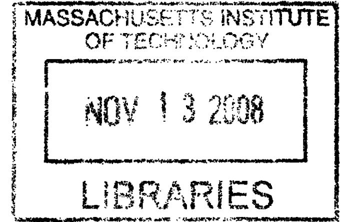
Hardware Acceleration of NetApp Virtual Tape Library Deduplication Algorithm



by

Amy Ancilla Wibowo

S.B. EECS, M.I.T., 2006

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2008

© Massachusetts Institute of Technology 2008. All rights reserved.

Author . . . .
Department of Electrical Engineering and Computer Science
August 20, 2008

Certified by . . . .
Scott Westbrook
Senior Engineer, NetApp Inc.
VI-A Company Thesis Supervisor

Certified by . . . .
Christopher J. Terman
Senior Lecturer, EECS
M.I.T. Thesis Supervisor

Accepted by . . . .
Arthur C. Smith
Chairman, Department Committee on Graduate Students

ARCHIVES

Hardware Acceleration of NetApp Virtual Tape Library Deduplication Algorithm

by

Amy Ancilla Wibowo

Submitted to the Department of Electrical Engineering and Computer Science
on August 20, 2008, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

In this thesis, I implemented an FPGA version of an algorithm used in the deduplication of backup storage data, to compare its performance to a software implementation of the algorithm. This algorithm flags sections of data with identifiers called "anchors". "Anchors" can be compared across versions of backup data, and if "anchors" are identical, sections of data are unchanged with high probability and might not have to be re-written to the backup disk. I also designed and implemented the environment by which this algorithm would communicate with a host computer. This environment can also easily be adapted for the acceleration of other shifting window algorithms. I tested the algorithm in simulation to verify its results as well as to estimate the speed at which the algorithm could process data.

VI-A Company Thesis Supervisor: Scott Westbrook
Title: Senior Engineer, NetApp Inc.

M.I.T. Thesis Supervisor: Christopher J. Terman
Title: Senior Lecturer, EECS

Acknowledgments

I would like to thank my VI-A company advisor, Scott Westbrook, for being so supportive and encouraging in this project. I could not have asked for a better mentor. I am grateful to my MIT thesis advisor, Prof. Chris Terman, for reading the drafts of this document and providing guidance and suggestions. Steve Miller, of NetApp, Inc. and Prof. Anantha Chandrakasan are dear to me for getting me started on my first digital logic design projects. Last but certainly not least, I want to thank my family for always rooting for me.

Contents

1 Introduction **13**

1.1 Motivation 13

1.2 Deduplication Algorithms 14

1.2.1 File Level Deduplication 14

1.2.2 Byte Level Deduplication 15

1.2.3 Block Level Deduplication 15

1.3 Hardware Acceleration 16

1.4 Thesis Overview 18

2 Anchor Detect Algorithm **19**

3 Algorithm Framework **23**

3.1 Existing FPGA Design 24

3.1.1 General Communication Between FPGA and Host 24

3.1.2 Communication Between Card Modules 25

3.1.3 Transferring Data from Host to Card 25

3.1.4 Transferring Data from Card to Host 27

3.1.5 Changes Needed to Support an Algorithm Module 28

3.2 Design Decisions 28

3.2.1 Data Transfer from Host to Card 28

3.2.2 Data Input to Algorithm 29

3.2.3 Reusability 30

3.3 Specifications for New Modules 30

3.3.1 Algorithm Data Requestor 30
3.3.2 Algorithm Address Counter 30
3.3.3 Algorithm Buffer 31
3.3.4 Tag Checker and Algorithm Router 32
3.3.5 Anchor Write Request module 32
3.3.6 Algorithm Module 32
3.4 Modifications to Existing Modules 33
3.4.1 Tx Format Module 33
3.4.2 Memory Arbiter 33
3.5 Potential Bottlenecks 33
3.5.1 Data transfer from host to FPGA 33
3.5.2 Data transfer from FPGA memory to algorithm buffer 35
3.5.3 Algorithm 35
3.6 Implementation Details 35

4 System in Action 37
4.1 Transferring data from host to card 37
4.2 Transferring data from local memory to algorithm buffer 37
4.3 Algorithm Buffers 40
4.4 Running the Algorithm 40

5 Simulation and Testing 41
5.1 Simulation Environment 41
5.2 Test File 42
5.2.1 Testing the Algorithm Infrastructure 43
5.2.2 Testing the Algorithm Logic 44
5.3 Performance Measurements 44

6 Conclusions and Further Thoughts 45
6.1 Performance Analysis 45
6.2 Possible Improvements 45

6.2.1 Parallelization 45
6.2.2 Host to card data transfer 46
6.3 Final Thoughts 46

List of Figures

1-1 Virtual Tape Library backup system. 14
1-2 Deduplication responsibilities of host and FPGA. 17

2-1 Anchors across two versions of backup data, newest backup on top. . 20
2-2 Block diagram of the Anchor Detect Algorithm logic. 22

3-1 Diagram of the PCI-E card design. 24
3-2 The process of the card reading host memory. 26
3-3 The process of the host reading card memory. 27
3-4 Dependencies of the two Algorithm Buffers. 31
3-5 Overview of algorithm framework. 34

4-1 Transfer of data from local memory to algorithm. 39

5-1 Block diagram of the simulation environment. 42

Chapter 1

Introduction

1.1 Motivation

Information is valuable. Digital information is susceptible to loss or corruption from disk failures, application failures, viruses, or natural disasters, among other causes. Effort has been made to develop more reliable and fault-tolerant data storage systems, systems that are either less likely to lose data or make lost data easier to recover. The most reliable storage systems come at a high price, and still can't guarantee that data will never be lost. For this reason data backups still exist as an important last precaution against data loss. In large organizations or companies, these backups are performed systematically and often. Because of the huge volume of data and the small chance of need for retrieval, backups have been traditionally stored on low-cost tape libraries. As disk space gets cheaper, however, there has been a trend towards moving to disk backup libraries. Disk libraries have faster backup time as well as retrieval time, and are more reliable than tape libraries. Particularly, virtual tape libraries have become popular because these behave just like traditional tape libraries from the point of view of the system performing the backup, interfacing with existing tape library backup software.

A traditional backup library stores many copies of the same data or similar data. To take advantage of data redundancy, sections of the latest backup data that have not changed since the previous backup can be replaced with pointers to the identical

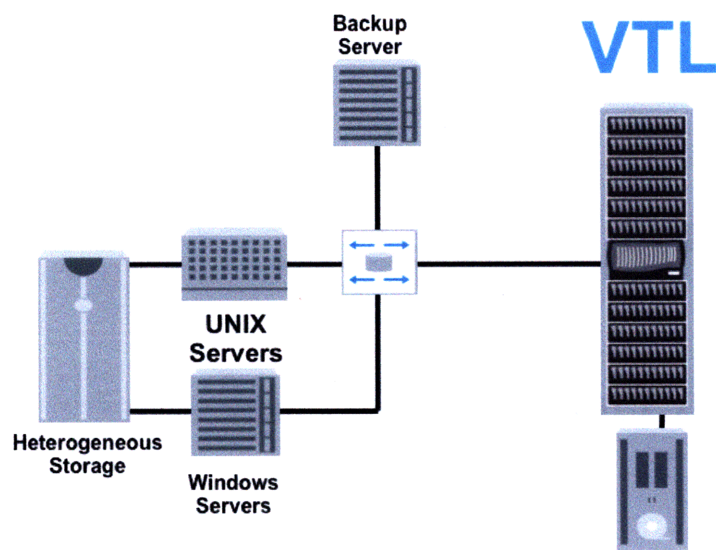


Figure 1-1: Virtual Tape Library backup system.

section from the last backup, while only new chunks of data are saved. This is referred to as deduplication, and NetApp, Inc.[1] has developed and patented its own method of deduplication for its virtual tape library backup system. Currently deduplication is performed in software on data after it has been backed up. But due to the nature of the deduplication logic, sections of it might be well suited for hardware implementation on a Field Programmable Gate Array (FPGA). If the hardware implementation runs fast enough, data could be deduplicated as it comes into the virtual tape library, instead of as an after-process. This would significantly reduce the overall time needed to back up data.

