Adapting the Skyline Operator in the NetFPGA Platform

By

Nathan Miller

Submitted in Partial Fulfillment of the Requirements

For the Degree of

Master of Computing and Information Systems

YOUNGSTOWN STATE UNIVERSITY

May, 2013

Adapting the Skyline Operator in the NetFPGA Platform

Nathan Miller

Signature:

Nathan Miller, Student Date

Approvals:

Dr. Graciela Perera, Thesis Advisor Date

Dr. John Sullins, Committee Member Date

Edmund Ickert, Committee Member Date

Bryan DePoy, Interim Dean of School of Graduate Studies and Research Date

ABSTRACT

Attacks directed at networks and end systems are continually on the rise. The issue has been gathering much attention as of late. In his 2013 State of the Union Address, the President of the United States addressed the severity of the issue in regards to national security. Systems which can detect and prevent attacks are an essential component in keeping our communication infrastructure safe. Many software solutions are publicly available, but are unable to meet the demands of a high rate networks. These environments require platforms which provide open source hardware accelerators designed for the Internet Protocol.

The goal of this study is to determine the complexity of adapting the Skyline algorithm – a well-known algorithm used in databases – to the NetFPGA hardware accelerated platform. Additionally, one possible application of Skyline could be detecting the presence of malicious packets under the assumption that criteria for malicious packet detection are given prior Skyline computation. The contribution of this work is twofold. Firstly, we propose a novel representation of the Skyline BNL algorithm using a state machine for two dimension multi-criteria consisting of the packet length and number of reoccurring IP addresses. Our goal is to create an implementation to help determine design considerations for translating Skyline to the NetFPGA. Finally, we develop an implementation of Skyline using the hardware description language Verilog. The evaluation of this implementation was performed using Icarus Verilog. The results indicate our prototype can accurately compute the Skyline of network communications across two dimensions.

Acknowledgements

I would like to dedicate this work to my late mother Pauline J. Miller. You are the most amazing person I will ever have the pleasure of knowing. No word can describe your endless love and compassion, passion for higher education, and desire to help others. You are an inspiration and a major driving force in my pursuit of higher education. Without you, I would not be the person that I am today.

I would like to give special thanks to my father Dennis Miller. Your computer classes when I was a child were what made me choose a career in computing in the first place. I would like to give special thanks to my sister, Rachel Miller, Aunt and Uncle, Debby and Richard Miller, as well as the Vancleve family in Colorado. Without your love and support none of this would be possible.

I would like to thank my thesis advisor Dr. Graciela Perera. Her passion for research and working with students is an inspiration. My research would have never been possible without her support, feedback, guidance, and wisdom. I would like to thank Dr. John Sullins for all his help and support. I would like to thank Edmund Ickert, Virginia Phillips, and Mark Welton. Without great instructors such as yourselves, I would not be where I am today. Your support, guidance, and classroom instruction have made my student career at Youngstown State University an enjoyable experience.

I would like to give special thanks to Stanford University and the National Science Foundation for the donation of the NetFPGA which made this project possible. I would also like to thank Adam Covington and the NetFPGA research group at Stanford University for the excellent summer NetFPGA workshop.

Table of Contents

List of Figures

Chapter 1: Introduction

1.1 Motivation

The Skyline operator or Skyline (Skyline query) is a recent research area in databases with many interesting applications to be explored in network security. There is much literature related to its computation since it was first introduce in the database area in 2001. The Skyline is a major paradigm used in resolving problems related to planning and decision-making (multi-criteria) in databases [20,31]. It can be stated as a fundamental problem in computer science, which studies finding all maximal over a set of vectors. Kung, Luccio, and Preparata proposed the first skyline algorithm in 1975. Since then, numerous applications in databases, wireless networks and mobile devices have been proposed [31]. Recently, in 2009 Mueller, Teubner, and Alonso described a query compiler called Glacier using FPGAs. Thus, they are part of a research effort studying the advantages of implementing database query processing tasks using FPGAs [32]. The work discussed [32] opens the possibility of implementing queries in FPGAs and investigating the application of Skyline in network security. Specifically, it is interesting to study queries for network security applications that allow detecting packet anomalies or malicious traffic over the Internet. These applications can be used to prevent the staggering growth of network security attacks occurring on the Internet and that continually threaten our communication infrastructure.

Standalone FPGAs may not be suitable to protect against network attacks because they lack network support for packet capture. They also lack support for Internet Protocol (IP) version 4 which is essential to perform packet header inspection. This will increase the complexity of implementing queries such as the Skyline over IP [33]. One possible

solution is using the open source NetFPGA platform. The NetFPGA employs a special purpose Field Programmable Gate Array (FPGA) that enables a researcher to prototype new or existing networking technologies directly into hardware. Additionally, a NetFPGA is more suitable for network security applications than the standalone FPGAs as it can provide 4-port Gigabit Ethernet Card with a special purpose onboard FPGA with support for IP [11,12,16]. Using a NetFPGA will increase the feasibility of using Skyline for network security applications and offer the opportunity to study challenges that arise when implementing applications with FPGAs.

1.2 Problem

Increasing attacks on the Internet threaten our communication infrastructure and safety of our critical systems. Systems that can detect packet anomalies or malicious packets can help determine potential threats. A particular challenge faced by these systems is the large number of packets that need to be processed at very fast speeds [16]. Specifically, when systems such as Intrusion Detection System (IDS) are capturing and processing packet anomalies or detecting malicious packet. The main problems IDS face when performing packet inspection may include increasing perceived network latency by the user, flexibility for multi-criteria queries, disrupting the network service. This problems maybe solved by adapting Skyline such that an IDS can process multi-criteria queries of network packets quickly and correctly. Existing software simulation methods may not provide suitable solutions because of the specific purpose hardware found in current computers [24,32]. A potential solution could be using FPGAs or network hardware accelerators as the NetFPGA to implement a two dimensional Skyline that can detect packet anomalies or malicious packets.

