Data Transfer Experiments with a Fusion-io Flash-Memory Disk and an XtremeData FPGA Accelerator

2008 Status Report for the Storage Intensive Supercomputing (SISC) Project

Craig Ulmer
Sandia National Laboratories
Livermore, California

1. Executive Summary

The Storage Intensive Supercomputing (SISC) project at Lawrence Livermore National Laboratory is an effort to improve the performance of data-driven applications through advances in system architectures. The Sandia contribution to this project during the summer of 2008 focused on the task of evaluating how data can be moved efficiently between disk and processing resources that are of interest in the SISC project. Specifically, we focused on a commercial flash-memory hard drive from Fusion-io and an FPGA accelerator from XtremeData.

Flash memory experiments were performed using Fusion-io's ioDrive. The ioDrive is a high-performance flash-memory device that is capable of delivering close to 700 MB/s of read performance and over 100K I/O operations per second (IOPS) to an end application. After observing that the ioDrive's performance improved when multiple I/O operations are issued at the same time, we developed a set of threaded data-analysis microbenchmarks to determine whether we could exploit this characteristic to improve application performance. In all four examples we found that flash memory boosted performance (2x-200x) and in general enabled applications to make better use of a multicore environment. We conducted additional data transfer experiments to gain a better understanding of the ioDrive's current capabilities as well as to estimate the complexity involved in utilizing different I/O techniques.

We then conducted data transfer experiments between different components of the XtremeData system. This system employs an XD1000 FPGA accelerator that plugs into an Opteron processor socket. We modified the hardware reference design for the XD1000 and built a small number of hardware configurations for measuring how fast data can be exchanged with the FPGA. After numerous attempts, we concluded that it was not possible to move data directly between the ioDrive and the XD1000 due to limitations of the FPGA's HyperTransport core and the Linux kernel's restrictions on direct data transfers. Instead, we focused on modifying the software for a previous n-gram text classification application to perform one-copy transfers between the ioDrive and the XD1000 in a threaded, double-buffered manner. Ultimately we were able to boost the overall processing rate of the system from <5 MB/s to 100 MB/s when processing a large number of small (2 KB) files.

2. Flash Memory Experiments

A key portion of our research this year focused on determining how application designers could exploit the fundamental characteristics of Fusion-io's ioDrive[1]. The ioDrive (Figure 1) is an emerging flash-memory storage device that is optimized for performance instead of compatibility with traditional interfaces for storage devices. The ioDrive's hardware has three characteristics that set it apart from other devices. First, it is a PCIe x4 card that can support high-bandwidth data transfers with the host. Second, the card employs a large number of flash-memory chips that are arranged to exploit parallelism both horizontally (i.e., bus width) and vertically (i.e., die stacks). Finally the card implements a high-throughput transaction manager in hardware that allows multiple transactions to be processed concurrently. These architecture features enable a single card to deliver up to 700 MB/s of read performance and 100K IOPS I/O operations per second (IOPS) to an end application.

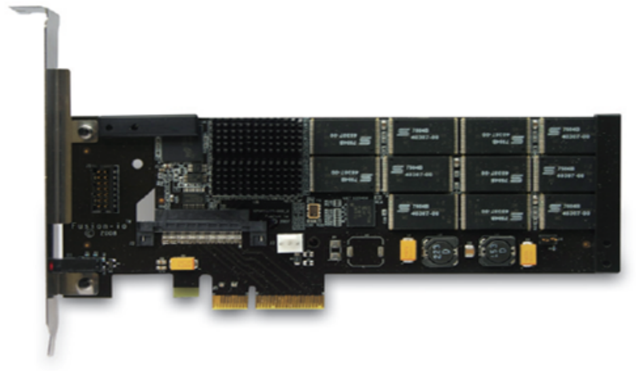


Figure 1: Fusion-io's ioDrive is a PCIe x4 card populated with 80-320 GB of NAND-Flash memory and an FPGA that is capable of managing many concurrent I/O operations.[1]

During the early stages of our investigating into the low-level performance characteristics of the ioDrive we observed an interesting effect: in several applications, increasing the number of I/O-performing threads increased the overall data-transfer performance of the ioDrive. This effect is opposite of what we have come to expect from traditional hard drives, where concurrent transactions generally degrade performance because they result in disk thrashing. As such we devised a number of tests to quantify the low-level performance characteristics of the ioDrive and provide examples of how threaded applications on a multicore system could exploit these characteristics. Additionally, we explored asynchronous I/O methods for achieving high-performance without resorting to threads.

The performance numbers reported are for a single server that employs two quad-core CPUs (2.33MHz Intel E5345s), 2 GB of memory (PC2-5300f), one 80 GB Fusion-io ioDrive, and three SATA drives arranged in a software-controlled RAID0. The system utilizes the Linux 2.6.23 OS found in Fedora 8. We utilized 8 GB input files (64 million vectors of 32 single-precision floating-point values) as well as cache-flushing routines to negate caching effects.

