Accelerated FPGA Based Encryption*

Joseph Fernando, Dennis Dalessandro, Ananth Devulapalli, Kevin Wohlever
Ohio Supercomputer Center - Springfield
{fernando, dennis, ananth, kevin}@osc.edu

Abstract: *As the trend for increased accessibility to data increases, encryption is necessary to protect the integrity and security of that data. Due to the highly parallel nature of the AES encryption algorithm, an FPGA based approach provides the potential for up to an order of magnitude increase in performance over a traditional CPU. Our effort will showcase the capability of FPGA based encryption on the Cray XD1 as well as other FPGA related efforts at OSC's center for data intensive computing.*

Keywords: FPGA, Encryption, MPI, Resource Manager

1. Introduction

Encryption in general and Advanced Encryption Standard (AES) in particular[1], is an application that is very friendly for Field Programmable Gate Array (FPGA) architecture. This is mainly due to the fact that all computations are based on bit manipulation. AES uses Finite Field Arithmetic for all of its computations. One of the characteristics of Finite Field Arithmetic is that addition and subtraction is done by XOR operations on the two inputs and consequently does not produce a ripple carry bit. This speeds up the computations considerably, making the AES algorithm a good candidate for FPGAs. Another advantage is that the computations can be replicated many times and could be easily parallelized for the Electronic Code Book (ECB) [1] mode of the AES algorithm.

The ECB mode of the AES algorithm is used in the FPGA based implementation. We also demonstrate a Resource Manager built on top of the Message Passing Interface (MPI) [2], which facilitates using multiple FPGAs in parallel to accomplish a task.

2. Motivation

Many researchers have studied the AES algorithm for both software and hardware implementations [3]. The AES implementation in the OpenSSL [4] software is extensively used at present. Most implementations of AES encryption like OpenSSL are software based, meaning they run on a CPU. However, CPUs are not efficient in bit operations like XOR, used extensively in AES. This is mainly due to the fact that the CPU cannot replicate processing units. In environments where encryption is used for secure file systems and communication, the CPU would be overloaded easily even at low traffic levels.

Our motivation for this project is to offload mundane encryption computations entirely to the FPGA, thereby freeing the host processor to other more useful computations. In addition, we want to leverage the unique features of FPGAs as well as the Cray XD1 [5] architecture to speed up the AES algorithm.

3. Problem Definition

Performing AES Encryption [1] is a computationally expensive operation. Often times encryption is not the main focus of an application, rather it is something that the application provides as part of its service, an example of this is OpenSSL [4]. OpenSSL uses AES encryption to secure the integrity of the data being transferred. The process of encrypting and decrypting data can take up a large percentage of the CPU's time. We aim to alleviate the amount of work done by the CPU by offloading AES encryption to FPGA.

This introduces a new problem. In the Cray XD1 [5], we have a unique system where user applications may have direct access to reconfigurable hardware (FPGAs). The problem lies in the fact that there are a limited num-

*Support for this project was provided to the Ohio Supercomputer Center through the Department of Energy ASC Program.

ber of FPGAs (six). If a user application needs the FPGA, it is often the case that it will have ownership of the FPGA for its entire lifetime. There is nothing to prevent this, other than the time limit exceeding on a batch job.

As with OpenSSL [4] and its use of encryption, the user application often times only needs to use FPGA for a small portion of its overall functionality. The time that the user application holds ownership of the FPGA and does nothing with it is wasted time. Other users could be waiting for access to the FPGA device.

We aim to alleviate this problem by creating a resource management infrastructure that will control the FPGAs and their associated functionality. This has the added advantage that the user application does not need to directly interface with the FPGA device. Through a simple client-server mechanism the user application can request services, and exchange the data. This also facilitates parallel use of multiple FPGAs. For example, instead of one FPGA encrypting 12MB of data, we can have 6 FPGAs encrypt 2MB of data each. MPI [2] over the high performance Rapid Array Interconnect [5] is used for this purpose. It is important to note that this FPGA infrastructure is not limited to AES Encryption, rather any FPGA based functionality could be made to work with this system, be it encryption, compression, signal processing, etc.