1.2 Deduplication Algorithms

1.2.1 File Level Deduplication

Deduplication can occur at a file level or at a lower level. Deduplication at a file level means that two files have to be exactly the same for one to be deduplicated. In a

[1] company name changed from Network Appliance, Inc. in March of 2008

backup file environment, it is likely for files to be similar, not identical, since often, only small changes have been made to files. It is possible to save only those small changes with deduplication of a smaller grain.

1.2.2 Byte Level Deduplication

Deduplication at the byte level is essentially compression. Byte level deduplication does not make use of the fact that two versions of backup data tend to have large areas of similarity.

1.2.3 Block Level Deduplication

Block level deduplication examines data in sections smaller than a whole file but larger than a byte. Most block level deduplication algorithms have 3 basic stages: chunking, fingerprinting, and duplicate identification [1]. In chunking, the data from a new backup is split into blocks that can be compared to blocks in the existing set of backup data. Both the offset and the size of the chunks can affect how many duplicate chunks are found between the new backup and existing backup data. Thus, the blocks may be of varying sizes, and different algorithms can be used to try to split up data into chunks in a way that allows for the recognition of as many duplicate blocks as possible. One algorithm that helps to split data into blocks is NetApp, Inc.'s Anchor Detect Algorithm, which is discussed in more detail in Chapter 2. The hardware implementation of the Anchor Detect Algorithm is the focus of this thesis.

In fingerprinting, the data in each chunk is given a unique (or almost unique) identifier by a hashing digest algorithm, such as SHA1 or MD5, or by a combination of a few hashing algorithms. Lastly, in duplicate identification, the blocks from this new backup are compared to the previous blocks of backup data by their fingerprints. Computing the fingerprints and using them to compare two blocks of data is faster than comparing the two blocks bit by bit.

Because hash collisions are possible, a fingerprint is not guaranteed to be unique; that is, it is conceivable for two non-identical blocks to result in the same fingerprint,

and for a block of data to be removed when it is not actually a duplicate. However, the chances of this happening are very small, 1 in 2^{160} with the SHA1 hash algorithm. This chance is even smaller if hashes are combined. Additionally, to avoid risk, the hash could be merely used to identify blocks to be compared bit by bit, which would still be less computationally expensive than comparing all blocks bit by bit. Whatever the method of duplicate identification, if two blocks are found to be identical, the recent one is replaced with a pointer to the earlier. If the two blocks are not found to be identical, both blocks are preserved.

1.3 Hardware Acceleration

Often, hardware is constructed to perform a specific computation, like the graphics card inside a PC. Whereas a processor would have to execute several general commands to perform a graphics operation, the graphics-specific chip can be designed to perform the graphics computation in one command, thereby reducing computation time. This approach is called Application Specific Computing. Manufacturing a special chip such as an ASIC (Application Specific Integrated Circuit) is an expensive option for someone who wants to accelerate a very specific algorithm on just one or a few computers. It is also costly to re-manufacture a chip to make changes or corrections to the logic, and so is not ideal for newer algorithms that are still evolving. Increasingly, Field Programmable Gate Arrays (FPGAs) are instead being used to implement algorithm-specific hardware. Since FPGAs can be reconfigured to perform different logic on the fly, it makes them a practical solution, with still significant speed up.

For a recent example, an FPGA system has been used to accelerate an algorithm for scanning protein sequences (Smith-Waterman algorithm) approximately 170 times the speed of the algorithm on a desktop computer [2]. Also, an image processing algorithm written to an FPGA on a Celoxica board-based development platform was shown to have the same performance as a graphics card [3].

The Field Programmable Gate Array (FPGA) is a chip with many identical logic

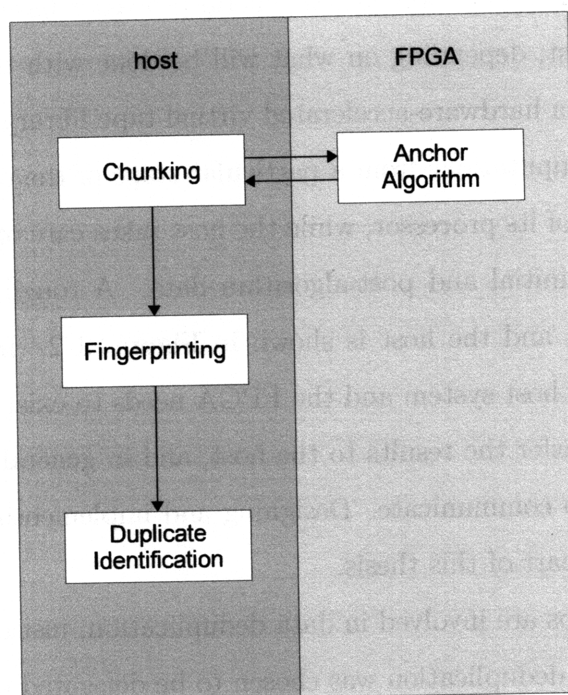


Figure 1-2: Deduplication responsibilities of host and FPGA.

blocks, which are all laid out into a larger array. Each logic block contains several basic logic elements, the exact types differing depending on the type and manufacturer of the FPGA. By feeding the FPGA a bitstream, one can change which logic elements are connected or disconnected. By the connecting certain paths and not others, the FPGA can be used to perform different logic. But when programming an FPGA, one does not have to think of the design in terms of which logic elements should be connected at the lower level; one can program the FPGA behaviorally at a higher level through a description language such as VHDL or Verilog. The existing FPGA design tools take care of converting behaviors into a bitstream that will connect the appropriate logic elements. The design tools will also optimize the logic design and selection of the logic elements such that the result is the most efficient possible, given what is available on the specific FPGA. In this way an algorithm can be configured into hardware using an FPGA.

In order to accelerate an algorithm, an FPGA can be used standing alone, or the

FPGA can be used in conjunction with a general processor-based computing system, referred to as the host, depending on what will be done with the algorithm results. This thesis describes a hardware-accelerated virtual tape library system in which the host can delegate computations from a particular stage of the deduplication process to an FPGA instead of its processor, while the host takes care of other computations and management of initial and post-algorithm data. A rough outline of the roles played by the FPGA and the host is shown in Figure 1-2. In such a system, an interface between the host system and the FPGA needs to exist to transfer the data to the FPGA, to transfer the results to the host, and in general to allow the FPGA and host computer to communicate. Designing and implementing such an interface is also an important part of this thesis.

Because many steps are involved in data deduplication, just one sub-algorithm of the chunking stage of deduplication was chosen to be delegated to the FPGA. In the future, more deduplication sub-algorithms may be implemented on the FPGA of this system.

1.4 Thesis Overview

Chapter 2 describes in detail the Anchor Detect Algorithm, the algorithm chosen to be computed on the FPGA.

Chapter 3 describes the interface by which the host system sends data to the FPGA, starts the FPGA, and gets the results back. Much of this interface is based on a PCI-E card victim cache design by NetApp, Inc.; new and modified modules are identified as such. Design decisions, tradeoffs, and possible bottlenecks are explored.

Chapter 4 describes how the system behaves in action.

Chapter 5 describes the simulation and testing environment, based on the testing environment for the PCI-E card victim cache design. It also lists the types of tests created to verify the results of the Anchor Detect Algorithm.