2.1. Multithreaded Microbenchmarks

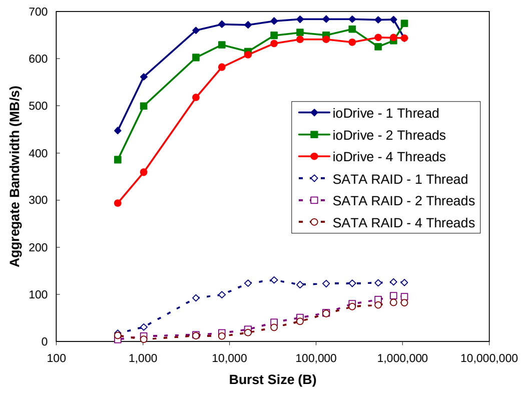
As a means of observing the impact of multithreaded performance on storage devices, we constructed four multithreaded microbenchmarks that perform operations commonly found in data-intensive applications. While the applications all perform computations involving vectors of floating-point numbers, we selected operations that are I/O bound instead of compute bound in order to stress the storage subsystem. The microbenchmarks are threaded at a coarse granularity and assume out-of-core operation, where datasets are much larger than the capacity of main memory. While a reasonable amount of effort has been made to maximize performance, we have not taken heroic measures to optimize the microbenchmarks to a particular system architecture (e.g., application-level caching). The intent is to give an idea of the performance that can be obtained using built-in features of the hardware (e.g., multiple cores) and operating system (e.g., OS file caches). The microbenchmarks are block transfer, k-nearest neighbors (kNN), external sort, and binary search.

[1] Photo credit: Fusion-io.

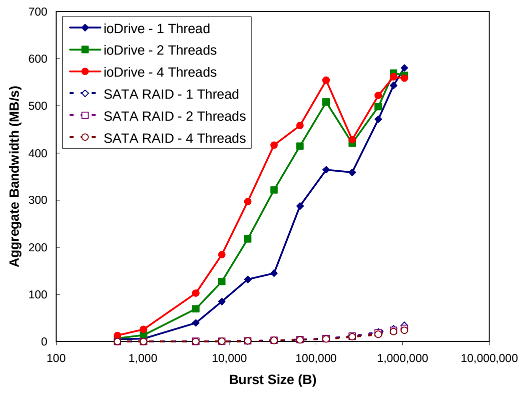
Block Transfer

Similar to other benchmark programs such as IOzone [2], the block transfer microbenchmark measures raw data-transfer performance characteristics of a storage device. The block transfer program invokes multiple threads that issue either read or writes to sequential or random locations within one or more files. Transfers are intentionally misaligned in order to minimize overlap and reduce caching effects. Performance is reported in terms of the aggregate amount of data transferred per second.

The results of the read tests for both the ioDrive and the SATA RAID are presented in Figure 2(a-b). In terms of (a) sequential read performance, the ioDrive provided 5x the bandwidth of the SATA RAID for all burst sizes, and achieved a maximum of 683 MB/s compared to the SATA RAID's 125 MB/s. Performance degraded in both devices when multiple reader threads were employed. We speculate that this drop for the ioDrive may be due to the fact that the kernel can orchestrate large, sequential transactions for a single thread very efficiently (i.e., large sequential transactions are broken into a set of smaller, independent transactions).



(a) Sequential Reads



(b) Random Reads

Figure 2: Read performance for (a) sequential and (b) random access patterns using the ioDrive and a three hard drive SATA RAID.

While the ioDrive's sequential performance is very good, the true benefit of flash memory becomes apparent when I/O is nondeterministic. The results of the random read test are presented in Figure 2(b). For all burst sizes the ioDrive provided at least a 17x improvement over the SATA RAID for random I/O. Additionally, the ioDrive's performance improved when multiple threads operated concurrently. The dip in performance at 256 KB for the ioDrive can be attributed to the internal block size that the hardware employs. Peak[2] performance measurements for all tests are listed in Table 1.

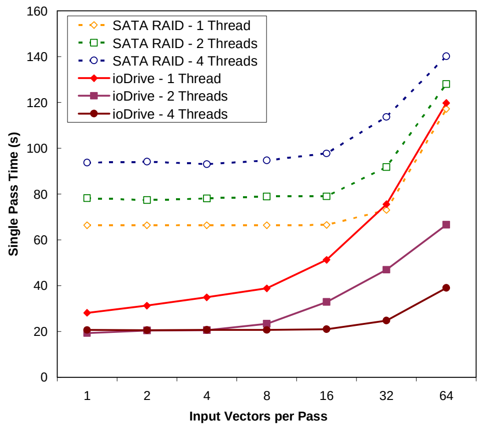
Table 1: Peak performance measurements in the block transfer microbenchmark.

Test	SATA RAID	ioDrive	Speedup
Sequential Read	125 MB/s	683 MB/s	5x
Sequential Write	139 MB/s	661 MB/s	4x
Random Read	34 MB/s	580 MB/s	17x
Random Write	46 MB/s	658 MB/s	14x

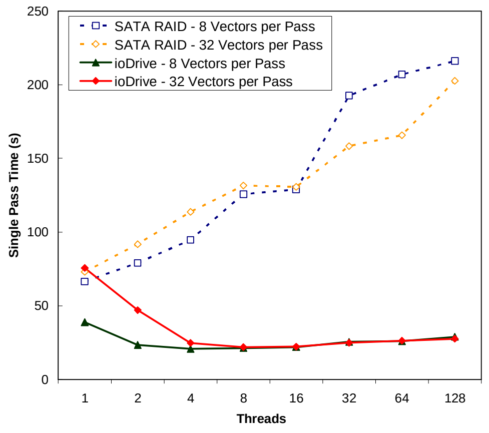
[2] Peak performance refers to the best value seen for all transfer sizes. It is particularly unfair in the random read benchmark, where the SATA drives provide less than 5 MB/s until large (512 KB) block sizes are used.

k-Nearest Neighbors (kNN)

The second microbenchmark implements the k-Nearest Neighbors (kNN) algorithm for classifying input vectors based on their similarity to labeled, training vectors. Each thread in the program reads its own section of the training data and locates the k training vectors that have the shortest Euclidean distance to an input vector. The kNN microbenchmark is also capable of processing multiple input vectors at a time. Increasing the number of input vectors that are processed in a single pass of the training dataset increases the amount of computation that must be performed in a pass, but decreases the number of passes that are required to process multiple input vectors. As such, it is possible to use the number of input vectors as a parameter for adjusting whether the application is compute bound or I/O bound.