Adapting the Skyline to the NetFPGA could help understand what constitutes a malicious packet and how we can detect packet anomalies or malicious packets without increasing user perceived network latency. Our principle assumption is that initial criteria for detecting packet anomalies or malicious packets exist. We can then use Skyline to discover new or more refine grained criteria that can be included or adapted in an intelligent network security protocol for the Internet or in an Intrusion Detection Systems. Thus, we first need to study the complexity and feasibility of adapting Skyline in the NetFPGA platform. This is constitutes the main problem addressed by this thesis.

1.3 Contributions

The principal goal of this thesis was to study how the Skyline could be adapted in the NetFPGA platform. It is the first such investigation to our knowledge.

The contributions of this thesis are two fold:

1. Firstly, we proposed a novel representation of the Skyline using a state machine for two dimension multi-criteria consisting of the packet length and number reoccurring IP addresses.
2. Finally, we describe the Skyline implementation and its verification using Icarus Verilog. The implementation of Skyline in Icarus Verilog can aid in fully implementing the Skyline in the NetFPGA. Thus, we implemented a small Skyline proof of concept prototype for particular cases of Skyline in the NetFPGA.

1.4 Organization of this thesis

The remainder of the thesis is organized as follows. Chapter 2 is the background information required to understand our proposed solution. Thus, we describe the principal

concepts including Skyline and NetFPGA. Chapter 3 describes the method used to setup and evaluate our adaptation of the Skyline in the NetFPGA. Chapter 4 illustrates the results obtained and discusses adapting issues for Skyline in the NetFPGA. Finally, conclusions and our outline of future research directions can be found in Chapter 5.

Chapter 2: Background

2.1 Skyline

Computing points of interest – finding a point or location, which a person may find intriguing or useful – is a non-trivial task when considering the preferences of users. Although, databases languages such as SQL are capable of enforcing referential integrity, these considerations are not natively inherent without writing a complex structured query. The phenomenon can be explained by the following two principles. First, not all criteria are created equal. As an example, a user might place more importance on the color of a product than a specific feature or price. Additionally, the person's values, beliefs, and ideals are also subject to change over time. All these things, which make an individual unique, means there is no one size fits all situation for everyone. Finally, not all criteria will agree with each other. A client may be looking for a new house in the city which is cheap and is in a low crime area. Although, exceptions can be made, it may not be the case that cheap inner city homes are in low crime areas. Algorithms, which consider user preferences, have many potential applications and are needed in these circumstances.

2.1.1 Example

As an example, assume a given dataset which consists of a list of hotels containing attributes such as distance relative to the beach and booking price. A customer is considering placing a hotel reservation at a location which is both cheap and

close to the beach. The criteria price and distance could be considered. However, these criteria are not complementary to one other; typically the hotels closest to the beach are the most expensive to book. Thus, an algorithm which takes into consideration user preferences, is required for computation.

2.1.2 Skyline Definition

Skyline, an algorithm used in relational databases, takes into consideration user preferences in relational queries [1,20]. Skyline performs a relational query against large datasets taking into consideration multiple (sometimes conflicting) criteria. The individual rows of the dataset in this context are called tuples. The contents of the computed Skyline are the points of interest which may warrant further investigation. In this scope, interesting data points are those points that dominate the non-interesting points in at least one of the specified criteria [2,19-20,22,25]. The Skyline computation is otherwise known as the maximum vector problem [1]. In this context, a vector can be defined as a row in a dataset and a component is an individual attribute belonging to a row.

The goal is to find the maximal subset of vectors which are not dominated by other vectors in the set. In this instance, a vector dominates another if each component has a value greater or equal to the corresponding component of the compared vector, or it has a higher value in at least of one of the corresponding components [1]. In Skyline, the row is referred to as a tuple and the column of each row an attribute. In the hotel example, the criteria for comparison could be the distance and price and the user preference of low cost and closeness to the beach. Based on this assumption, the

computed Skyline of the hotel dataset will contain the final list of interesting hotels according to both the criteria and user preferences.

If a hotel is not considered worse than all others in the dataset by at least one or both attributes, it should be added to the list. Since both criteria conflict one another, there may not exist any hotels which are both close to the beach and are cheap. Despite this, the hotels that are not dominated in at least one dimension might still be worth consideration. Once the interesting hotels have been determined and added to the Skyline; the user can then book a reservation based on their personal preference for distance from the beach and/or price. This consideration of user preference could be potentially advantageous for administrators of communication networks. In this study, we propose an adapted version of the Block Nested Loop (BNL) variant of Skyline.

2.1.3 Skyline Block Nested Loop

The Block Nested Loop (BNL) algorithm is a scan based algorithm and compares each tuple with every other tuple present in the data set. Improved performance is achieved through self-organizing lists. Each point that dominates another in one or more attribute is moved to the beginning of the list. This is in effort to prevent tuples being compared more than once. The computed Skyline is then stored in main memory. The final contents in memory contain the best tuples. Performance of BNL is directly correlated to the size of the dataset which results in $O(n^2)$ comparisons. For this reason, BNL does not scale well when applied to larger data sets. We propose a state machine representation of Skyline BNL exhibited in Figure 1.