Chapter 6 contains performance analysis results, conclusions, and ideas for further work.

Chapter 2

Anchor Detect Algorithm

The portion of the deduplication algorithm chosen to be implemented on the FPGA is NetApp, Inc.'s patented Anchor Detect Algorithm. This algorithm is used to find unique "landmark" sections, also called "anchors", in a version of backup data during the chunking stage of data deduplication. The algorithm computes a hash on a 4093 byte sized section of incoming data, and if the hash of the current section of data fits some criteria, the section is marked as a "landmark". Each "landmark" is then given a unique identifier. The algorithm is essentially a rolling hash algorithm, meaning that to compute its next hash, it shifts, or in other words, it accumulates the effect of the most recent incoming byte while canceling out the effect of the byte exiting the "window frame" of the section size, the byte that came 4093 bytes before the incoming one. There are no bytes to exit the window frame until after the first 4093 bytes. The criteria that qualifies a section of data to be classified as a landmark is whether or not certain bits in its hash are ones or zeros. Currently 14 bits are inspected, to yield on average, 1 landmark per 16kB. The location and number of the considered bits were randomly generated, then tweaked, and shown in practice by NetApp to yield useful spacing of landmark points. Attempts to generate better results with different bits are still ongoing.

When comparing previous backup data to new backup data, their respective lists of landmark point locations and values are compared. When identical landmarks appear in both the new and previous backup data during chunking, data is split into

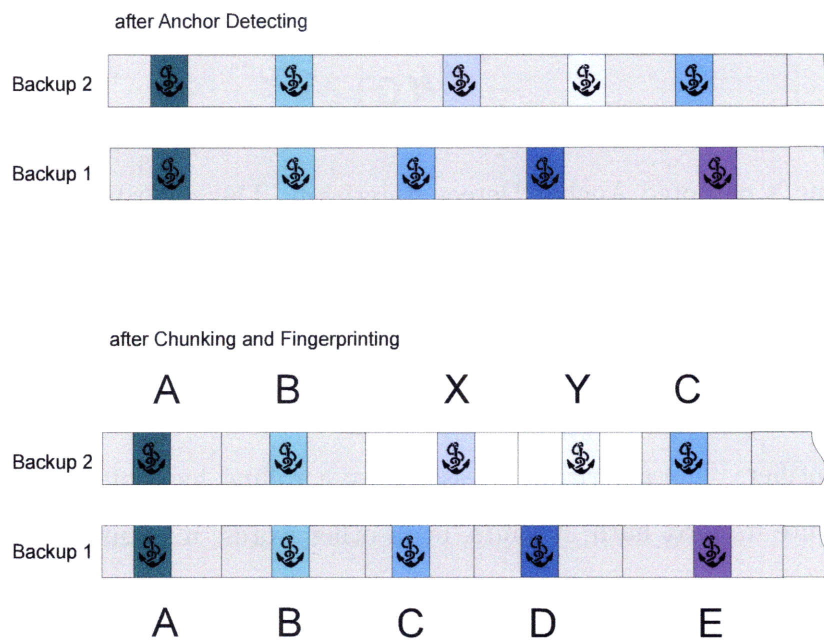


Figure 2-1: Anchors across two versions of backup data, newest backup on top.

chunks around these landmarks, generating blocks that will hopefully yield many matches. Figure 2-1 shows how anchors help in deduplication. The top diagram shows anchors found after the anchor algorithm is applied, and the bottom diagram shows the result of chunking using those anchors, and fingerprinting. Of the chunks shown from the second backup, only the white chunks X and Y need to be saved.

The algorithm computes the rolling hash for a sliding window of incoming data by running the data through 6 look up tables, XORing outputs with shifted previous results, XORing all outputs together, and comparing the masked output to a fixed register value. The values of the look up tables were also randomly generated but tweaked to yield well spaced landmark points that lead to finding duplicate chunks. A diagram depicting the anchor algorithm logic is shown in Figure 2-2.

In order to cancel the effect of the byte exiting the window, the "exit" input byte goes to a look up table whose values exactly cancel the effect of the "entering" set of lookup tables, when XOR'd with the previous results. The lookup tables, shifting, masking, and bit comparisons make the algorithm a good candidate for hardware acceleration.

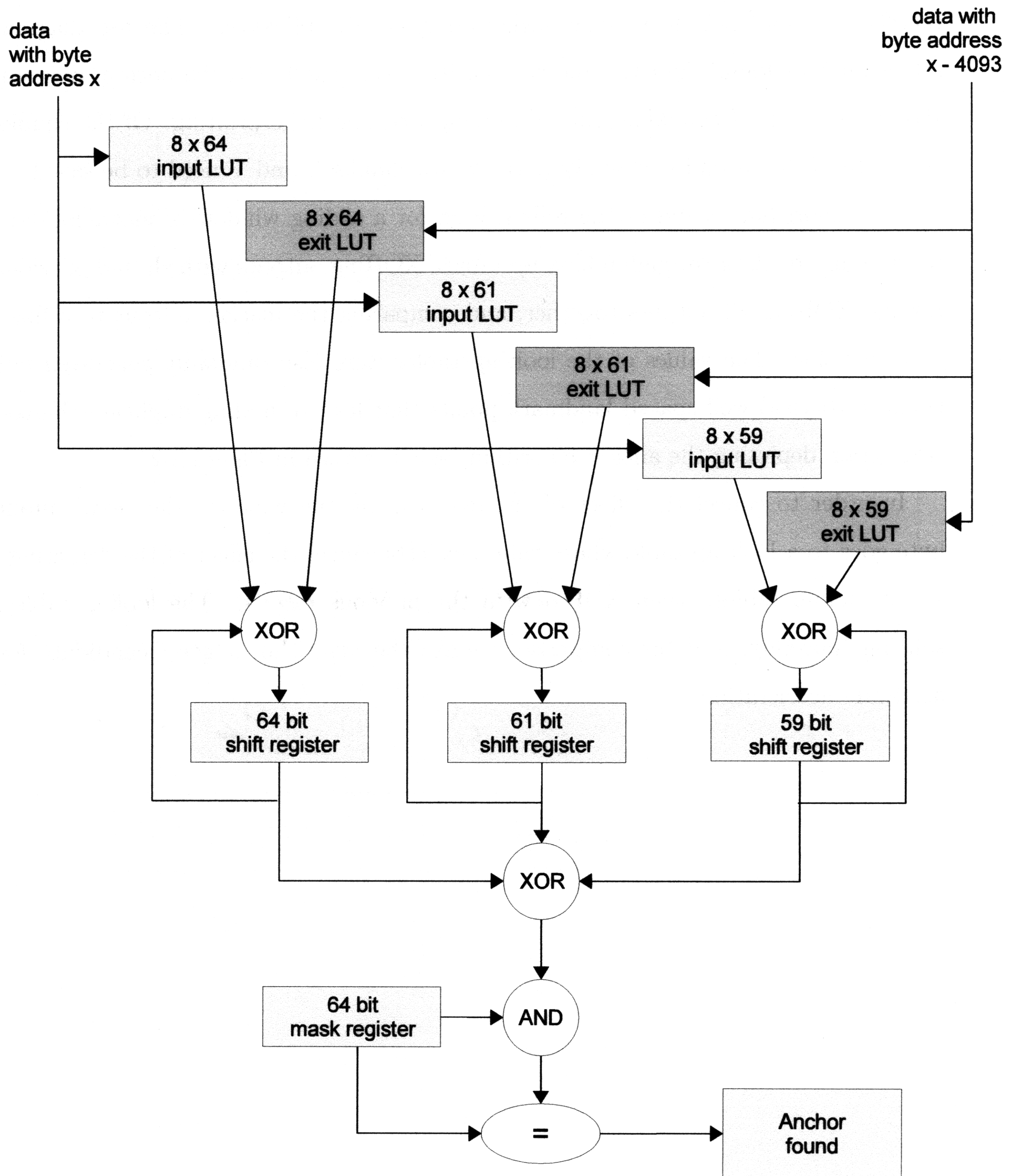


Figure 2-2: Block diagram of the Anchor Detect Algorithm logic.