(a)



(b)

Figure 3: Total time required for kNN to complete a single pass of an 8 GB dataset when (a) varying the number of input vectors processed at a time and (b) varying the number of concurrently operating threads.

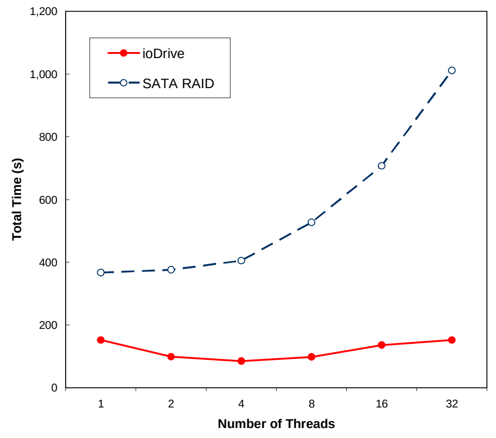
The amount of time required to make a full pass of the training data for a given number of input vectors and threads is presented in Figure 3(a). For the optimal number of threads, the ioDrive is at least 3x faster than the SATA RAID for all input vector loads. Both systems switch from being I/O bound to compute bound when 32 input vectors are processed simultaneously. Figure 3(b) highlights the impact of threading for fixed numbers of input vectors. While the SATA RAID's performance degrades with multiple threads, the ioDrive's improves, up to approximately 4 to 8 threads. Execution times for peak performance are listed in Table 2.

Table 2: Peak performance measurements in the kNN microbenchmark. Values are reported in terms of total execution time.

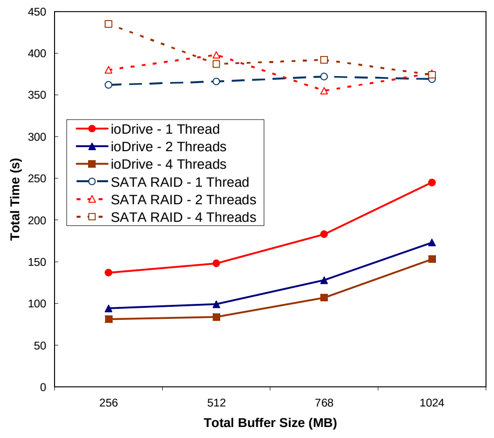
Test	SATA RAID	ioDrive	Speedup
16 inputs per pass	66 s	21 s	3x
32 inputs per pass	73 s	22 s	3x

External Sort

The external sort microbenchmark converts an unsorted file of vectors into a sorted file of vectors. Due to the large size of the input file, this implementation must process data out-of-core in two phases. First, multiple threads read in different sections of the input file, quicksort the individual sections, and then write out the results to intermediate files. Second, a single thread merges all of the intermediate files into an output file in a streaming manner. This thread utilizes a tree data structure[3] to minimize the number of comparisons performed when merging the data streams.



(a)



(b)

Figure 4: External sort performance for ioDrive and a three hard drive SATA RAID varying (a) the number of threads when using a total buffer size of 512 MB and (b) the total buffer space available to the application.

Performance measurements for the external sort microbenchmark are presented in Figure 4(a-b). In (a) the number of threads is varied while the total buffer space available to the program for holding data values is limited to 512 MB. In terms of execution time, the ioDrive tests completed 2-4x sooner than the fastest SATA RAID test. Performance decreased for the SATA RAID when more threads were employed while the ioDrive achieved maximum performance at 4 threads. In (b) the total buffer size for the application was varied while employing a small number of threads. While buffer size did not have a significant impact on execution time for the SATA RAID, the ioDrive's degraded as buffer size increased. This characteristic is likely to be a side effect of the algorithm's implementation[4], as opposed to deficiencies in the hardware. Peak performance values are summarized in Table 3.

Table 3: Peak performance measurements in the external sort microbenchmark. Values are reported in terms of total execution time.

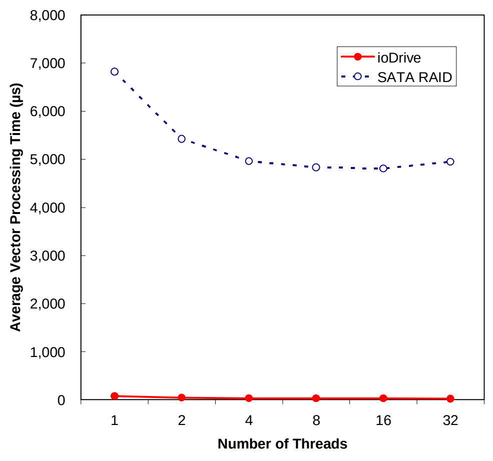
Test	SATA RAID	ioDrive	Speedup
8 GB Sort	361 s	81 s	4x

[3] When merging n files, this structure reduces the number of comparisons performed for each output from n to log(n).

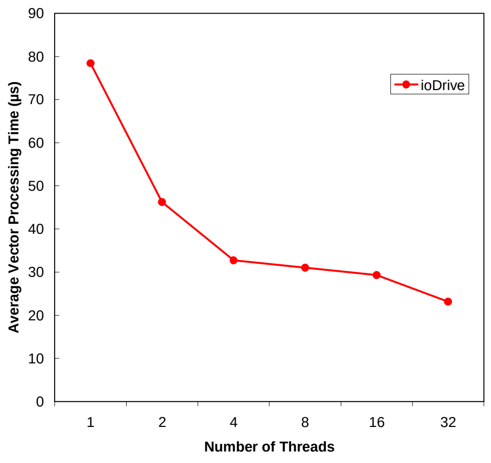
[4] The impact of decreasing the total buffer size is that a larger number of smaller intermediate files are generated. A well-tuned system would find the size and number of files that yields the best performance.