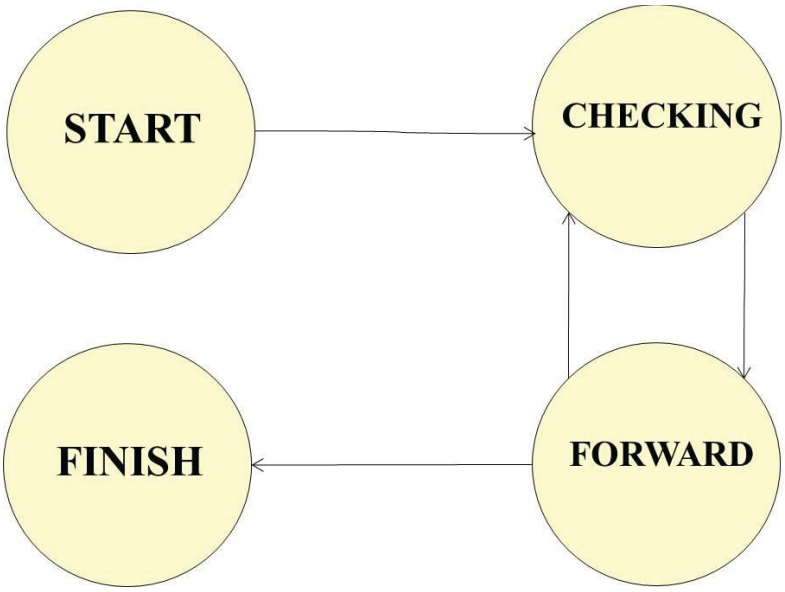

Figure 1. State Machine Diagram

Figure 2 provides the detailed psuedo code for the state machine. When in the START state, one element is copied from the object (O) to the top of the Skyline (W). The state machine then transitions to the CHECKING state where each element of W is compared to that of O. Depending on the condition met, the variable obj_flag is set to either true or false. If an element from O dominates an element from W then the element in W is removed and obj_flag is set to true.

If an element from W dominates and element from O then obj_flag is set to false. An element is added to the Skyline based on this value. Once an iteration is finished and the value of obj_flag has been set, the state machine then transitions to the FORWARD state. This state advances the current iteration and adds an element to the Skyline if the value of obj_flag is true. The state machine then transitions back to the CHECKING state where the next element from O is compared. This process repeats until all elements of O are compared to those in W. Once complete, the state machine transitions to the FINISH state and contents of W contain the final Skyline for the dataset. Figure 3 illustrates a

potential application of the Skyline BNL algorithm when applied to network communications.

```
STATE 1: START

Upon windown_size = 0 and obj_num > 0
    //Copy one element to Skyline window (W)
    window_iter = 1
    window_size = 1
    obj_iter = 1
    W [window_iter] <- O [obj_iter]
    obj_iter = obj_iter + 1

STATE 2: CHECKING
Upon windown_size > 0 and obj_num > 0 and
 window_iter ≤ window_size and obj_iter ≤ obj_num
//Window dominates object
IF  (W [window_iter].acc ≤ O [obj_iter].acc and
      W [window_iter].temp < O [obj_iter].temp)
                        obj_flag <- false
                        window_iter <- window_size
//Object dominates windows
ELSE
            obj_flag <- true
            IF  ( O [obj_iter].acc ≤ W [window_iter].acc and
             O [obj_iter].temp < W [window_iter].temp)
                                    W [window_iter] <- W [window_size]
                                    window_size <- window_size – 1
                                    window_iter <- 1
window_iter <- window_iter + 1

STATE 3: FORWARD

Upon window_iter > window_size and obj_num > 0 and
obj_iter ≤ obj_num
    //Compare next object in position obj_iter with all
    //objects in the Skyline window (W)
    IF ( obj_flag = true )
                        window_size <- window_size + 1
                        W [window_size] <- O [object_iter]
    obj_flag<- false
    window_iter <- 1
    object_iter <- object_iter + 1

STATE 4: FINISH

Upon obj_iter > obj_num
    // All objects in O compared
    // W contains the Skyline
    0 < window_size ≤ obj_num
```

Figure 2. Psuedo Code for State Machine

2.1.3.1 Motivating Example

In figure 3, incoming network transmission are ranked according to the packet attributes length and a measure of packet loss. Let ‘O’ represent a set of inbound network packets. Let ‘W’ represent the computed skyline. We propose the following states for Skyline computation: START, COMPARE, FORWARD, and FINISH. In our proof of concept prototype, we consider the user preferences of minimum packet length and a measure of minimum packet loss, which equates to the number of packets lost versus the total number of packets received. Upon arrival of the first packet, the START state is initiated. We extract the source IP address and packet length fields of the first packet received and store these values in ‘O’ and ‘W’. For demonstration, the associated packet loss is statically defined.