Chapter 3

Algorithm Framework

This section describes the interface between the overall deduplication system and the system that will specifically compute the Anchor Detect Algorithm. In order to support the computation of the Anchor Detect Algorithm, this interface needs:

- a way to transfer data from the host system to hardware computing the algorithm
- a way to let the host control the algorithm
- a way to input data to the algorithm 2 bytes, spaced 4093 bytes from each other, at a time, 1 for entry into the sliding window and 1 for cancellation from the sliding window.
- a way to store results to be retrieved by the host at a later time.

The interface is based on a PCI-Express card based victim cache design developed by NetApp, Inc. PCI stands for Peripheral Component Interconnect, and PCI-E specifies an interface for connecting peripheral devices to a motherboard. The existing card design includes attached memory and an FPGA with a few basic memory storage and retrieval functions, such as registers which can be accessed by the host, and the ability to transfer data host CPU to the card local memory, and vice versa, via DMA (Direct Memory Access) Engines. Additional modules were added to the system to fulfill all of the above requirements, and new modules are specifically indicated as

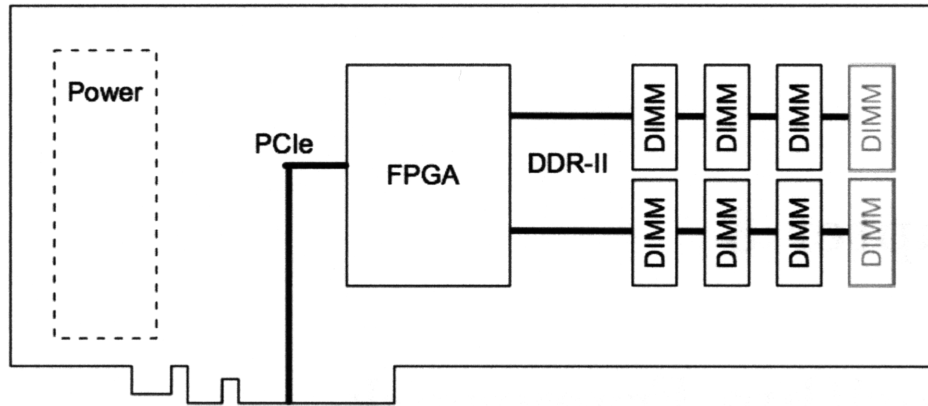


Figure 3-1: Diagram of the PCI-E card design.

such in this chapter. A block diagram of the basic PCI-E card design is shown in Figure 3-1. Figure 3-5 shows a block diagram showing important existing modules and new modules in the algorithm framework.

3.1 Existing FPGA Design

In this section, I discuss in further detail the existing FPGA modules that perform memory transfer and allow host control over functions. Understanding these modules and how they interface with each other was important in designing new modules that fit in seamlessly with the existing system.

3.1.1 General Communication Between FPGA and Host

At a low level, the host and the computer are connected by a PCI-E interface. But understanding the PCI-E format is not necessary for understanding how to initiate data transfers between the host and the card. The host can send control signals to the card by writing to Memory Mapped Registers (MMRs) on the FPGA. These registers are in the host's address space but are actually assigned to registers on the FPGA. If the host completes the sequence of register writes that make up the DMA initiation process, the DMA engines on the FPGA will carry out the memory transfer.

Other MMRs are intended for initiating interrupt packets, and the FPGA can write to these registers when it wants to grab the host's attention, in the case of an error, for example.

3.1.2 Communication Between Card Modules

Even though the PCI-E interface between the host and card is abstracted away in the design, data is transferred between some modules of the FPGA in a modified PCI-E style packet. For example, memory read/write requests given to the memory controller interface are packets that have a header in PCI-E style, with pertinent information such as the local memory address in a 128-bit data field, like a PCI-E data packet.

3.1.3 Transferring Data from Host to Card

In order to write to the card's local memory, the host writes a DMA read (downstream) descriptor to an MMR, indicating the host address to be read, the local address to write to, and the length of the data to be read. The descriptor is picked up by the DMA engine, and a distinct tag is assigned to keep track of the request. The tag manager stores the local address and length of the request in the associated tag register. The DMA engine sends a request packet to the host for the data. Once the data comes back from the host, using the tag, the data is matched with its intended local address, and the DMA engine generates and forwards to the memory arbiter a request to write this data to memory. The memory arbiter that arbitrates among all requests to read or write to local memory, giving equal priority to all DMA reads/writes to or from FPGA memory. When it finally grants the request, the arbiter passes the local address and the data to be written along to the memory controller interface. After the transaction, the tag number is free to be used again. This process is diagrammed in Figure 3-2.

The memory arbiter that arbitrates among all requests to read or write to local memory. It gives equal priority to all reads/writes to FPGA memory.

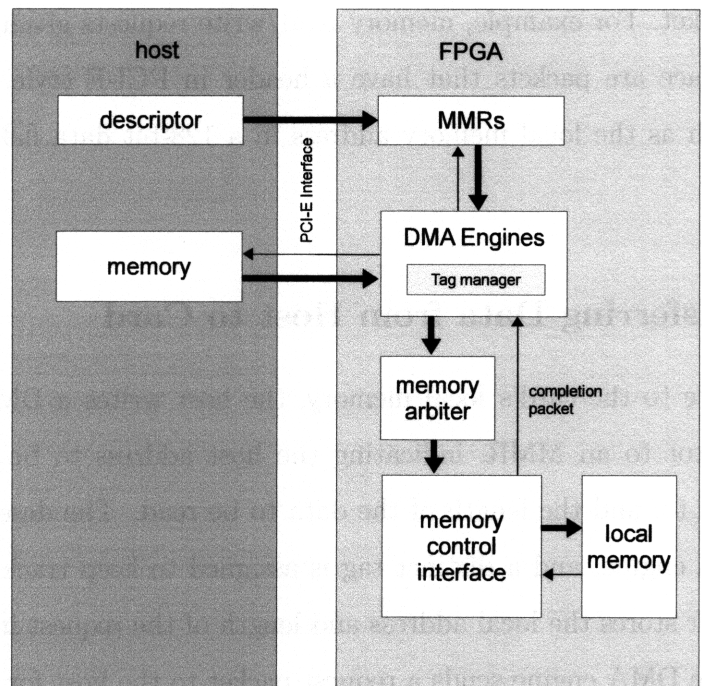


Figure 3-2: The process of the card reading host memory.