Binary Search

The final microbenchmark performs a binary search on a sorted file to determine whether it contains one or more input vectors. This search requires log(n) vector comparisons to determine where the vector should be located within the sorted file. In order to minimize the number of disk read requests that are issued when searching for an input vector, the microbenchmark utilizes an index that is built in main memory at start time. Performance is reported in terms of the average amount of time required to process an input vector and does not include the one-time initialization cost of building the index.



(a)



(b)

Figure 5: Average amount of time to perform an indexed binary search (a) for both the ioDrive and the SATA RAID and (b) the ioDrive by itself.

The results of an experiment where 64K input vectors were located in an optimally-indexed 8 GB file using a variable number of threads are presented in Figure 5(a-b). For both the ioDrive and the SATA RAID, performance improved as the number of threads increased. When averaging all runs, the ioDrive's average vector processing time was 40.1 µs, which is 132x faster than the SATA RAID's average vector processing time of 5.3 ms. As summarized in Table 4, the ioDrive gave a peak performance speedup of 210x when 32 threads were utilized.

Table 4: Peak performance measurements in the block transfer microbenchmark.

Test	SATA RAID	ioDrive	Speedup
Create 128 MB index	387 s	38 s	10x
Process 64k inputs	315 s	1.5 s	210x

2.2. Asynchronous I/O Experiments

In conversations with Fusion-io, the ioDrive's developers reported that they had had a great deal of success in boosting IOPS performance by switching from a threaded approach to a single-thread performing asynchronous I/O (AIO). The developers explained that the ioDrive's hardware queues are deep enough that a host application can have more than 32 requests in flight before the ioDrive's performance saturates. Observing that our test system only has eight processor cores, we theorized that it was possible that our threaded tests were not reaching peak performance because the processor cores needed to be oversubscribed with threads in order to generate enough I/O requests. Therefore we performed a preliminary investigation into AIO possibilities.

We conducted a brief survey of AIO options that are available for modern Linux systems. After confirming that traditional I/O operations are always blocking commands when used with storage devices[5], we investigated the built-in AIO support provided by glibc's realtime library. The glibc AIO interface is straightforward and easy to use: the user simply issues one or more lists of I/O operations and then polls for completion of the work. We constructed a simple benchmark program to measure the speed at which a file could be read in a streaming, double-buffered manner using glibc's AIO functions. In this test the application manages two sets of buffers. Each read request is comprised of a set of I/O operations that fill independent regions in the buffers.

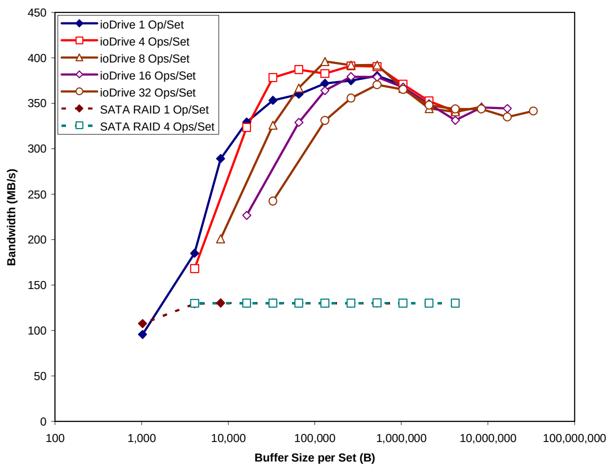


Figure 6: Asynchronous I/O performance with the glibc AIO interface for both the ioDrive and a three hard drive SATA RAID.

We conducted an experiment that varied the number of I/O operations issued in a request (or set) as well as the amount of buffer space available. The test utilized both the Fusion-io ioDrive and three SATA hard drives arranged in a RAID0. As depicted in Figure 6, the ioDrive's performance improved when a small number of outstanding requests were issued at once and the total request size was between 16 KB and 512 KB. Four operations per set provided the best general performance. The SATA RAID did not demonstrate any performance difference when using AIO.

While the glibc's AIO interface demonstrated that performance could be improved in a single-threaded application by allowing multiple I/O transactions to be in flight at the same time, performance was considerably less than what we have observed in the block transfer tests

[5] Passing O_NONBLOCK to an open() call reported success, but read() commands always blocked until they completed. Kernel mailing lists explain that this is due to the one-sided nature of file operations, compared to the two-sided nature of sockets (i.e., file reads in the kernel do not have a thread managing the file that can be left alone to do its work).

reported in section 2.1. One observation is that the block transfer test assigns different portions of the input file to each thread in order to remove caching effects. This assignment may provide the ioDrive hardware with better parallelism, as data may be more evenly distributed across different die planes. However, it is also quite probably that we are losing performance in the glibc AIO implementation. While investigating the AIO implementation, we learned that glibc's AIO calls are implemented through user-space threads. It has been widely reported that the Linux kernel's support for asynchronous file I/O is a better approach because these I/O operations are managed by kernel threads. The libaio library provides AIO through the kernel's asynchronous file I/O interface. However, this library is more challenging to use than glibc. Our initial tests indicate that it does give better performance (approximately 430 MB/s), but we have not yet conducted a full set of experiments with libaio.

Our experiences with both the glibc and libaio AIO interfaces have provided us with insight into the hardships of working with AIO. In general, AIO can be challenging to utilize in a manner that maximizes performance because the developer must do a fair amount of profiling and hand tuning to make sure transactions are scheduled properly. While this optimization may not be difficult to perform in applications that have regular I/O patterns, many typical applications will not map well to this form of I/O. As such, our preference is to continue developing parallel I/O applications using threads.