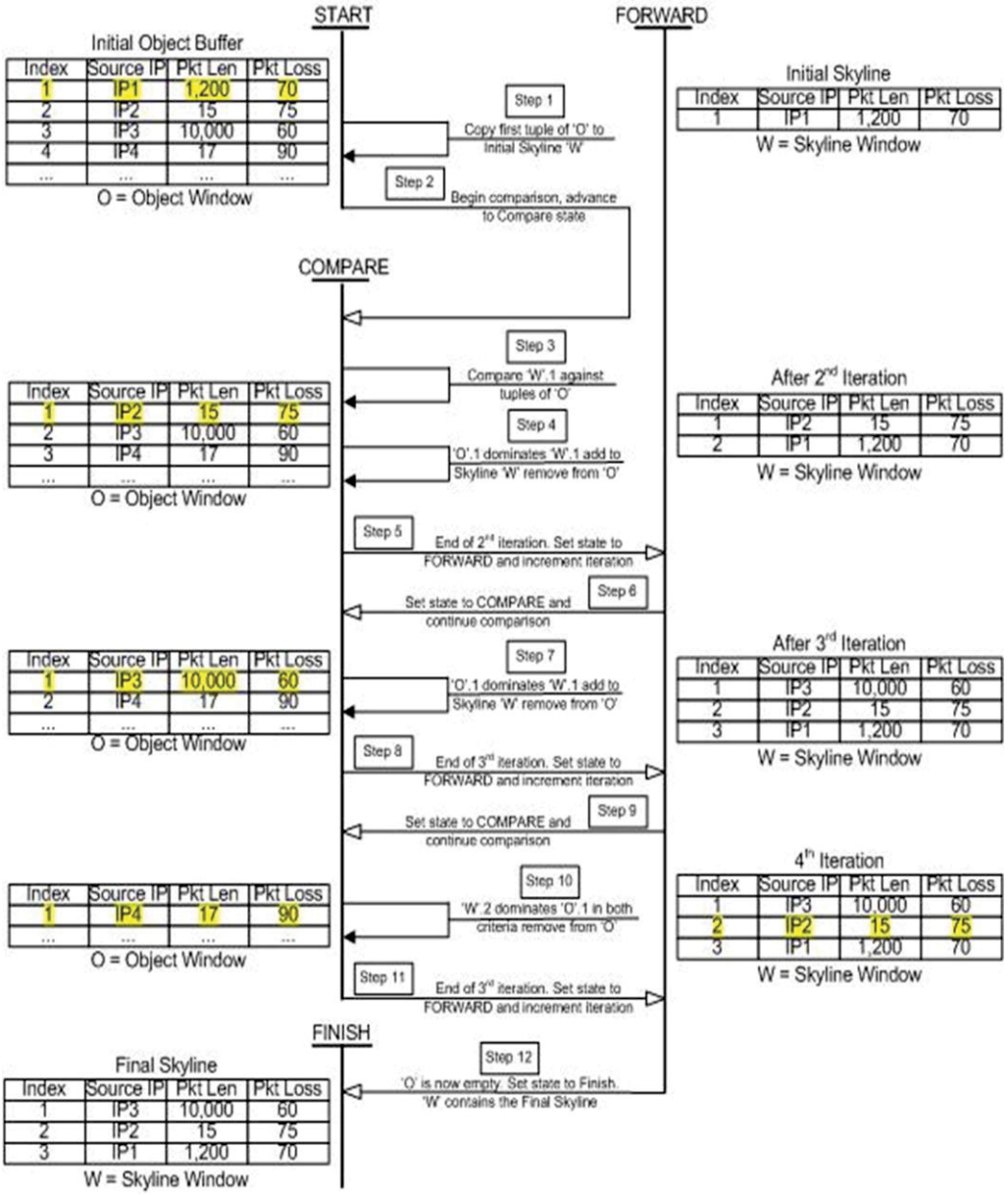

Figure 3. Skyline BNL State Machine

We then populate ‘O’ with each successive packet received within a window of time. Once ‘O’ has been populated the state machine then transitions to the COMPARE state. The attributes of the first tuple of ‘W’ are compared against the first of ‘O’. The contents of both buffers are then compared according to the following rules:

- A tuple in ‘W’ dominates those stored in ‘O’ only if the packet length of ‘W’ is less than that of ‘O’. If the packet loss of ‘W’ is less than that of ‘O’. If at least one of the condition is true then add the tuple in ‘O’ to the top of the set in ‘W’.
- A tuple in ‘O’ dominates those stored in ‘W’ only if the packet length of ‘O’ is less than that of ‘W’. If the packet loss of ‘O’ is less than that of ‘W’. If at least one condition is true then discard the compared tuple in ‘W’. Then store the new dominant tuple of ‘O’ at the top of the set in ‘W’.
- A tuple is considered incomparable if at least one condition is true in both the previous scenarios. If so, add the tuple in ‘O’ to the top of the list in ‘W’.

Once the tuple in ‘W’ has been compared to all of those in ‘O’ the state machine transitions to the FORWARD state. This marks the end of the first round of comparisons. At this state, the iteration is incremented and the next round of comparisons begins. The state machine then transitions back to the COMPARE state. This process is repeated until the contents of ‘W’ contain the computed skyline of the dataset. As illustrated, this approach could potentially prove useful in identifying abnormal traffic patterns. The Skyline could be potentially used to complement an existing Intrusion Detection System by offloading performing this pre-luminary packet header inspection. Packets exhibiting unusual characteristics could be filtered by Skyline and forwarded to a software based IDS for deep level packet inspection.

2.2 Intrusion Detection

2.2.1 Overview of Publicly Available IDS Systems

A typical Intrusion Detection System can be classified as model based or non-model based. In model based system baseline analysis is required to establish the normal

conditions of traffic patterns. This is done in order to construct a predictive model or rule set. Non-model based IDS systems attempt to classify packets using statistical methods. A related study surveys different Intrusion Detection techniques for communication networks [4]. Much has been accomplished in developing new systems for signature and anomaly based Intrusion Detection Systems (IDS) [3,4,13,17]. Anomaly detection solutions such as ADAM, Snort, and Bro require the use of machine learning algorithms which require additional processing time. Skyline could be used to complement such systems in order to achieve increased performance.

2.2.2 Hardware Accelerated IDS

Similar work in IDS acceleration has also been investigated with the Shunt system [24]. The Shunt employs a hardware packet filtering system utilizing the NetFPGA platform. Shunt performs inline operation, LAN operation and IDS load balancing [24]. The authors conclude there are significant advantages, with regard to performance, in offloading pre-luminary packet header investigation on to a dedicated hardware platform. Another study implements a basic signature based Network Intrusion Detection System (NIDS) using string pattern matching [14]. These systems utilize an open programmable FPGA which is discussed in the following section.