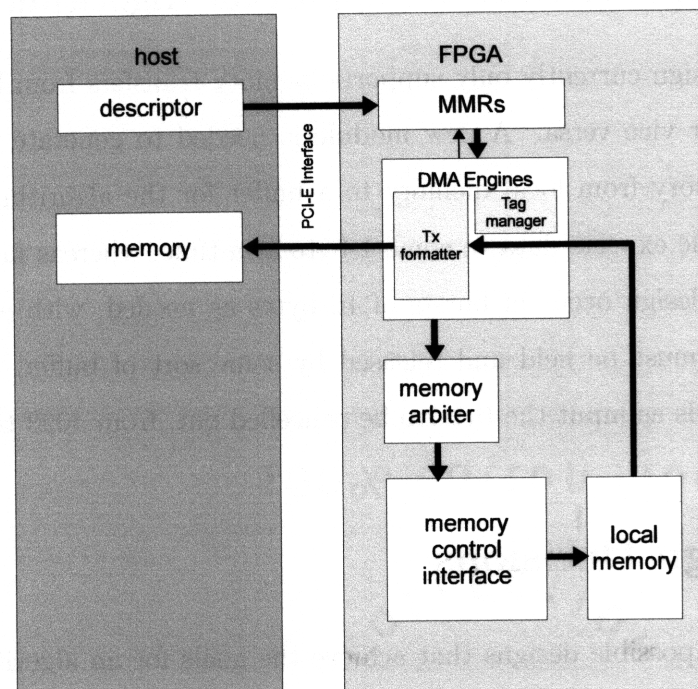


Figure 3-3: The process of the host reading card memory.

3.1.4 Transferring Data from Card to Host

In order to read the card's local memory, the host writes a DMA write (upstream) descriptor to an MMR. This descriptor indicates the local address to be read, the host address to write to, and the length of the data to be read. The descriptor is picked up by the DMA engine, and a distinct tag is assigned to keep track of this request. A separate range of tag values exist for write and read requests. The tag manager module stores the local address and length of the request in the associated tag register, to match to the packet that comes back from local memory. The request to read memory is sent to the memory arbiter where it waits its turn for access to local memory. When the request is granted, the read address goes to the memory controller and the data is retrieved. The data that comes back is matched with its tag and prepared to be sent as a write packet to the host by a module called the Tx Format module. Figure 3-3 overviews this process.

3.1.5 Changes Needed to Support an Algorithm Module

The existing design currently only supports memory transfers from host memory to local memory or vice versa. A new module is needed to generate request packets to transfer memory from local memory to a buffer for the algorithm module. The algorithm module expects data to come 1 byte at a time, whereas memory transfers in the existing design occur in bursts of 16 bytes as needed, with a minimum of 4 bursts, so data must be held and released by some sort of buffer. The algorithm module also needs as input the byte to be cancelled out, from 4093 bytes before.

3.2 Design Decisions

There are many possible designs that achieve the goals for an algorithm framework previously described. One of the most challenging aspects of the thesis was analyzing these possible designs for tradeoffs and deciding which ones to implement, keeping in mind the goals of efficiency and reusability.

3.2.1 Data Transfer from Host to Card

There are several ways to arrange the transfer of data from the host to the algorithm. The thesis implementation streams data from the host and stores it to local memory using the DMA engines. When the algorithm needs more data, data is retrieved from local memory and dropped off in a buffer that feeds into the algorithm. Having data in local memory means the data is close at hand if the algorithm needs it a second time. This might be the case, depending on the implementation of the input mechanism of the algorithm, since both a byte with offset x and a byte with offset $x - 4093$ are entering the algorithm at a time.

Another data transfer option is for data to stream from host memory to a buffer for the algorithm, skipping local memory entirely. Manufacturing boards without attached memory would be a significant reduction in cost. In terms of speed, skipping local memory replaces 2 steps— writing to FPGA memory and retrieving from FPGA

memory— with one step. However, retrieving from FPGA memory can occur alongside the transfer of data from host to card. Also it seems likely that the performance bottleneck will not be the transfer from local memory to the algorithm.

3.2.2 Data Input to Algorithm

To calculate its first result, the algorithm requires an input of 4093 bytes, one at a time; for each result thereafter, it requires an input byte and the byte that entered 4093 bytes before. This can be accomplished by a 4093 deep FIFO that pops only when full, paired with a buffer that feeds it, both providing data for the algorithm simultaneously. A FIFO generated by the Xilinx core generator can be at most 512 entries deep. According to the Virtex-4 user guide, the latency of 2 cascading fifos can be up to double that of 2 FIFOs joined together; the latency of 8 cascading FIFOs might be an efficiency problem.

Instead, the input structure for the algorithm is composed of two FIFO structures filled with the same data, but the second one delaying its output until the first one has output 4093 bytes. The inefficiency of this design is that data must be retrieved from local memory twice, but even retrieving from local memory twice should not take as many clock cycles as transferring from host to local memory.

Alternatively the FIFO could be built from a 4k deep, 1B wide block RAM, generated by the Xilinx core generator, with incrementing read and write addresses that circle back after reaching 4093. This 4k deep FIFO would supply the byte to be cancelled and be connected a shallower buffer FIFO for input (taking in 4 bytes from local memory at a time and ouptutting one), which would also supply the byte to be input to the sliding window. This design only requires data to be transferred once from local memory, and would be useful if the data could be transferred from the host to the FPGA at a faster rate.

3.2.3 Reusability

The algorithm block in the design could be replaced with any sliding window algorithm with change of the window size parameter, or any algorithm that needs a byte at a time if the 2nd buffer is removed. Very few changes would need to be made to allow any algorithm with a particular input size requirement to be swapped out with the anchor algorithm module. This is possible since the modules outside of the algorithm are not specific to an anchor detecting algorithm.

3.3 Specifications for New Modules

3.3.1 Algorithm Data Requestor

This module generates requests for data to be transfered from local memory to the algorithm data buffer. The requests are in the form of packets to the memory arbiter with the local read address, which is the lowest memory address whose data has not yet been transferred to the buffer for algorithm data. These requests read 64 bytes from local memory at a time. When the algorithm buffer has 64 bytes or more of free space, a signal is sent to the Algorithm Data Requestor. Only one request for data can be sent to the memory arbiter at a time, but up to four pending requests can be enqueued.

3.3.2 Algorithm Address Counter

This counter keeps track of the last address whose data has been written to the algorithm buffer. There is also a counter for how many addresses in local memory have been filled with host data, and transfers should only occur when there is free space in the buffer and the last local memory write address counter value is greater than the algorithm's read address counter value.

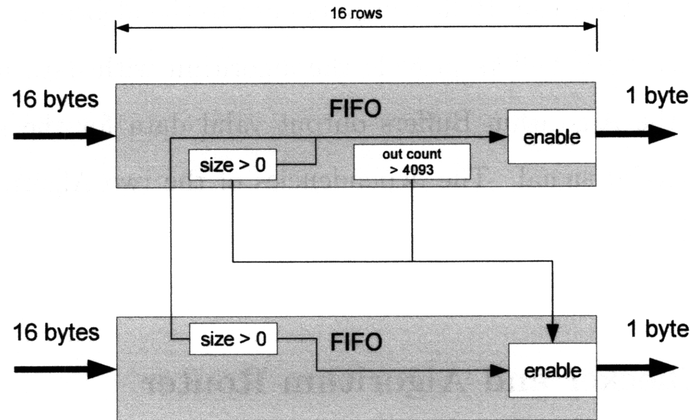


Figure 3-4: Dependencies of the two Algorithm Buffers.

3.3.3 Algorithm Buffer

The Algorithm Buffer module holds data and feeds it to the Algorithm, 1 byte per clock cycle. Such a module is necessary because with the existing memory interface, reads of FPGA memory result in 4 bursts of 16 bytes each (or more generally, multiples of 4 bursts), while the Anchor Algorithm specification requires an input of 1 byte to factor in, and 1 byte to factor out, with offset 4093 bytes earlier, at a time. Two buffers are used in the design, one holding data whose effect should be added to the algorithm and one holding data whose effect should be cancelled out.

An Algorithm Buffer module takes in 16 bytes of data at a time (coming from 1 burst), but outputs data 1 byte at a time. It consists of 16, 1 byte wide, 16 bits deep FIFOs, and simple arbitration logic to enable the smaller FIFOs such that the data is output in the original order. Thus, it holds data from 4 data requests made by the Algorithm Data Request module. Whenever there are 64 bytes of empty space in the buffer, the Algorithm Data Requestor module asks for more data for the buffer. The depth of the buffer can be varied to find the optimum depth for performance.