2.3. Additional Experiments: Fusion-io RAID

Fusion-io reported in April that their drivers were stable and that multiple cards could be used together in a single computer. We inserted three ioDrive cards in sisc2.llnl.gov[6] and attempted to configure them as a single RAID. While the device drivers recognized all of the cards and were able to perform the low-level formatting operations that are specific to the ioDrive, the system locked up when we attempted to create a software RAID with the drives. Fusion-io confirmed that there were still physical-layer compatibility issues with PCIe in some motherboards that were being examined. While Fusion-io has since reported that this bug has been fixed in newer firmware, we have not had time to conduct the experiment again.

[6] This machine utilizes AMD components. All of our other tests have been conducted on Intel components.

3. XtremeData XD1000 Experiments

XtremeData is a company that develops data-analysis systems that are optimized for applications with massive datasets. While XtremeData largely utilizes commodity hardware and software in their commercial, distributed-database product, they have developed two key pieces of intellectual property that give their products a competitive advantage. First, they have adapted a PostgreSQL database engine to run in a distributed manner across a cluster. Second, they have developed an FPGA accelerator that plugs into an Opteron socket and can be used to accelerate data-processing operations. LLNL purchased a single XD1000 system in order to evaluate the performance characteristics of the FPGA accelerator. While the database software is not available on this platform, there are sufficient hardware and software examples available that we can prototype SISC data-processing applications on this system. Our primary goal during this phase of the project was to install the Fusion-io ioDrive in the XD1000 and measure how fast the system could process data from the disk.

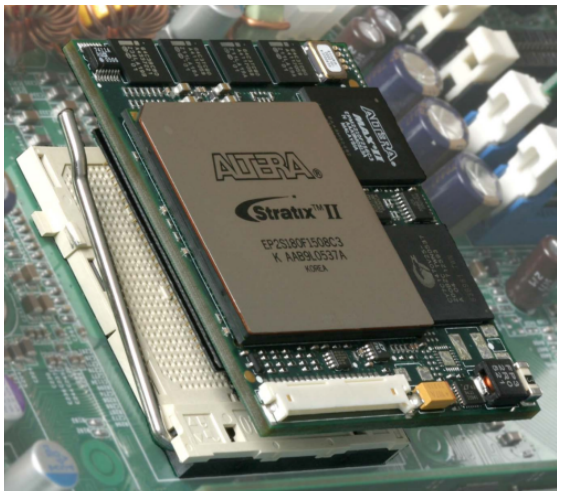


Figure 7: The XtremeData 1000 FPGA accelerator features an Altera FPGA that plugs into a standard AMD Opteron socket and communicates with the host using HyperTransport.[7]

3.1. XD1000 Environment

The XtremeData XD1000 is an FPGA accelerator board that plugs into an AMD Opteron socket and communicates with the host via a HyperTransport (HT) link that operates at 800+800 MB/s. The XD1000 board features an Altera Stratix-II part, 4 MB of on-board SRAM, and a DDR interface to the socket's memory slots.

XtremeData provides a reference hardware design for the FPGA that demonstrates the basic components of the XD1000. The design employs two interfaces for exchanging data with the host: one for host-initiated transactions (i.e., control) and another for FPGA-initiated transactions (i.e., DMA). Host-initiated transactions are handled by a control unit that uses the address specified in the transaction to select the hardware unit in the design that is the target of the operation. The control unit is expected to be used as a means of interacting with control/status registers in the FPGA, and therefore was not designed to support high-bandwidth communication[8]. High-bandwidth transfers are instead achieved through a pair of DMA engines

[7] Photo credit: XtremeData, Inc.
[8] The control bus only extracts the first 32-bit value extracted from a HyperTransport packet.

that allows the FPGA to initiate block transfers of data with the host. This DMA interface operates in a streaming manner that is regulated with FIFOs.

Based on these interfaces, the procedure for utilizing the FPGA to process an application's data is as follows.

- **Compute Engine Setup**: A user application issues programmed-I/O (PIO) operations to the control unit to configure the computational core loaded in the FPGA (e.g., computation length, job ID).
- **DMA Setup**: The application pins and translates host memory for both input data and output data. The physical addresses and sizes for each transfer are programmed into the FPGA's read and write DMA engines.
- **Streaming DMA**: The FPGA's DMA read engine pulls data in and sends it word-by-word to the computational circuit. As the user's circuit processes data it pushes it to the DMA write engine, which in turn pushes data out to host memory[9].
- **Completion Interrupt**: Notification of completion is typically handled with an interrupt. Either of the two DMA engines can be configured to issue an interrupt when a specific number of transactions have completed.
- **Compute Engine Read**: A user may also use the control interface to read data out of the FPGA (e.g., status registers or final results).

3.2. Previous XD1000 Work at LLNL

During the summer of 2007, Arpith Jacob implemented an n-gram classification application that utilized the XD1000 FPGA as a means of performing many comparisons in parallel. Ideally this program would read data from disk, stream the data to the XD1000 for analysis, and then extract statistics from the analysis to estimate the language of the file. After initial tests confirmed that the hard drive could not source data fast enough to keep the XD1000 saturated, Arpith modified his software to read all data files into host memory a priori and then measured the rate at which data in host memory could be processed. In the best case, the system was able to achieve 478 MB/s. The next step in this work would have been to measure how fast the system could process data using the ioDrive as a disk source. Unfortunately, these tests were not possible last year due to firmware incompatibilities between the ioDrive and the XD1000. Since then driver enhancements have fixed these problems, thus allowing us to reopen work with the XD1000.

3.3. FPGA Experiments

As a first step in building hardware accelerators for the XD1000's FPGA, we examined the reference design in detail and made modifications to speedup the build process.