2.3 FPGA Introduction

Although, the objective of this study does not include a comprehensive view of Field Programmable Gate Array (FPGA) architecture, it is still relevant to understand the basic concepts and components on an FPGA. An average CPU found in a typical desktop computer has a set of static functions associated with it. These functions are utilized by software and remain unchanged. Although, instructions may be executed and interpreted

by the CPU in a different order, the chip logic always remains the same. An FPGA is a special type of computer chip, which contains programmable logic components called Configurable Logic Blocks (CLBs) and reconfigurable inter-connects that connect these components together [10,28].

2.3.1 Reconfigurable Computing

Using a concept called reconfigurable computing the designer can reconfigure the actual chip circuits. In essence, the hardware can be tailored to the individual needs of the program or application. For instance, once the hardware has been configured and shipped by the manufacturer, the chip circuits can then be reprogrammed after its initial deployment, hence the term Field Programmable. The basic FPGA configuration is typically provided by the manufacturer. This design can then be modified by the end-user using a Hardware Description Language (HDL) such as Verilog or VHDL. In addition, this offering can achieve higher efficiency than a typical CPU. Another study compares the efficiency of elliptic curve cryptography using an FPGA versus an optimized software application executed on a Xeon processor [6,21]. The results indicate that the reconfigurable computing design was 540 times faster at a 40 times slower clock speed [6,21].

Additionally, the power consumption of a reconfigurable platform – such as an FPGA – when optimized for a specific application, reduces power consumption compared to general purpose processors [21]. A related study determines that moving critical software loops over to reconfigurable hardware results in average energy savings of 35% to 70% and an average increase in speed of 3 to 7 times, depending on the device used [8,21]. An alternative to reconfigurable devices are ASIC (Application Specific

Integrated Circuit) chips, which consume less power, requires less area, and are more efficient. “The flexible nature of an FPGA comes at a significant cost in area, delay, and power consumption: an FPGA requires approximately 20 to 35 times more area than a standard cell ASIC, has a speed performance roughly 3 to 4 times slower than an ASIC and consumes roughly 10 times as much dynamic power [9].” Despite the advantages ASIC chips offer, reconfigurable platforms are typically quicker to the market and more flexible since a change in design does not affect the external hardware. “ASICs typically take months to fabricate and cost hundreds of thousands to millions of dollars to obtain the first device; FPGAs are configured in less than a second (and can often be reconfigured if a mistake is made) and cost anywhere from a few dollars to a few thousand dollars [9].”

2.3.2 FPGA Components

An FPGA contains CLBs which contain the logic for implementing the actual functionality of the circuit. According to the book titled FPGAs 101, the author suggests the basic components of an FPGA are: basic building blocks, Input/Output (I/O) interfaces, and interconnections [10,28]. The I/O interfaces are responsible for both passing data from internal logic to external sources or vice versa. The basic building blocks are the pre-configured logic circuits which contain the internal logic to perform design specific functions. This is where the actual circuit logic is implemented. Finally, the interconnections tie the different building blocks together as well as the I/O interfaces to interface the internal logic with external sources.

Although different FPGA manufacturers utilize different terminology, each FPGA contains these same basic components. In relation to Xilinx FPGAs the basic components

are: Configureable Logic Blocks (CLBs), Input/Output Blocks (IOBs), Programmable Interconnect, and other resources such as memory, multipliers, global clock buffers, and boundary scan logic.

Each FPGA contains a finite number of CLBs and inter connects. CLBs serve as the basic logic units in an FPGA. These CLBs can be repurposed and reconfigured after the manufacturing process. The open nature of an FPGA offers higher flexibility for prototyping experimental systems. In disciplines such as computer networking; open reconfigurable hardware platforms are not available to the research community. This can make the prototyping of new hardware designs difficult. Additionally, the price of individual components for creating customized hardware platforms can be costly. With research and industry collaboration; the NetFPGA open source development platform has been made available to consumers and the research community.

2.4 NetFPGA

2.4.1 Description

The term NetFPGA stands for Networked Field Programmable Gate Array (NetFPGA). The NetFPGA is an open source hardware development platform developed by Stanford University and Xilinx. This platform is used by Students, Teachers, and Researchers for prototyping and deploying high performance network hardware designs. The NetFPGA variant used in this study is the NetFPGA 4x1G card. The NetFPGA 1G card is a 4-port Gigabit Ethernet card with a PCI bus interface and employs an onboard Xilinx Virtex-II Pro 50 FPGA. The onboard FPGA yields the previously stated advantages when used for customized computing. These advantages translate extremely

well when performing packet processing on live network communications. The FPGA logic allows for line rate processing of packets at Gigabit speeds.

The logic implemented in the FPGA determines the functionality of the hardware and can be redesigned or repurposed in the field. This gives the researcher/designer some freedom whereas normal hardware configurations come already pre-configured for a specific purpose.

2.4.2 NetFPGA Components

NetFPGA consists of two key components the hardware and NetFPGA Base Packages (tools, source code, and reference designs). Figure 4 contains the block diagram of the NetFPGA hardware. The NetFPGA offloads processing from the CPU via DMA transfer through the PCI bus interface. Figure 5 illustrates the NetFPGA Reference Pipeline. All projects in the provided source code utilize the modular design illustrated in Figure 5. As a packet traverses the NetFPGA it travels through the unified pipeline.