Another issue is, that a user application may only access the FPGA that is attached to the host node. Since FPGA nodes could be used for other jobs, applications which do not need the FPGA may prevent those that do from having access. The client-server model we have chosen allows user applications running anywhere on the XD1, perhaps even outside of the XD1 to have access to the FPGA's functionality.

What we have implemented thus far is, a resource manager which controls the operation of AES encryption and decryption in the FPGAs. The client-server interface between this resource manager and user applications is currently in development. Fig. 1 shows the overview of the proposed resource manager. Further details on the current implementation follow in section 5.

4. Technical Resources

The Cray XD1 [5] cluster computing system was used as the host for this research effort. The goal was to leverage the unique features of FPGA devices that are available on the XD1 for AES encryption.

The Cray XD1 that is available at the Ohio Supercomputer Center (OSC) is a cluster with 36 Opteron processors running at 2.2 GHz in three chassis. Each chassis has six Symmetric Multiprocessor Processor (SMP) units, and each SMP has two Opteron processors. One of the chassis also contains six FPGA accelerator cards. Each

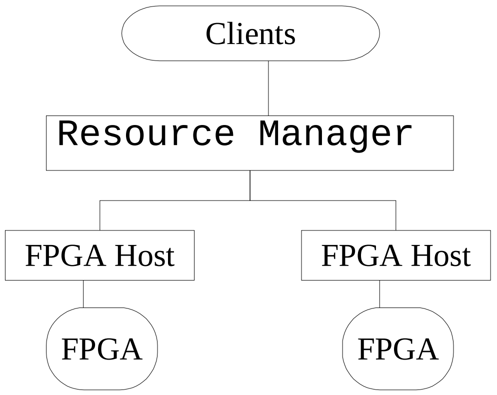

Fig [1]. FPGA Resource Management System.

accelerator card hosts a Xilinx Virtex II Pro 50 device with a -7 speed rating. All of the SMPs are connected through a high speed interconnect known as the Rapid Array [5] with an effective bandwidth of nearly 10.5 Gbps [6]. The FPGAs are connected to the SMP through a Rapid Array Processor (RAP). Given that the maximum clock rate for the FPGA is 200MHz and the FPGA is 64 bits wide, the maximum data transfer rate between the SMP memory subsystem and the FPGA is 12.8 Gbps (200MHz * 64).

4.1 Software Tools

The XD1 at OSC hosts the Riviera SE mixed language HDL design and simulation environment [7]. This environment supports many hardware design languages also called HDLs. In particular, Riviera enables mixed language, VHDL and Verilog simulation. This environment was extensively used for design, development and debugging of circuits. The Xilinx ISE 6.3i development tool set [8] was extensively used to synthesize and map as well as for the place and route of the circuits developed.

5. Implementation Details

In this section, a high level overview of the resource manager is presented. We will also look at one implementation of AES FPGA based encryption as well as its performance, shortcomings, and our plans for possible improvement.

5.1 Resource Management

Currently we have implemented the resource manager and its interactions with the FPGA hosts, as well as the FPGA hosts interactions with the respective AES FPGA bin file. The bin file describes the logic that is loaded into the FPGA and executed. The resource manager and FPGA hosts are started together as one MPI job. The maximum number of processes that can run in this job is the `number_of_FPGAs + 1`. One should ensure that the resource manager is allocated to a non-FPGA node.

When the MPI [2] processes are started, the resource manager waits for all the FPGA hosts to check in. The check in process involves the FPGA host opening the FPGA device and resetting it. A message is passed to the resource manager, to in effect, check in. Once all FPGA hosts have checked in, the resource manager informs the FPGA host, which bin file to load, in our case AES encryption or decryption.

With the bin files loaded, the resource manager begins exchanging the data to be worked on by the FPGAs. Non-blocking MPI send and receives [2] are used in order to avoid waiting for a slow FPGA host if such a situation should arise.

5.1.2 Resource Manager Performance

The performance of the resource manager is also presented to give an idea to the reader about its capabilities. Note that the following results are valid for both encryption and decryption.

Encryption has the same latency as does decryption, the difference between the two especially with files as large as 500MB is statistically insignificant. In other words there may be a 1 or 2 second difference between encryption and decryption, but decryption itself can vary by as much as 1 or 2 seconds between each experiment, as does encryption.