Minimal Design: While the reference design demonstrates all of the hardware components on the XD1000, it takes hours to build due to the timing requirements and sizes of the high-speed components. Our first hardware effort was to make a minimal design that housed a simple processing core. Removing the unwanted cores (DDR and SRAM) was a straightforward task, as the reference design allows the user to select a NIL architecture implementation for each component at the top level of the design. We then updated the computational unit of the design with instrumentation registers to monitor data flow from the host. Finally, we modified the build

[9] The outgoing DMA engine requires both a valid DMA transaction from the host and data from the user's FPGA hardware in order to complete a transaction. This requirement is the source of many lockups when debugging an application.

directory for the design to be more organized and added Makefiles to allow the user to build the design without having to use the Altera Quartus GUI. The minimal design now builds in approximately 15 minutes on fpga2.llnl.gov.

Block RAM Design: In order to conduct the PIO experiments described in section 3.4 we needed a design that allows a user to write a sizable amount of data on to the control bus's address space. While the default design provides a 64 KB window, its address space only allocates 256 bytes to each unit in the design. Therefore we modified the minimal design to provide 4 KB of addressing to each unit and then updated the design to instantiate a 4 KB block RAM to house data

Floating-point Design: While current generation FPGAs lack native support for floating point, most vendors provide floating-point cores that can be utilized in a design. We investigated Altera's options and found that the Megawizard could be used to generate pipelined floating-point units for all typical operations. As a means of verifying that these cores functioned properly, we updated the computational portion of the minimal design with an array of four single-precision floating-point units. We then loaded the design on the XD1000 and verified that data values programmed into the device were properly added by the floating-point units.

3.4. Injection Tests: Programmed-IO and O_DIRECT Experiments

One issue with the XD1000 reference design is that the FPGA is expected to move data to and from host memory. While this technique is sufficient for many applications, it implies that data being transferred from disk must be moved in a 1-copy manner (i.e., data must be transferred to a host buffer before being moved to the FPGA). During our conversations with Fusion-io, there was speculation that it might be possible to perform a 0-copy (i.e., data is transferred directly from disk to the FPGA) through simple addressing techniques. In this approach, the application would (1) memory map a portion of the FPGA card's address space for programmed-io access and then (2) supply the memory-mapped address to the disk read calls. The hypothesis was that virtual-to-physical address translation would take place when the read call was issued, and that the translation would be sufficient for allowing the storage device to write to the intended destination. We constructed a hardware design that contained a hardware unit for accepting data from the control port and added instrumentation to monitor transactions.

In our first set of tests we issued data transfers using the method that was just described. While the OS allowed us to perform these data transfers, we observed that performance was only approximately 75 MB/s. Additional tests measuring the speed at which data could be transferred from the host to the FPGA using PIO revealed the same performance. Traditional techniques for overcoming PIO performance (e.g., write combining and non-temporal SSE copies) did not provide any benefit. In discussions with XtremeData, we learned that this performance was typical for the control port through which we had been routing data, and that in the best case the port was only capable of operating at 133 MB/s.

Following the advice of Fusion-io, we modified the host application to open the source file in raw mode (i.e., set the O_DIRECT flag when opening the file and page align buffers). The hypothesis was that the previous tests were actually performing a 1-copy internally, as the kernel moved data into a buffer cache and then issued an additional copy to transfer the data to its final location on the FPGA. Unfortunately the O_DIRECT flag caused the read command to exit with errors. While researching the problem, we located two other instances in newsgroups where users had tried to move data directly from a disk to a memory-mapped location (e.g., a video card screen buffer). Both of these threads pointed to the kernel's current inability to allow a PIO address to be the target of a transfer while O_DIRECT is enabled. As such we were forced to abandon a zero-copy approach.

3.5. Injection Tests: One-Copy

The next step in our testing of the XD1000 was to determine its performance characteristics given that a one-copy approach was necessary for moving data between the ioDrive and the FPGA. For the hardware side of this work we utilized a minimized version of the XD1000 design, which simply accepted and discarded all data pulled in by the DMA engine. We conducted two sets of experiments: one for determining application injection performance when working with large files and another for small files.

Large File Tests

The first set of one-copy injection tests focused on transferring a single, large (8 GB) file from the ioDrive to the XD1000 FPGA using a double-buffered, one-copy approach. These tests provide an upper bound on performance because large files generally yield better performance than small files. Host memory for the buffering was pinned and translated during initialization to remove kernel overheads. The experiments varied whether direct mode was enabled when reading data from disk in order to answer whether overhead could be reduced by forcing incoming data to land in host memory without any kernel-level buffering.

Table 5: Injection performance when moving an 8 GB file from disk to the XD1000 using an intermediate host buffer.

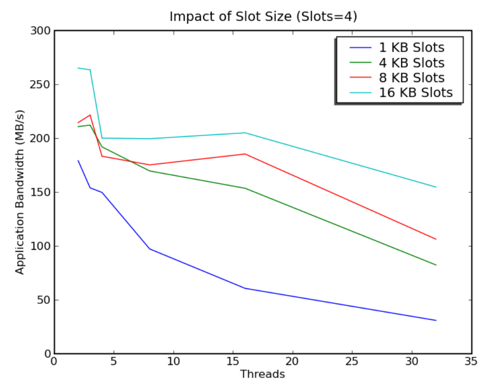
Transfer Source	O_DIRECT	Performance
Hard Drive	no	63 MB/s
	yes	51 MB/s
ioDrive	no	401 MB/s
	yes	513 MB/s

Performance results for the large-file injection experiments are presented in Table 5. As these numbers indicate, the ioDrive performs approximately 9x faster than the local hard drive. While the overhead of moving data to the FPGA degrades the ioDrive's performance (i.e., from >600 MB/s to 513 MB/s), the ioDrive is still is able to deliver a considerable amount of bandwidth. Interestingly, the O_DIRECT flag hinders the hard drive's performance while improving the ioDrive's. In terms of concurrent DMA operations, we found that an application needs to issue 16-32 page-sized transfers (64 KB – 128 KB of data) to the XD1000's DMA engine at a time in order to achieve more than 500 MB/s.

