + CR/RS:*	Relative Error:
32KB	39 ns	41 ns	4 %
64KB	40 ns	40 ns	0 %
128KB	43 ns	40 ns	6 %

Considering the *20 µs* per-request cost of the pump mechanism itself, we can isolate the network sink costs (incurred on the sender) as ***Cb = 20 ns*** and ***Cr = 20 µs***. The isolated network source costs (incurred on the receiver) are: ***Cb = 40 ns*** and ***Cr = 0 µs***

3.5 Local Disk to Disk Copy

Having measured the components, we then measured the performance of the data pump transferring data from one local disk to another. Based on the experiments with isolated disk sources (Section 0) and sinks (Section 3.3), the bandwidth should be that of the bottleneck disk and the per-byte and per-request CPU costs the sum of the pipeline components. The disk bandwidth for the read disk is 24 MB per second and 22.5 MB per second for the write disk., Figure 5 shows the results of the disk to disk transfer. The bandwidth of 22.4 MB per second matches our expectations.

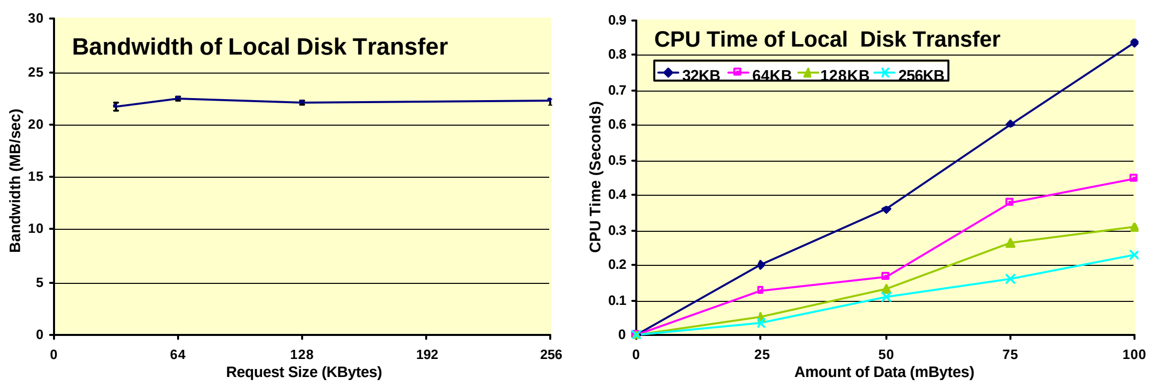

Figure 5. Disk to Disk data transfer costs. The left graph shows the bandwidth of the transfer, peaking at 22.4 MB per second for 64KB buffers. The right graph shows the linear CPU cost functions for different request sizes.

The numbers measured in Section 3.2 and 3.3, during the isolated disk source and sink experiments, should allow us to predict the per-request and per-byte CPU costs. According to our CPU cost model, which should apply uniformly across all disks, the two cost components are each the sum of the corresponding components (*Cb* and *Cr*) of the source, the pump, and the sink: *Cb = 0.5ns + 0ns + 1.6ns = 2.1ns*, *Cr = 66 μs + 20 μs + 53 μs = 139 μs*. Table 5 compares the result of this analysis with the measured overall costs per byte for each request size.

Table 5: CPU Costs of Local Disk-to-Disk Transfer: Actual and as modeled by predicted Cb = 2.1 ns and Cr = 139 ns

Request Size:	Observed per Byte Cost:	Model Prediction: *CB + CR/RS:*	Relative Error:
32KB	8.1 ns	6.4 ns	28 %
64KB	4.8 ns	4.2 ns	13 %
128KB	2.9 ns	3.2 ns	9 %
256KB	2.2 ns	2.6 ns	17 %

3.6 Network Disk to Disk Copy

This experiment combines a disk source and a network sink on one site, and a network source and a disk sink on another site. Figure 6 shows the results. Because of the already described asymmetry between sender and receiver the receiver's CPU costs are much higher. The overall bandwidth is that of the network connection because it forms the bottleneck.