Both buffers keep track of the other's output count and size. For the first result,

and first 4093 bytes to enter the algorithm, no bytes need their effect cancelled out yet, so only the first Algorithm Buffer outputs data to the algorithm. For the second result and onwards, both buffers provide the algorithm with data if neither buffer is empty. When the Algorithm Buffers output valid data for the algorithm, they also assert a data_valid signal. The dependencies of the two Algorithm Buffers are diagrammed in Figure 3-4.

3.3.4 Tag Checker and Algorithm Router

This module checks packets containing read-back data as they leave local memory, to see if they are intended for the Algorithm Buffer. Packets intended for the algorithm buffer will have a special tag in their header, distinct from packets from DMA read/write tags. If this special tag is found, the next 4 packets (64 bytes) will be forwarded to the Algorithm Buffer module, and will not be passed on to the host. This module is necessary because in the original design, tags for local read data are not checked until the data is being prepared to send to the host, within the DMA Engine.

3.3.5 Anchor Write Request module

This module generates requests to write anchor data to memory once an anchor is found. It gets anchor information from the algorithm module and passes it on to the memory arbiter, along with a local address to which data will be written. The starting address for the entire set of anchor data is decided on beforehand.

3.3.6 Algorithm Module

This module computes the Anchor Algorithm and keeps track of the number of results. When an anchor is found, it forwards the offset (result number) and unmasked value of the anchor to the Anchor Write Request module.

3.4 Modifications to Existing Modules

3.4.1 Tx Format Module

The original module receives data leaving local memory, checks its tag, and constructs a write request to host memory. This module is modified for the anchor algorithm environment to recognize one more specific type of tag, one for transferring data from local memory to the algorithm buffer. When the Tx Format Module sees this special tag, the current data will be discarded instead of written to the host.

3.4.2 Memory Arbiter

More inputs are connected to the memory arbiter, since both the Algorithm Data Request module and the Anchor Write Request module need to access local memory.

3.5 Potential Bottlenecks

The three potential bottlenecks of the system are the data transfer from the host to the FPGA memory, the data transfer from the FPGA memory to the algorithm buffer, and the algorithm itself.

3.5.1 Data transfer from host to FPGA

A DMA descriptor can specify a request for a data transfer from the host to the FPGA with length any multiple of 64 bytes. But packets between the host and the FPGA can only carry 64 or 128 bytes at a time, so requests for other amounts of data have to be sent by multiple packets. It is assumed that to prepare for the algorithm, the host will make these requests constantly until local memory has all of the data. If this step is the bottleneck, the algorithm buffer will often need more data, but there will not be enough data in the FPGA to fill it. If the algorithm is started after all data is transferred from the host to local memory, this will not be a bottleneck. However, the time it takes to transfer all data from the host to local memory cannot be considered

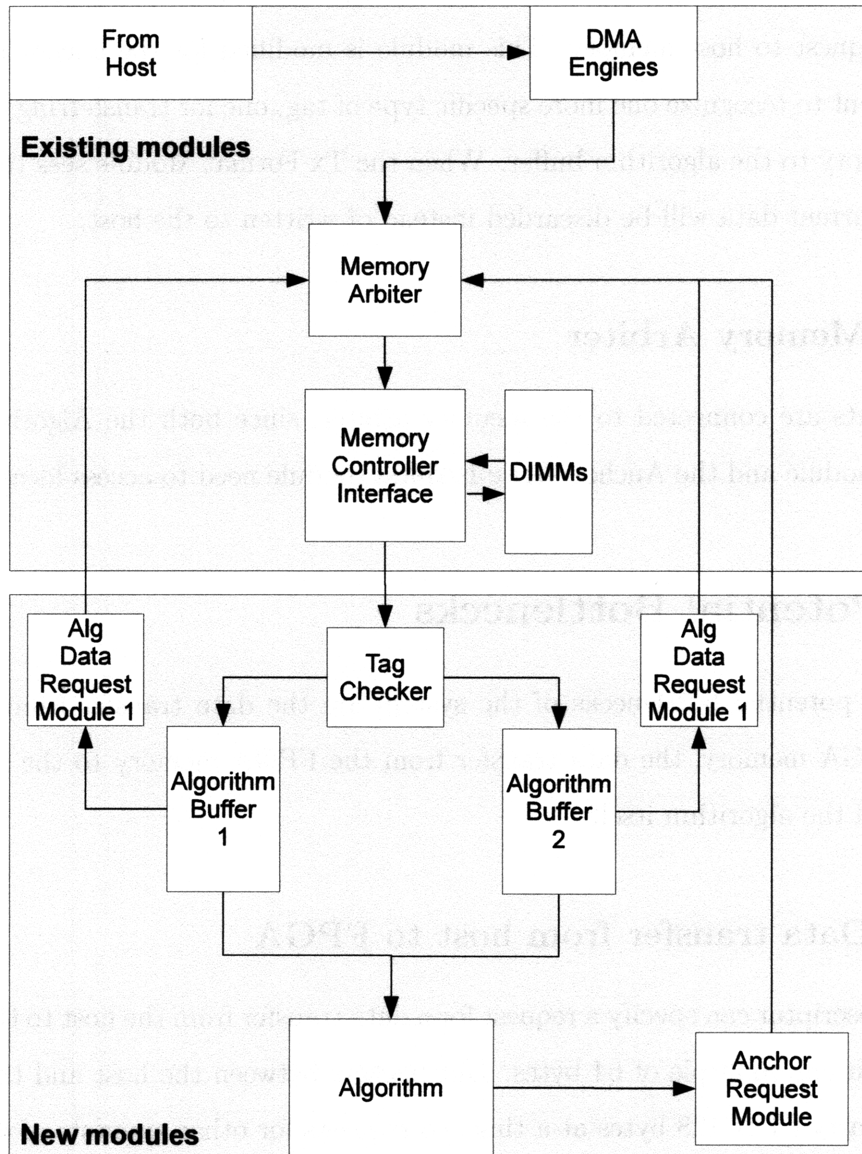


Figure 3-5: Overview of algorithm framework.

as “setup” time since it is proportional to how much data will be computed.

3.5.2 Data transfer from FPGA memory to algorithm buffer

The algorithm buffer fills up with data from local memory, and requests more data as the Algorithm Buffer empties. Requests can be made for 64 bytes of data at a time. Each request needs to go through the memory arbiter, indicating that being able to request more than 64 bytes of data at a time could be more efficient. However if the requests are larger, the buffer needs to be more empty before requesting more data, and could risk running out of data before new data arrives. If requests are made larger, the buffer size could be increased as well, to lessen the chance that data runs out.

3.5.3 Algorithm

Each step of the anchor algorithm depends on the result of the step before it, and this means that the processing of the current byte depends on the result for the previous byte. As such, the algorithm cannot be pipelined in a simple way, and the thesis implementation of it was straightforward. This means that the algorithm is limited by the clock rate for a PCI-E x4 design, 125 MHz, and can at maximum process 125 MB per second. The restrictions on the clock speed are necessary because the Virtex-4 FX60 cannot hold a PCI-E core larger than x4, and because the PCI-E x4 core cannot be placed and routed with a clock of faster than 125 MHz. More complicated parallelization could have been attempted and is described in Chapter 6.