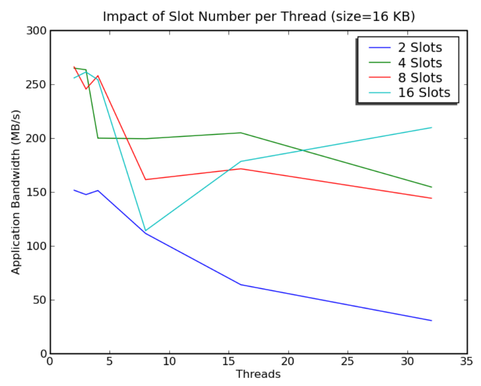
Small File Tests

We conducted a second set of injection tests to determine how the use of small files affects performance. In general we expect small data transfers to perform worse than large transfers. For example, the sequential block read benchmark (Figure 2(a)) shows that the ioDrive loses about 100 MB/s when burst size is reduced from 4 KB to 1 KB. Unfortunately, the n-gram applications a more difficult challenge because it involves reading a large number of small (2 KB) files. It is difficult to obtain high bandwidth in this scenario because the amount of overhead associated with opening and closing an individual file is substantial compared to the amount of time spent reading data from the file.

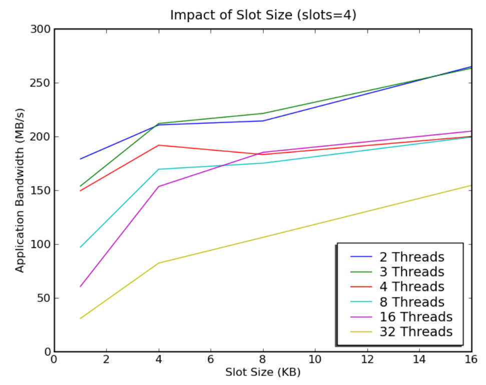
In order to improve concurrency, we constructed a threaded file streaming application that performs work on multiple files at the same time. This software employs a small number of threads for pulling data from different files into host memory, and a writer thread that pushes data out to the FPGA. Each reader thread has a fixed amount of buffer space for housing data that is divided into slots, and a list of input files that need to be processed. At runtime the writer selects the first reader thread that has data and then streams through all of its buffers until the file is transferred to the FPGA. This approach results in (1) double-buffered transfers from the disk to the FPGA, (2) the ability for multiple page-sized DMA transfers to be in flight at the same time if a slot is larger than a page, and (3) the file reader being able to begin reading from the next file before the current file is completely processed by the FPGA.



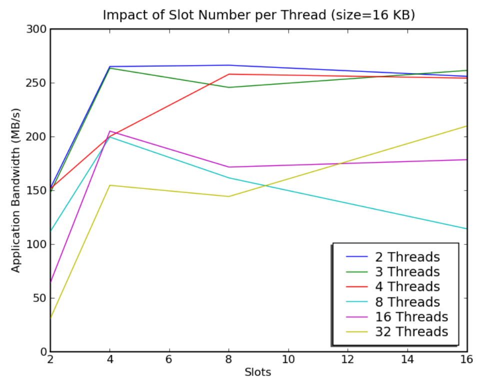
(a)



(b)



(c)



(d)

Figure 8: Transfer performance was measured for different configurations. The four views of the same data represent (a, c) the impact of slot size for a fixed number of slots and (b, d) the impact of the number of slots for a fixed slot size.

We measured the bandwidth that could be obtained by the injection system while varying the number of slots, the size of the slots, and the number of reader threads. The results are presented in Figure 8 (a-d). In general the system worked best when a small number (2-3) of threads and a small number (2-4) of slots were employed, and the slot size was large enough (16 KB) to hold an entire file. These measurements indicate that the input files were too small to benefit from streaming (i.e., breaking a file read into multiple transactions was not worthwhile), but at the same time, the overhead for opening and closing files could be hidden partially by allowing multiple files to be processed concurrently. Working on too many files at the same time caused interference that degraded performance. Ultimately the system delivered approximately 270 MB/s under optimal conditions.

3.6. Updating the n-gram Design

Based on our injection experiments we updated Arpith's software to make use of the threaded file reader. This implementation took a considerable amount of time to debug due to the complexities involved in orchestrating DMA transfers properly in the flow of a real application. After numerous system crashes we were forced to abandon our "pin all buffers at start time" approach in favor of a "pin buffers at injection time" technique. This change reduces performance but was necessary for stability. We verified that the implementation produced the same results as the implementation from last summer.

Using the 40 MB French language reference dataset, we tested the updated n-gram design using both the local hard drive and the ioDrive. As summarized in Table 6 the ioDrive provided a 20x speedup over the hard drive.

Table 6: The end-to-end performance for the n-gram application.

Transfer Source	Performance
Hard Drive	5 MB/s
ioDrive	100 MB/s

Observing that the FPGA design is capable of operating at speeds much higher than 100 MB/s, we began investigating the bottlenecks in the system. We categorized the main data flow operations in the host program into three functions: reading data from the ioDrive, using the XD1000's DMA engine to move data to the FPGA, and reading the result values out of the FPGA's memory. We then timed the application when various functions were removed. The results of the tests are presented in Table 7. From these measurements we observe that injection performance (Fusion-io+XD DMA) is about 70 MB/s less than what we observed in the injection tests. This drop may be due to new overheads caused by on-demand pinning or by application-specific PIO operations that take place when data is injected. We expect that performance could be improved by tuning the software to guarantee that data transfers are aligned to take place in parallel as much as possible.