It seems almost intuitive that utilizing multiple FPGAs in parallel will increase overall performance, and for completeness we show this in Figs. 2 and 3. The figures show the throughput of encryption for varying numbers of FPGAs. A 500MB file was used for this experiment. The throughput refers to the rate at which the Resource Manager is able to encrypt the data. As expected the throughput is directly proportional to the number of FPGAs.

As we increase the number of FPGAs the amount, or portion of the 500MB file that each FPGA needs to handle decreases. Since the FPGAs are encrypting less data they complete the task at a lower latency. The Resource Manager has to send and receive the same amount of data, irrespective of the number of FPGAs. This indicates that the Rapid Array Interconnect [5] is not the the bottleneck.

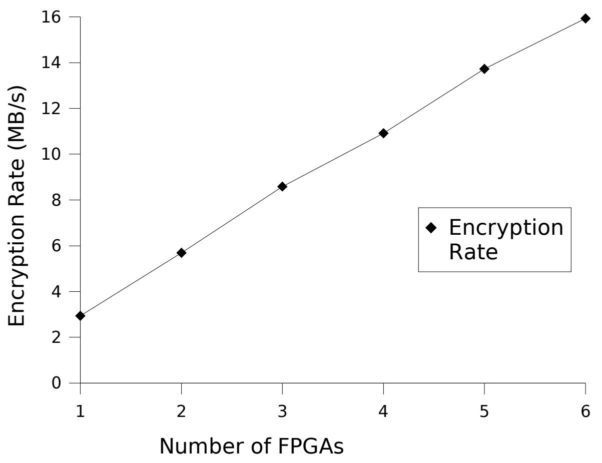

Fig [2]. Encryption rate for multiple FPGAs working in parallel.

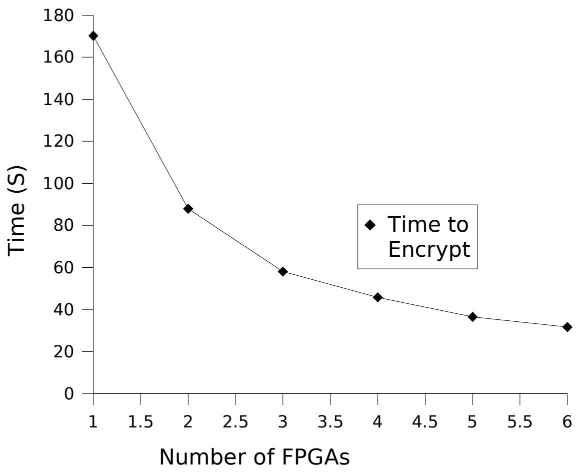

Fig [3]. Time to encrypt 500MB for multiple FPGAs working in parallel.

Fig. 4, shows the latency for software and FPGA based implementations of the AES encryption algorithm [4]. Clearly, a single processor out performs a single FPGA on large data sets, but we will show in the following sections, that the task of encrypting data in multiple FPGAs is much faster than in a CPU. Due to current limitations of our implementation there is significant overhead incurred in getting the data from FPGA host application into the FPGA itself.

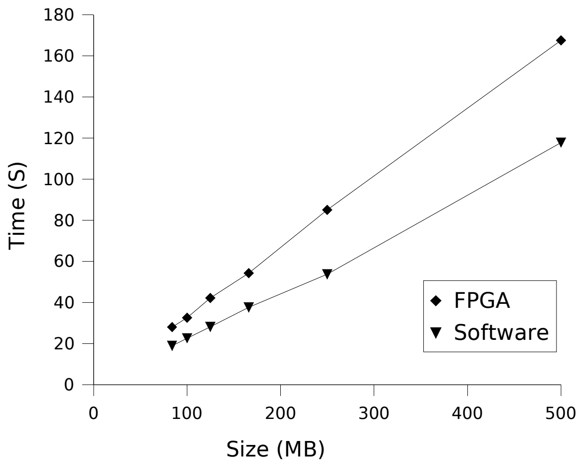

Fig [4]. Time to encrypt files of varying size for both software and FPGA based methods.

Our current implementation however benefits by encrypting in parallel. As shown in Fig. 4, software based encryption of 500MB takes around 115 seconds, while 1 FPGA takes around 175 seconds. However, in Fig. 3, two FPGAs in parallel only take about 90 seconds for the same task. If all six FPGAs are used in parallel it requires approximately 39 seconds, which is considerably less than the software based version.