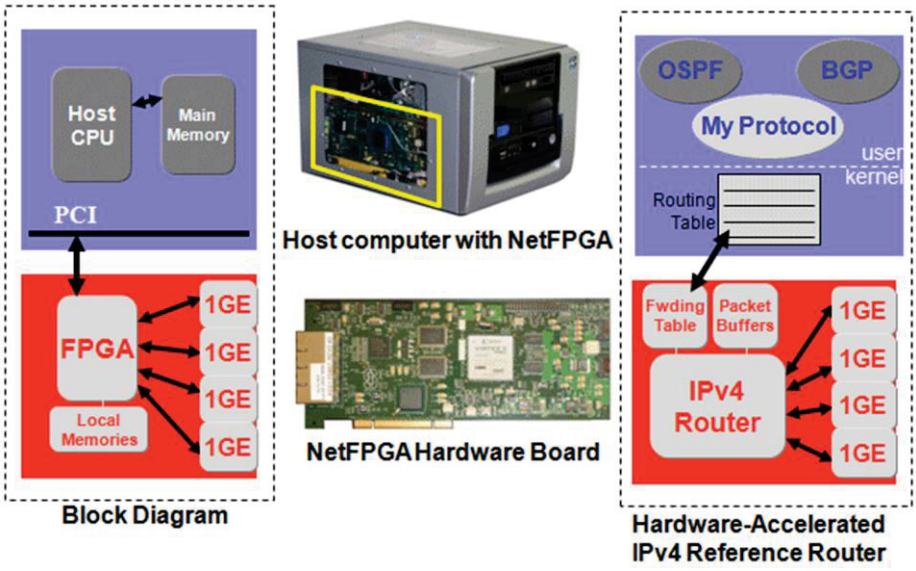

Figure 4. NetFPGA Hardware Block Diagram

The NetFPGA contains 4 Gigabit Ethernet ports. Each port has access to both a physical (MAC) and DMA (CPU) transmit (TxQ) and receive (RxQ) queues. The MAC TxQ and RxQ queues correspond to packets being received or transmitted to/from the wire. Packets received on the CPU RxQ queue are those packets which are arriving from the PCI bus via DMA transfer. Packets placed in the CPU TxQ are those packets which are to be sent to the network stack. The hardware interface driver initiates a DMA transfer though the PCI bus. Once the driver receives an interrupt signal, the DMA transfer has been completed.

2.4.2.1 Hardware Architecture (Life of a Packet Traversing the NetFPGA)

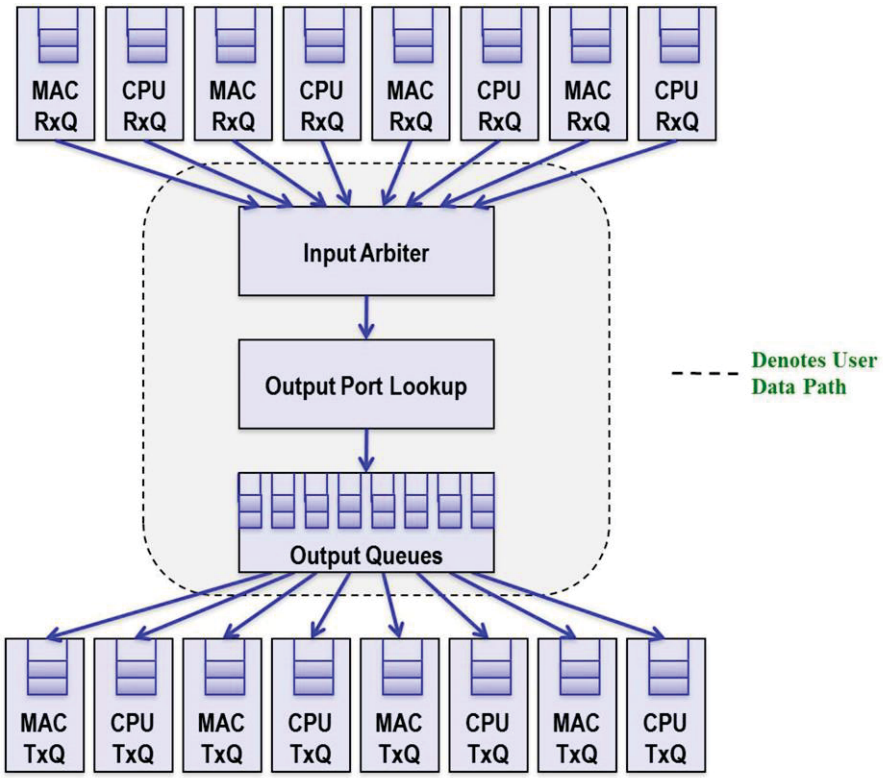

Figure 5. NetFPGA Reference Pipeline

Tying these concepts together, a software process sends packets to the NetFPGA via network socket which is received by the hardware interface driver. The interface driver then sets up and initiates the DMA transfer. An interrupt signal indicates the completion of the transfer. The packet then traverses the pipeline of the NetFPGA and is routed through the FPGA switching fabric. At the first stage of the pipeline, the packet is received and placed in an RxQ corresponding to the port it was received on. If the packet was received via the PCI bus the packet is placed in the corresponding CPU RxQ. Likewise, if the packet was received from the wire the packet is placed in the MAC RxQ for that port.

The queues connect to a wrapper called the User Data Path (UDP). Figure 5 depicts the UDP in the reference pipeline. At each stage of the UDP a different level of processing is performed on the packet. The UDP is modular by design which allows a designer to insert functionality at any stage and even introduce a new module into the pipeline. This process is also further simplified through the use of a unified pipeline across all ports. This simplifies the design process for the hardware designer as the hardware logic must only be implemented in one place. The first stage is the Input Arbiter (IA) module. This module chooses which RxQ to service and transfers the packet from the RxQ to the Output Port Lookup (OPL) module. The OPL is responsible for making forwarding decisions which determines the port the packet will be sent out.