Table 7: Breakdown of performance in n-gram.

Operation	Performance
Fusion-io + XD DMA + Readout	100 MB/s
Fusion-io + XD DMA	135 MB/s
Fusion-io	200 MB/s

4. System-Level Observations

The work performed in this project has provided us with insight into data transfer challenges in the local node of a SISC-like system. While the ioDrive provided a 20x performance improvement in the end-to-end performance of the n-gram application, the current application implementation delivers only a fraction of the performance the hardware is capable of achieving. It is important to highlight a few of the key weaknesses of the implementation in order to shape future efforts.

4.1. One-Copy Limitations and Injection Issues

The largest disappointment in this work was that data could not be routed directly from the disk to the FPGA accelerator because the current Linux kernel does not allow peripheral devices to be the target of an O_DIRECT read. This limitation forced us to employ host buffers for one-copy transfers. While the host system provides enough memory bandwidth to support one-copy transfers at reasonable rates, the application software requires a good bit of hand tuning to ensure that enough transactions are in flight at the same time to keep the data flow running at peak levels. Additionally, there is the hazard that one-copy overheads may interfere with other processing tasks that are running in the system and vice versa.

A second source of performance limitation in the XD1000 system was PIO injection performance. The XD1000's HyperTransport core provides a low-bandwidth PIO interface and a high-bandwidth DMA interface. This split is based on the common false assumption that the only way to transfer data efficiently with a peripheral device is through the device's DMA engine. Through techniques such as write-combining or SSE copying it is in fact possible to achieve significant application-to-peripheral bandwidth. For example, the Cray XD1 [3], which employed a HyperTransport FPGA accelerator much like the XD1000's, was able to achieve over 1 GB/s in injection performance using PIO. The advantage of using PIO is that end applications can use simple memory copy operations to orchestrate injections instead of having to queue DMA transaction requests and poll for completion. Additionally, one-copy transfers with PIO become more automated, as the user simply provides the memory-mapped address of the peripheral device's memory to a read call, which in turn moves the data from disk to peripheral device by way of a kernel buffer. As we have observed with the ioDrive, simplifying the programming interface makes it easier to scale up to a larger number of threads.

4.2. FPGA Design Limitations

The original n-gram FPGA design was implemented in a streaming manner that matched the XD1000's reference designs: the user issues PIO requests to initialize the unit, data is streamed into the FPGA through DMA engines, and then the user uses PIO requests to DMA the results to host memory. While this approach works well for individual jobs or cases where all input data can be batched in memory a priori, the book keeping overhead may be significant in applications where there are a large number of jobs or small amounts of data. In order to improve this performance, the design could be modified to queue multiple output results together to reduce how frequently the application needs to pull output out of the card. Further queuing could be employed on the input side of the design to provide slack in the data flow and allow multiple requests to be queued at the same time.

4.3. Accelerator Data Flow Philosophy

Based on our experiences, we are currently formulating a philosophy for how applications should be designed in order to leverage hardware accelerators. This philosophy follows a data-flow design style and is comprised of the following key points.

- **Distributed Control:** Control in systems with multiple resources is managed either centrally (e.g., the host application orchestrates all data transfers) or in a distributed manner (e.g., each resource manages its own transfers). While centralized control simplifies the amount of development that is required in the resources, it is challenging to implement centralized control in a way that maximizes performance for systems with more than one resource. Instead we advocate a distributed approach where each resource is equipped with enough sophistication to be able to perform its computations and data transfers without significant external guidance.
- **Many Jobs In-Flight:** In systems where different resources are used to perform different computational operations, it is beneficial to allow multiple jobs to be in-flight at the same time. This concurrency allows designers to hide overheads by overlapping data transfer and computation at a resource without byte-by-byte streaming[10] interfaces.
- **Push-based Transfers:** Whenever possible, data should be pushed (e.g., moved by the sender) instead of pulled (e.g., moved by the receiver). In a push-based system data is transferred as soon as it can be moved. In a pull-based system, the receiver may need to block until it retrieves the data it requires. Both approaches must handle virtual/physical address translation when exchanging data with the host.
- **Input-Buffered Queues:** In order to efficiently support having both many jobs in-flight and push-based transfers, it is necessary to implement input-buffered queues at the resources. These queues allow designers to hide injection/ejection rate mismatches between stages and provide a system where a resource's input data is buffered in close proximity.

5. Summary

During this phase of the SISC project we investigated the data transfer properties of key components in a SISC node. The Fusion-io ioDrive provided exceptional performance and enabled several microbenchmark applications to execute in less than half the amount of time that they had required previously. More importantly, the ioDrive works extremely well in a multithreaded environment. This feature is especially attractive as multicore architectures become more available. Likewise, it is reassuring that well-known programming methodologies such as threads work well with the ioDrive.

However, as we observed with the n-gram application on the XD1000, application designers must still pay close attention to the characteristics of the hardware in order to obtain peak performance results. I/O work is still challenging when data transfer sizes are small. Additionally, real applications may have constraints that cause performance to evaporate unless a developer micromanages data transfers. Ultimately we believe these hardships can be managed, provided that designers follow the lessons learned in data flow architectures.

References

[1] Fusion-io website: http://www.fusionio.com
[2] Iozone Filesystem Benchmark: http://www.iozone.org
[3] Cray Canada, Inc., "Cray XD1 Technical Specification Release 1.4," 2005.

[10] While streaming interfaces provide a powerful programming abstraction, they can be problematic in several real-world instances, including instances where the computation fetches data non-sequentially (e.g., FFT), or when an I/O bus re-orders data transfers to improve performance.