3.6 Implementation Details

The anchor detect algorithm was written in the Verilog hardware description language targeted for a Xilinx Virtex-4 FX60 FPGA, and Synplicity tools were used to simulate the logic. The design is based on the assumption that the FPGA will be located on a PCI-E based card, so the host and FPGA communicate on a low level through

the PCI-E protocol. The lookup tables of the algorithm were implemented as 8 bit input, 64 bit output block ROMs, generated by Xilinx Core Generator tools. The initialization files for the "cancelling out" lookup tables were generated by a Java program that rotated, shifted, and XOR'd the "input" lookup table values the appropriate amounts. Implementing a driver for the PCI-E card was beyond the scope of this thesis.

Chapter 4

System in Action

This section describes in detail how the existing modules and new modules work together to find anchor points in data.

4.1 Transferring data from host to card

As a first step, the card performing the Anchor Detect Algorithm needs to be initialized like any PCI-E device. This means telling the card which addresses in the host's memory space will correspond to which registers on the card. Next the data in which to find anchor points must be transferred from the host to the FPGA. The host will write DMA Read descriptors indicating the location in host memory where this data is stored and where in local memory the data should go. These descriptors will be processed and data will be written to FPGA memory.

4.2 Transferring data from local memory to algorithm buffer

Even before the algorithm starts, the Algorithm Data Request module will request data until the algorithm buffer is full. Subsequently, whenever the Algorithm Buffer has 64 bytes or more free, and when 64 bytes or more are available in the card's

local memory, a request is made for data to be transferred from local memory to the Algorithm Buffer to be ready for processing by the algorithm. This makes sure the Algorithm module has data for processing as soon as possible.

All requests to access FPGA memory, whether from the algorithm module or from the host, must go through the memory arbiter. In the case of a memory read request, the memory arbiter needs to know the local address to be read from and how much data to read. This information is sent in a packet, in a format similar to a PCI-E packet, generated by the Algorithm Data Request Module. The packet also contains a tag field, which is distinct for each type of transaction. If a transfer is being made from local memory to the algorithm buffer, the tag will be recognized by other modules as specific for memory-to-algorithm data transactions.

When the memory arbiter grants the algorithm's request to read local memory, the request packet gets sent to the memory control modules. Eventually it gets to the DDR2 controller, which interacts directly with the DIMMs (dual in-line memory modules) on the card. The data at the given address is read from memory. After error correction decoding, it gets organized into a packet with header and data by the DDR2 interface. The memory interfaces cannot distinguish it from read data intended to be sent to the host, so it gets sent to the Tx format module of the DMA Engine. This module normally forwards packets of data read from FPGA memory to the host. But it was modified to recognize the special tag of algorithm-related transactions and will not pass these packets on to the host.

The read-back data will also be forwarded to a new module, which checks tags from header packets coming out of the memory control modules. When tags are checked and the special tag for transferring data to the algorithm is found, the next four packets are forwarded into the algorithm buffer. This is because data read from the FPGA memory and intended for the algorithm is always returned in four bursts of 16 bytes each. This process is illustrated in Figure 4-1.

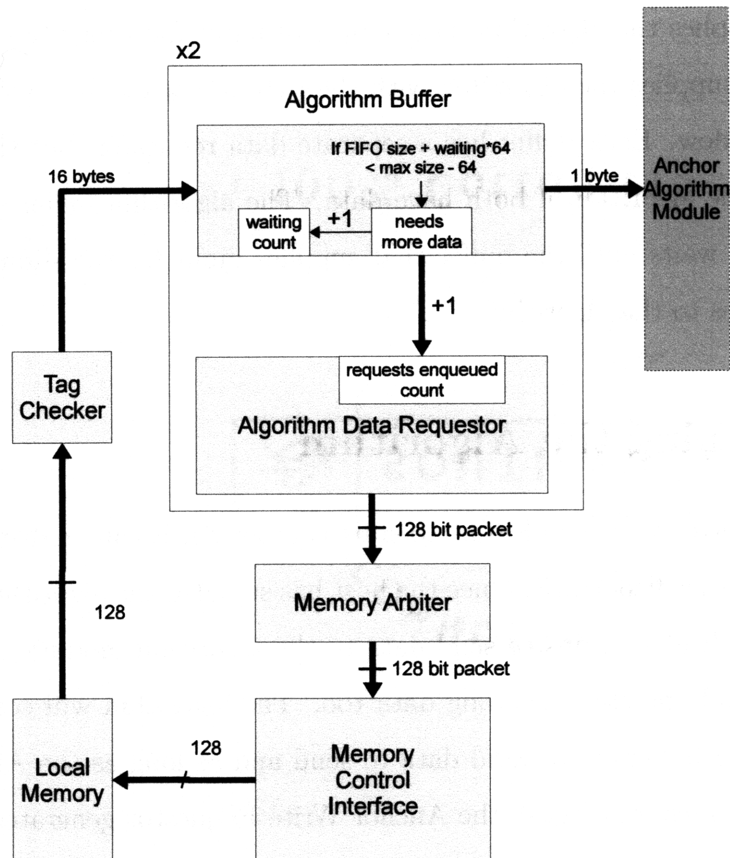


Figure 4-1: Transfer of data from local memory to algorithm.

4.3 Algorithm Buffers

After data is read from local memory, it waits in an Algorithm Buffer. There are 2 Algorithm Buffers each consisting of 16 byte-wide FIFOs, each with depth 16. Each buffer will allow data to come in 16 bytes at a time (since data contained in a packet is 16 bytes) and each buffer stores data from a total of 4 algorithm data requests. One buffer supplies the algorithm with 1 byte of input data for the sliding window, and the other supplies the algorithm with the byte of data to be cancelled out from the sliding window. Each buffer has a separate data requestor, but the buffers only send data to the algorithm if both have data. The algorithm buffer supplying data to be cancelled waits until the buffer that supplies input for the sliding window has given 4093 bytes to the algorithm.

4.4 Running the Algorithm

Whenever the host wants the FPGA to start computing, it can enable the algorithm by writing to a specified MMR. Once the host has signaled the algorithm to start, the first Algorithm Buffer begins to send data to the algorithm module, and after 4093 bytes, the second one starts sending data too. The algorithm will compute as long as the algorithm buffers have valid data to send and as long as the ALG_GO flag is high. When an anchor is found, the Anchor Write Requestor generates a request for the unmasked anchor value to be written to host memory, at an address designated for anchor results.

Chapter 5

Simulation and Testing

This chapter describes the process of running the Anchor Detect system in simulation and verifying its operation.

5.1 Simulation Environment

The simulation and testing environment is based on one previously set up by NetApp to test the PCI-E card based victim cache design. Its main components are:

- the **FPGA logic** to be tested
- the **simulated memory model** of the DIMMs on the PCI-E card.
- a **text test file**, to act as the host and issue commands to the FPGA. This test file can be swapped out with other test files.
- a **command translator** that parses the text test file and converts it into a hex file that will serve as input to the command processor.
- a **command processor** that parses the hex test file. It connects to the top FGPA design and executes each test command by asserting the appropriate control signals or by calling on another module to construct PCI-E packets to forward to the top design module. It also prints out a log file to document the test.