Once a decision is made the packet is then passed to the Output Queues module. This module places the packet in the TxQ determined by the previous module in sequence. As previously stated, if the packet destination is a remote network, the packet is placed into a MAC TxQ. In direct contrast, if the packet is to be sent to a software process then the packet is placed in the CPU TxQ. If the latter is true, an interrupt signal notifies the

hardware interface driver that a packet in the CPU TxQ is ready for transmission. The driver interface then sets up and initiates the DMA transfer. The packet is transferred and an interrupt signal is sent to the driver which indicates the completion of the transfer. The driver then passes the packet back to the network stack.

2.4.2.2 NetFPGA Packages

The final component is the provided packages and tools. The purpose of this section is to provide a basic overview of the testing infrastructure. The publicly available NetFPGA Base Packages provide the tools for compilation, testing and verification as well as the reference designs, libraries, and source code. The provided software libraries include scripts as well as additional features for testing, verification, and synthesis of hardware designs. However, additional software packages such as Xilinx ISE Foundation and Mentor Graphics ModelSim are required for simulation, visualization, and verification of hardware designs [12,15-16,23].

The simulation software enables the designer to test hardware functionality without having to go through the lengthy process of synthesizing the hardware. The provided software libraries contain the source code required the interface with the NetFPGA. The NetFPGA supports the following high level languages: C, Perl, Java, and Python. The testing infrastructure is Python based; test scripts can be written to test different features of a proposed hardware design through simulation. Once the hardware design has been synthesized and uploaded to the FPGA, the same testing scripts can be utilized to verify the design deployed in the actual hardware. When writing testing scripts the shared libraries provide the Python class objects responsible for grabbing the

parameters required for the creation of the IP packet. These parameters are then passed to a software process called Scapy which is responsible for the generation of the raw packet.

The developer assigns each test both a major and minor name. To initiate a test, a python script is invoked using the major and minor names previously defined as command line arguments. If invoked as a simulation, the CAD software will simulate the hardware design and generate the raw signals to pass through the hardware, according to the provided test script. Additional software packages, such as Model Sim, can be utilized to provide a visual representation of the simulated waveforms – which are the signals passing through the hardware during simulation. This visual representation is an invaluable tool for troubleshooting and debugging.

Chapter 3: Method

The NetFPGA was first developed by researchers at Stanford University in the year 2007. Although, the NetFPGA 1G card is well supported through the available website [26], and wiki [11,12], the documentation is not entirely comprehensive. Working knowledge in Linux system administration and computer networking is essential. In this section, we describe the approach utilized to create an experimental lab for NetFPGA research. This section does not provide a detailed step by step process for NetFPGA configuration. Only an overview of the major steps is discussed. However, these steps can be adapted and reproduced for constructing a NetFPGA research lab elsewhere. This discussion has been split into two components: platform and software. The platform component includes any software and configuration steps required to make the hardware operational. The software component includes all additional software packages required for setting up the testing and simulation infrastructure.

3.1 Platform Configuration

3.1.1 System Configuration

The NetFPGA hardware can be purchased separately or bundled in pre-assembled system. If assembling the system build separately, the wiki [11] provides a list of supported hardware. The first step is the installation of the user operating system. This process is generally straight forward. We chose Cent OS 5 (variation of Red Hat) as the user operating system, when we originally acquired the system components, CentOS 5 was a supported operating system, however, at the time of this writing Fedora is now the only officially supported operating system. Thus, official support for CentOS has been discontinued. This is a crucial point because the platform hardware, software, and toolset is not finalized and is constantly evolving. During installation the virtualization option and SELinux service must be disabled. Once completed any daemons which generate discover packets i.e. Avahi should also be disabled. This is not required but may be helpful in troubleshooting potential problems. The installation of the NetFPGA Base Package – which comes with the tools, libraries, reference designs, and scripts necessary to interface with the NetFPGA hardware – requires at least Java 1.6 and the Java Development Kit 6 update 6 to be installed. The NetFPGA base package is available through the wiki [11,12] or through the publicly available yum repository.

3.1.2 Hardware Configuration

Once successfully installed, the memory modules, for the Micron DDR2 SDRAM and Cypress SRAM, must be installed in order to conduct hardware simulations. The next major step is the compilation of the NetFPGA hardware interface drivers. To do this, the kernel development packages must be installed. The drivers can then be compiled by

issuing the ‘make install’ terminal command at the root of the NetFPGA package directory tree structure. This initiaties the driver compilation process and subsequently loads the compiled drivers. After this task, the CPCI must be reprogrammed using a provided script in the NetFPGA base package. Additionally, the CPCI must be reprogrammed each time the system boots. The hardware can then be verified by downloading the provided selftest bit file to the FPGA using provided scripts. At this phase the platform component should be operational.

3.2 Software Configuration

The NetFPGA Base Package provides the Verilog source code required for users to compile, simulate, and synthesize hardware designs for the NetFPGA. The software component of installation requires the installation of CAD tools, simulation software, and other software packages for testing and verification. Instructions for installation and version information on the CAD tools and simulation software are available through the corresponding section on the wiki [12]. This process is not arduous but is required for rebuilding the hardware circuits. The CAD software used in this study is Xilinx ISE Foundation 10.1 with Service Pack 3. This is also the only officially supported version of the Xilinx tools for NetFPGA development. Before installation, a license for the software and V2Pro TEMAC core from Xilinx must be obtained. The license can be obtained in one of three ways. Developers can request a 30-day evaluation license for both the core and software or purchase the full licenses through a sales representative. Academic users can also obtain the required licenses by mentioning the use of the NetFPGA when submitting a request through the Xilinx University donation program [12,27].