5.2 Register-based FPGA AES implementation

In the XD1 [5], the CPU can communicate with the FPGA in several ways. The XD1 reserves 128MB of system memory for CPU-FPGA communication[9] [10]. Of the 128MB, currently only 64MB is available for communication, of which 32MB is specifically reserved for interaction with user application registers. The Cray XD1's FPGA API provides the function `fpga_wrt_appif_val()` as well as the function `fpga_read_appif_val()` for writing and reading user application-specific registers. We used this API for our register-based FPGA implementation.

5.2.1 Implementation

The register based FPGA AES implementation is based on a push model framework. The CPU takes the initiative and pushes the data actively to the FPGA, which processes the data. Finally the CPU reads back the results.

Fig. 5 lists the pseudo code of our implementation. AES encryption [1] is designed to operate on 128-bit chunks of data at a time. In our implementation each encryption operation of a 128-bit chunk involves 10 iterations. Therefore a direct translation of algorithm to FPGA

```
foreach 16B of data
    fpga_wrt_appif_val 16B raw-data
    16B enc-data ← fpga_rd_appif_val
end foreach
```

Fig [5]. Register based FPGA algorithm.

	Frequency(MHz)	*Area in slices*
Encryption	160MHz	3010 (12%)
Decryption	150MHZ	3508 (14%)

Table 1. Frequency and resource requirements of AES modules.

hardware will have a latency of 11 clock cycles, with each iteration processed in a cycle. For register based encryption, we followed this approach. Our code was based on an open-source implementation [11]. Table [1] shows the core frequency and FPGA resource requirements of the encryption and decryption modules.

One might argue for a pipeline based approach instead of the non-pipelined approach that we implemented. However, register-based implementation inherently favors the non-pipelined version for two main reasons.

First, the delay of the non-pipelined version is not visible to the CPU, which reads the output immediately after writing to the registers. The function call overhead is high enough for the FPGA to process 16B of data, which is the maximum size of a register [10]. In other words the host application can immediately read the data back after it has written the input data. This is because the FPGA can process the data faster than the host application can call the next function to read.

Secondly, a pipeline-based implementation makes sense for an asynchronous implementation where there is enough data to fill the pipeline. With the register-based implementation only 16 bytes are dealt with at one time. Hence there is not enough data to fill in the pipeline.

5.2.2 Performance of register based AES encryption

We shall now look at some primitive performance metrics of our encryption implementation. We will also show the performance compared to the software based version.

	Time (µ secs)
fpga_wrt_appif_val	0.37
fpga_rd_appif_val	1.49

Table 2. Timings of communication primitives.

	Time (µ secs)
FPGA Encryption time	3.74
FPGA Decryption time	3.79

Table 3. Timings of encryption and decryption modules when operated on 16 bytes of data.

Table 2 lists the timings of the communication primitives that are used in the FPGA implementation. We observe that the function `fpga_rd_appif_val` is approximately 5 times slower than `fpga_wrt_appif_val`. Since both encryption and decryption involve two function calls for writing and reading registers, their timings in Table 3 can be easily explained in terms of latency of primitive operations.

5.3 Improving FPGA performance

As the numbers in Table 3 indicate, it would take around 4 microseconds to process 16 bytes of data. That translates to a throughput of about 4MB/sec. The communication overhead further decreases this to 3MB/sec as shown previously in Fig 2, we see this by examining the slope of the graph.

The drawback of the register based FPGA implementation is the high overhead of function calls. This can be resolved if we tackle the problem by using a pull model. In the pull model, the FPGA takes the initiative to pull the raw data from the source, process it, and finally store the results in the destination.

To implement this, we have used the API call `fpga_set_ftrmem` [9] to allocate shared memory space for communication between the FPGA and CPU. On the FPGA, we need to design an I/O subsystem that is capable of communicating with the FPGA transfer region (ftrmem) in the host. The following sections explain the issues that are involved in the design of the I/O subsystem.

5.3.1 Motivation for I/O subsystem

The I/O subsystem is motivated by I/O bound applications where data needs to be streamed continuously, for example, encryption and compression. These types of applications are also compute intensive. The reconfigurability of FPGAs facilitates pipe-lining, which enables the compute engine to consume data as fast as the I/O subsystem can supply. As a result the bottleneck lies with I/O.