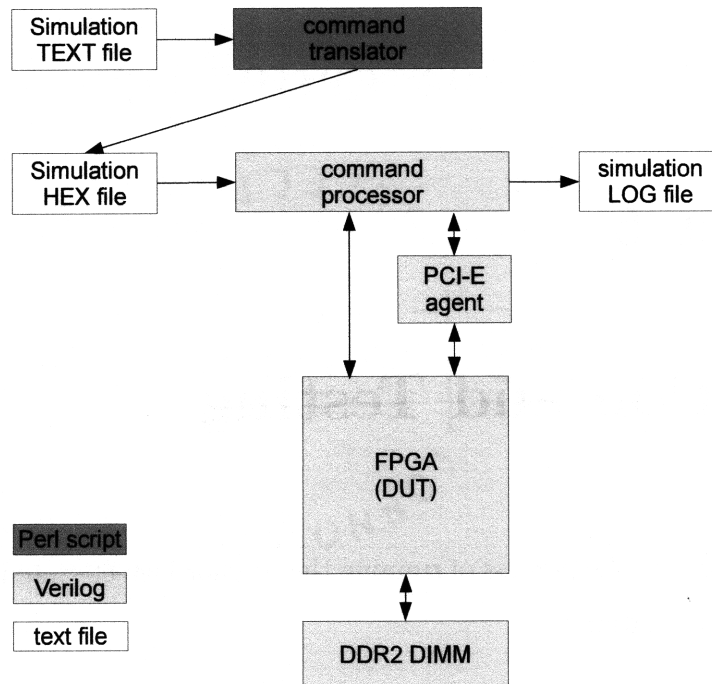


Figure 5-1: Block diagram of the simulation environment.

- a **PCI-E agent** module that is called on by the command processor to construct PCI-E-style packets to forward to the host whenever the test file calls for it. It also receives TLPs from the FPGA design and forwards them to the command processor.
- a **simulator**, namely ModelSim, to simulate the design.
- Perl scripts to glue it all together.

The testing environment is diagrammed in Figure 5-1.

5.2 Test File

The simulation environment defines a simple language for writing tests that includes commands to write certain values to MMRs, read MMRs and expect certain values back, and send data packets to the PCI-E device under test. This means the tests can

send the FPGA data for the algorithm, start the algorithm, and get results back. The packets intended for the host, as response from the FPGA logic, are received by the simulated PCI-E interface. If the packets received differ from the expected packets, the test will flag as "failed". The testing language also allows print statements, and the command processor adds them to the log file, along with print outs of packets expected and received.

A typical test first sets up BAR 0 values and other config space values, following the usual initialization procedure for PCI-E devices. The DMA engines are initialized by writing to specific MMRs. First, an MMR write enables the DMA engines. Then, there are head and tail DMA descriptor pointer registers whose values let the DMA Engines know where to look for the next descriptor. The most recent descriptor is written to the tail address, and the descriptor at the head is the next descriptor that will get processed. The tail address can either be updated before sending each descriptor to the FPGA, or one time before sending many descriptors at once. After receiving a DMA read descriptor, the FPGA will send the test bench a request for data. Once all transactions have been performed, memory locations can be read to make sure there were no errors in the file transfer. The algorithm can be enabled by an MMR write or enabled by default, meaning they start as soon as there is data. After the algorithm has finished computing, the test can read the anchor locations and values from the expected local memory addresses by DMA writes.

The testing data in which to find anchors was obtained from the Virtual Tape Library group of NetApp. The testing environment does not support looping or variables, so the test code was generated by a script that reads the raw byte data from a file and formulates it into packets of 64 bytes according to the test code format.

5.2.1 Testing the Algorithm Infrastructure

A sufficient test of the algorithm infrastructure would be able to verify:

- that all data arrives from host to FPGA
- all data arrives from FPGA to algorithm buffer

- algorithm buffer requests more data when room is available
- algorithm buffers give data only when both can

Test files can automate the verification of the arrival of data into local memory, but not any other of these points. With a waveform viewer, I verified that all the data written to the FPGA was passed on to the algorithm. I also verified with counters that algorithm buffers only gave data when neither was empty, and that there were as many algorithm results as bytes passed to the algorithm.

5.2.2 Testing the Algorithm Logic

To verify the the logic of the hash algorithm, I used 2 data sets, each with identical data except for the first byte. Seeing that this one byte difference causes the output of the final XOR of the algorithm in both cases to be different until 4093 bytes later verifies that it takes the expected amount of time for the effect of a byte to be cancelled out. As an automated and final test of the algorithm logic, the six anchor values and locations found by the Anchor Algorithm in 64 kB of data were compared to those found by the software version.

5.3 Performance Measurements

To measure the running time of the algorithm, I noted the wave time viewer mark for when the algorithm started processing (not including the time for initialization), and then noted the wave time viewer mark for which the algorithm finished computation on the last byte that was input. This was 151,000 nanoseconds total to process 16 kB of data. This means that the algorithm would be able to process 102 MB of data in 1 second. Processing 64 kB of data took 788,487 nanoseconds, for a rate of 81 MB of data processed per second. The slowdown was due to the Algorithm Data request address catching up to the latest address written to the FPGA.

Chapter 6

Conclusions and Further Thoughts

6.1 Performance Analysis

Transferring data from local memory to the algorithm buffers took on average 64 cycles for 64 bytes of data. However, transferring 128 bytes of data from the host to local memory took on average 224 cycles. The transfer of data from the host to local memory was the bottleneck of the design.

6.2 Possible Improvements

6.2.1 Parallelization

Each step of the anchor algorithm depends on the result of the step before it, and this further means that the processing of the current byte depends on the result for the previous byte. As such, the algorithm cannot be pipelined in a simple way. However, as a possible future improvement, the algorithm could be parallelized based on the fact that as a windowing algorithm, results for a 4093 byte size window at a certain shift would be totally independent from the results at a shift that was more than 4093 bytes greater than that current shift. Doing this would require a separate algorithm "engine" for each parallelization, where an engine includes an algorithm module, and two algorithm buffers. Processing 4093 bytes of the algorithm for 1

result as if it is totally independent is 4093 times more expensive than processing the next result by depending on the previous result, so each additional parallelization comes with an overhead cost. Separate algorithm engines, then, should process data of shifts quite far apart from each other. Various implementations would need to be simulated and compared to determine how many parallelizations exist in the most efficient configuration, and how widely spaced they should be.

6.2.2 Host to card data transfer

Since each transfer has an overhead cost independent of how many bytes are being transferred, the smaller number of transfers it takes. Currently, each descriptor in the system transfers 128 bytes, but this could be increased to 4kB per descriptor in order to achieve closer to the maximum data transfer rate. The maximum transfer rate for one DMA engine is about 400 MB/s, constrained mostly by the limited descriptor pre-fetching allowed in the simplified design of the DMA engine itself. The maximum rate limit of a PCI-E x4 interconnect is about 800 MB/s and perhaps a little less after considering the overhead of the header for each TLP, where each TLP is limited to 128 bytes by the host system requirements.

6.3 Final Thoughts

The algorithm was successfully implemented on the FPGA with acceptable performance but was not faster than the software implementation, which improved in efficiency over the course of implementing the thesis design and now computes at 150 MB per second. A non-parallelized version of the algorithm at best would process 125 MB per second, and a parallelized version could be faster. With host descriptors that transfer more data to the FPGA at a time, parallelization could be used to achieve significant speedup. In this case, transfer of data from local memory to the algorithm would also need to be made faster. In the end, the hardware version would still be limited by the I/O bandwidth to host memory from the PCI-E interface, whereas the host processor has a memory bandwidth of approximately 3GB/s.

Hopefully this implementation can serve as a starting point to build up in order to accelerate the Anchor Detect algorithm and other deduplication algorithms.

Bibliography

[1] Preston, W., *The Skinny on Data Deduplication* 2007: Storage Magazine. available online at `http://storagemagazine.techtarget.com/magItem/0,291266,sid35_gci1237737,00.html`

[2] Oliver, T. and Schmidt, B. and Maskell, D., *Hyper customized processors for bio-sequence database scanning on FPGAs* 2005: Proceedings of the 2005 ACM/SIGDA 13th international symposium on Fieldprogrammable gate arrays, p 229-237

[3] Bensaali, F. and Amira, A. and Bouridane, A., *Accelerating matrix product on reconfigurable hardware for image processing applications* 2005: IEE Proceedings: Circuits, Devices and Systems, v 152, n 3, p 2360246.