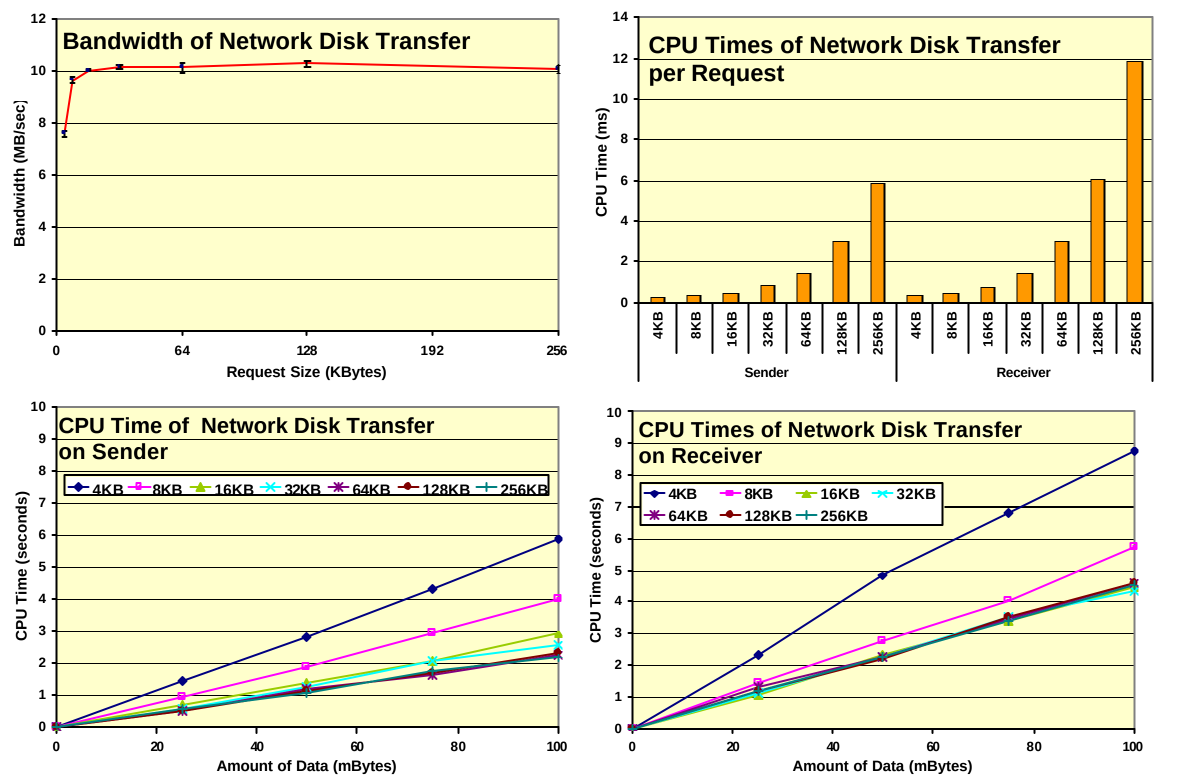

Figure 6. Disk to Disk data transfer costs across a network connection. The upper left shows the overall bandwidth. The upper right compares the CPU times per request for different request sizes on sender and receiver. The lower two graphs show the linear CPU cost graphs for sender and receiver each for different request sizes. The CPU cost on the receiver is generally higher, probably due to incomplete requests that have to be reiterated and copied.

The following tables compare the measured per-byte costs for each request size with our prediction based on the per-byte and per-request costs of the components. For the sender, *Cb = 0.5ns + 0ns + 20ns = 20.5ns* and *Cr =*

66µs + 20µs + 20µs = 106µs. For the receiver: *Cb = 40ns + 0ns + 1.6ns = 41.6ns* and *Cr = 0µs + 20µs + 53µs = 73µs.*

Table 6: CPU Costs of sender in Disk-Network-Disk Transfer: Actual and as modeled for predicted Cb = 20.5ns, Cr = 106µs

Request Size:	Sender per-Byte Cost:	Sender Model Prediction: *CB + CR/RS:*	Relative Error:
32KB	25.5 ns	23.7 ns	7 %
64KB	22.3 ns	22.1 ns	1 %
128KB	22.9 ns	21.3 ns	8 %
256KB	22.5 ns	20.9 ns	8 %

Table 7: CPU Costs of receiver in Disk-Network-Disk Transfer: Actual and as modeled for predicted Cb = 41.6 ns, Cr = 73µs

Request Size:	Receiver per-Byte Cost:	Receiver Model Prediction: *CB + CR/RS:*	Relative Error:
32KB	45.2 ns	43.8 ns	3 %
64KB	46.1 ns	42.7 ns	8 %
128KB	46.3 ns	42.2 ns	10 %
256KB	45.3 ns	41.9 ns	8 %

3.7 Summary

In this configuration, a request depth of one for disks and of two for the network is sufficient. Thus, only few buffers are tied up during the execution of the data pump.
The size of the buffer is a more difficult issue. The chosen buffer size is irrelevant for the CPU costs of network sources and sinks, due to the dominance of the network's transfer size. Disk read bandwidth favors 32KB requests, while write bandwidth increases even with larger buffers, but at less than 5% beyond 64KB. This size has much higher CPU cost than 32KB, while further increases would not add cost. Differently for writes, the CPU cost nearly doubles from 64KB to 128KB.
Buffer sizes from 32KB through 256KB seem reasonable, depending on the available memory. With respect to constrained memory – e.g., for pumping data between all sites of a cluster – and CPU costs, 64KB seems a good choice.
The CPU load can be modeled as: *A*(Cb_Src+Cb_P+Cb_Snk)+A/RS*(Cr_Src+Cr_P+Cr_Snk).* Where *A* is the amount of data, *RS* is the request size, and *Cb_xxx and Cr_xxx* are the respective per-byte and per-buffer CPU costs of the used source, sink, and the pump. For a network source, the per-request costs are computed per complete request. We gave our approximations for these parameters and compared them with our measurements for each isolated component as well as for a local and a remote disk copy combining different components. Table 8 summarizes these results.

	Pump:	Disk Source	Disk Sink:	Network Source:	Network Sink:
Cost per Byte:	0 ns	0.5 ns	1.6 ns	40 ns	20 ns
Cost per Request:	20 µs	66 µs	53 µs	0 µs	20 µs

4 Acknowledgements

Thanks go to Joe Barrera and Josh Coates, on whose earlier code versions our data pump is based. Thanks to Donald Slutz for support with the Rags cluster. Thanks to Maher Saba, Ahmed Talat, and Brad Waters for their help in performing and understanding the soaker experiments. Thanks to Leonard Chung, whose soaker code we used.

5 References

[1] Leonard Chung, Jim Gray, Bruce Worthington, Robert Horst: Windows 2000 Disk IO Performance. Microsoft Research Technical Report MS-TR-2000-55, 2000.

[2] Riedel, Erik, Catherine van Ingen, and Jim Gray: A Performance Study of Sequential IO on WindowsNT 4.0. Microsoft Research Technical Report MSR-TR-97-34, 1997.