Any custom core needs to interface with the outside world through the RTClient on the XD1. Currently, the Cray XD1 is supplied with a interface for communicating with the Rapid Array Processor (RAP). The I/O subsystem of the XD1 relies on hyper transport [11] technology. One important advantage of hyper transport is "burst communication" mode where a maximum of 8 quad-words (eight 64-bit words) can be transferred in a single request. Any I/O subsystem design should leverage this feature. To our knowledge, currently there is no Intellectual Property (IP) core that can be used freely by XD1 users. Our goal is to design an efficient general purpose I/O interface to the RTCore.

5.3.2 I/O subsystem design issues

Some of the issues that influenced the design of the I/O sub system are discussed next.

- Hyper transport packets have 8 quad-words in a Burst Communication (BC). BC is further augmented by the RTCore, by allowing up to 32 outstanding BC requests. Therefore, the user logic can issue 32 BC requests in consecutive clock cycles and get 256 (32 x 8) quad-words in subsequent clock cycles. Any design should make an efficient use of this facility.

- The RTCore is designed such that user logic address requests and written data, share the same pins. In the XD1 these are the *ureq* lines [10]. This constrains the design to schedule read address requests, write address requests and written data on the same lines.

- Though one can theoretically issue 32 read requests, the 32 read responses need not be in the same order as the requests were issued. The HT guarantees that data within a burst packet, that is 8 quad-words, will be in-order. The RTCore does not guarantee order between the 32 packets. The user logic is responsible for ordering the responses.

- The Opteron is based on little-endian architecture. As a result communication of data from a byte-aligned buffer will be interpreted in different manner by the FPGA since the communication order is big-endian. The API provided by Cray constrains us to take into consideration this issue while designing our cores. As an example take the function call `fpga_wrt_appif_val [9]`, which takes an unsigned long as the argument for the data to be written. If one needs to pass a character buffer as an argument, a simple cast to unsigned long is not sufficient. The bytes within the buffer need to be also reordered.

- When data is read from the RAM of the CPU, the data that arrives is not necessarily in order. To get the data back in the correct order we need to buffer the data within the FPGA. Currently the XD1 uses the Xilinx Virtex II Pro FPGA [10] as accelerator modules, which have hardcore on-chip RAM modules that can be configured as RAM for scratch-pad purposes. Typically, writes to RAM are registered, meaning the RAM controller needs to lock the memory region as the state of the bits is changed. This means that data is written one clock cycle after registering the address, but reads from RAM are usually unregistered, and data is available in the same cycle that the address is requested. However, on the BRAM modules of the Xilinx Vertex, the reads are also registered. This means that the data is available only after one clock cycle and has to be factored into the I/O subsystem design.

- The Cray XD1 Programming Guide [9] mentions that one cannot allocate more than 2MB in each call to the function `fpga_set_ftrmem`. This call is used to allocate space in an application buffer that the FPGA can then access. During our development we found that a single process cannot call that function more than twice. The reason is the XD1 is still considered a beta level system, and the system software and firmware are not throughly tested. As a result, for the time being, we are constrained to 2MB of memory space for our applications.

5.3.3 Overview I/O subsystem Design

In this section we give an overview of the I/O subsystem design. The main modules of the design and their interactions with each other are described.

- **Ftrmem:** Due to the constrains of the current API implementation, one cannot allocate more than 2MB of an FPGA transfer region (ftrmem). This 2MB is partitioned into 1MB each for loading the raw data by CPU and storing processed data by FPGA.

- **Control Registers:** On FPGA fabric we have five 64-bit registers configured for control and status reporting purposes. These registers are used for communicating with the FPGA. The virtual addresses of the data source and destination for the processed data needs to be communicated to the FPGA. These registers are used to trigger the FPGA logic.

- **BRAM:** As mentioned in the previous section, the user logic can issue up to 32 outstanding requests, but the packets corresponding to the requests do not necessarily arrive in order. Therefore, we need a scratch-pad to store the packets in order. The BRAM is made up of 32 x 64-byte entries. The BRAM is dual-ported for concurrent reading and writing. Previously, we described some issues with registered reads and our approach, another feature of BRAM is that the data widths of read port and write port are independent of each other. Therefore, it is possible to configure one BRAM with a 64-bit write port and a 128-bit read port. This fact can be exploited in the case of AES encryption [1] where 128-bits are processed at a time.

- **Semaphore:** The BRAM acts as a store where data from the CPU's RAM, is written and is read by user-logic on the FPGA fabric. Therefore, we have a producer-consumer synchronization problem which we solve using a 32-bit binary semaphore. Each bit of semaphore acts as a guard for each entry within the BRAM. This BRAM with 32 entries, is analogous to a 2KB cache with 32 cache-lines of 64 bytes each. The semaphore bits are analogous to valid bits for a cache line.

- **Source-tag to Address mapping:** When the FPGA fabric issues a read request, it gets a 5-bit source tag. Finally, when the response arrives, it has a source tag associated with it, but the I/O subsystem needs to know the BRAM address where the data has to be stored. Source-tag to address mapping provides this service.

- **Read module:** This module, as its name signifies, reads the raw data from the RAM into the BRAM. It synchronizes with the write module using semaphores. When the read module issues a read request for an address, it gets a source tag. This source tag is matched with the 8 least significant bits of the read address, then the source-tag and address map is updated. When the response arrives with the source tag, it is used to lookup the destination address in the BRAM.

- **Write module:** This module is responsible for reading data from BRAM and passing it to user logic. It is also responsible for writing the final processed data back to destination buffer in CPU's RAM.

Next we describe how these modules are used in I/O subsystem using pseudo code of the CPU and FPGA actions Figs 6 and 7 respectively. The CPU simply copies the raw-data into one half of ftrmem, updates FPGA registers and finally triggers FPGA. The FPGA meanwhile is waiting for the signal from CPU. As soon as the FPGA gets the trigger, the read module starts reading from RAM. When 8 quad-words are read, the write module is activated, which reads data from BRAM and passes it to user logic for processing. When the processed data is ready, the RAM write module issues write requests to the RTCore. During the operation the read and write modules always synchronize among themselves using the semaphore bits.

```
copy raw data into src_addr
fpga_wrt_appif_val src_addr
fpga_wrt_appif_val src_len

/*writing dest_addr triggers fpga */
fpga_wrt_appif_val dest_addr

done = 0
while (done = 0)
  done ← fpga_rd_appif_val
end while

copy data from dest_addr
```

Fig 6. Pseudo code for CPU operation.

```
Wait until trigger
READ MODULE:
while (read_cntr < src_len)
  if (semaphore (i) == 0)
     issue read request to RTCore
     update src_tag to addr map
     write 8 quad-words to BRAM
     read_cntr += 8
     semaphore(i) = 1
     i++;
   end if
end while

WRITE MODULE:
while (wrt_len < src_len)
  if (semaphore(i) == 1)
     read 8 quad-words from BRAM
     pass data to user logic
     wrt_len += 8
     semaphore(i) = 0
     i++;
  end if
end while

RAM WRITE MODULE:
when user logic is ready to write
issue write requests to RTCore
```

Fig 7. Pseudo code for FPGA operation.

5.3.4 Preliminary results

We have implemented a simple user logic on FPGA that uses the I/O subsystem. The essential operation of user logic involves reading raw data from a source buffer and copying it back to a destination buffer. It essentially simulates what an ftrmem based AES implementation would do.

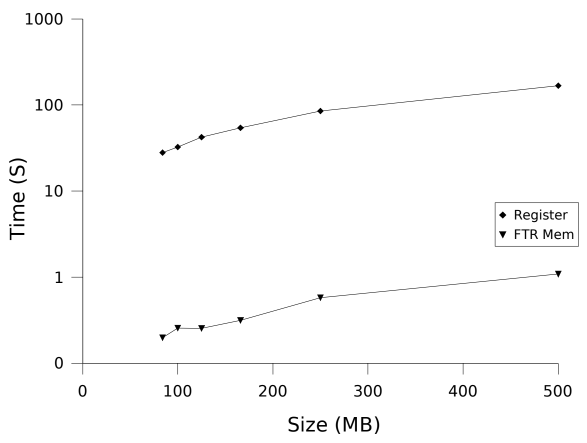

Fig 8. Time to write and read data.

Fig. 8 plots the copy time (logarithmic scale) versus size of data. The two curves shown are register based and ftrmem based implementations. The latency for the ftrmem based implementation are two orders of magnitude smaller than the register based ones. Since register based is comparable to the software-based implementation, as shown in Fig 4, we can safely project that an ftrmem based AES implementation would be approximately about two orders of magnitude lower latency than the software approach. This work is still in progress, but the preliminary results are encouraging.

Frequency	199 MHz
Bandwidth	796 MB

Table 4. Performance readings of ftrmem

We have also conducted another experiment to find the native throughput of ftrmem, without any overhead. We were able to get sustained bandwidth of around 800MB/sec. This result was obtained by timing the copy operation of 1MB, which is size of the source ftrmem, as described in Section 5.3.3. The theoretical maximum bandwidth is 1.6 GB/sec. We are able to achieve 50% one-way of the rated bandwidth. There are three main reasons for this. One, as discussed in Section 5.3.2, there is only one bus that has to be shared for read requests and writes. Secondly, we have a pace counter that adds 3 cycles of overhead for every switch between read and write states. The other main reason is that every time the FPGA requests a memory operation, the DMA controller must arbitrate whether to grant the request or not. Despite the overheads we are able to achieve 50% of the one-way maximum bandwidth.

From Fig 8, it is clear that the throughput of ftrmem based method is 500 MB/sec. This value was below the sustainable native bandwidth of about 800 MB/sec. When data is larger than 1MB, there are two copy operations involved. One copy from raw data source to ftrmem, and another copy of processed data from ftrmem to final destination. These copy operations are in the critical path of execution, therefore utilization further drops. There are two ways to overcome this limitation. One is to remove the bug that prevents the programmer from making multiple calls to `fpga_set_ftrmem`. The other is to increase the size of allocated ftrmem.

6. Future Work

Future work includes extending the resource management system to the proposed client server model. We also plan to implement more FPGA services that user programs may take advantage of, such as compression.

Specific to encryption we plan to increase the efficiency of our encryption algorithm by preprocessing and saving some key independent computations. This is commonly referred to as the "ram based" approach. Currently, all computations are computed on the fly on the FPGA.

In order to more efficiently encrypt larger data sets, greater than 16B, we are planning to implement a version which takes advantage of the FPGA Transfer Region of memory which was discussed in section 5.3.3. The rate at which we can encrypt data on the whole, from the resource managers point of view will greatly benefit from this enhancement as well. Another way to benefit the performance of an ftrmem based implementation would be to remove the overhead of the pace counter mentioned in 5.3.4.

7. Conclusion

We have shown in this paper the performance, drawbacks, and possible improvements we can make concerning FPGA based encryption. We have also introduced the preliminary version of an FPGA resource management system which will aid in making FPGA resources easily accessible to user applications with no knowledge of the actual FPGA itself.

8. References

[1] J. Daemen, V. Rijmen, AES Proposal: Rijndael, Ver 2, 1999.

[2] E. Swankoski, R. Brooks, V. Narayanan, M. Kandemir, M. Irwin, A Parallel Architecture for
Secure FPGA Symmetric Encryption, In
Proceedings of the 18th International Parallel and

Distributed Processing Symposium, 2004.

[3] The OpenSSL Project, OpenSSL: The Open Source toolkit for SSL/TLS , http://www.openssl.org, 2005.

[4] Cray Inc., Cray XD1 System Overview, Ver 1.1, 2004.

[5] The Message Passing Interface (MPI) Standard, http://www-unix.mcs.anl.gov/mpi/, 2005.

[6] Dennis Dalessandro, Why use RDMA?, http://www.osc.edu/~/dennis/rdma/rdma.html, 2004.

[7] Aldec Inc., Riviera Overview, http://www.aldec.com/products/riviera/, 2005.

[8] Xilinx Inc., XST User Guide. http://www.xilinx.com/support/sw_manuals/xilinx6/download/xst.zip

[9] Cray Inc., Cray XD1 Programming, Ver 1.2, 2005.
[10] Cray Inc., Cray XD1 FPGA Development, Ver 1.1, 2004.

[11] Open Cores, http://www.opencores.org.

[12] HyperTransport Consortium, http://www.hypertransport.org