The simulation software used in this study is ModelSim 6.2G by Mentor Graphics. This is the only officially supported version. ModelSim allows for the simulation of hardware circuits and provides a visual representation of the simulated waveforms. When running simulation tests, the test bench assumes the use of this version of ModelSim [12]. The test bench generates signals to pass through hardware for use in simulation and verification. This allows for the testing of hardware logic without having to perform synthesis. Academic users may submit a request to Mentor Graphics for a full license of ModelSim. The Python version 2.7 compiler was used in this study and is required for conducting simulations using user defined test scripts. As previously stated, the testing infrastructure is based on Python. The user defines the experiment through a test script written in Python. The parameters used for packet generation are also defined in Python and passed to another software process. For this reason, the software package Scapy must also be installed – which is responsible for the generation of the raw packets when performing both hardware and simulation tests.

Chapter 4: Results

In this chapter, we provide an overview of the Skyline BNL implementation in Icarus Verilog. In the following section, we describe the data structure and data representation used in the design. This chapter concludes by describing the testing methods and results obtained.

4.1 Overview of the Hardware Description Language Solution

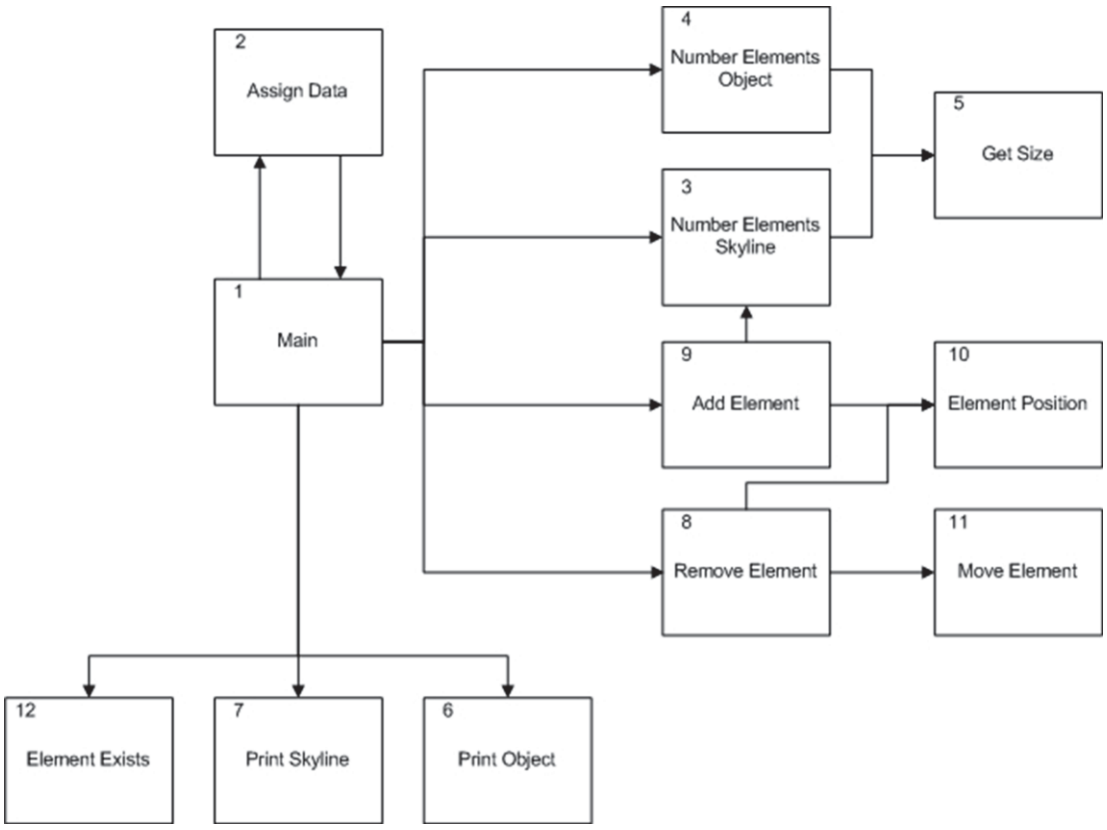

Figure 6. Module Relationship Diagram

The diagram in Figure 6 depicts the high level design of the Skyline Block Nested Loop implementation. In the proof of concept prototype, we sought to test the feasibility of implementing the Skyline BNL algorithm, typically used in databases, through a

hardware description language. This section provides an overview of the HDL implementation. The detailed source code can be viewed in Appendices A, B, and C. The main module consists of three phases: initial, skyline, and finish. In the initial phase we assign data to the initial object and initialize the Skyline window. The Skyline phase compares every object element to every Skyline element to compute the minimal values of data assigned in the initial phase. The Skyline phase can be further broken down into a start and compare process. The start process prints the contents of both the object and the Skyline window. In the compare process the object element is compared to what is currently in the Skyline window. It is here where the Skyline computation occurs. Finally, the finish phase prints the final contents of the Skyline window. The remainder of this section describes each phase in more detail.

4.1.1 Initial Phase

When the main module is in the initial phase, a task is called which assigns data to the initial object. In this context, the initial object contains the data for which we seek to identify the minimal values. For the proof of concept prototype, we assume:

- Packets are traversing a network and have been received from the wire by our node employing the Skyline hardware implementation.
- These packets have been intercepted in transit through the NetFPGA hardware. We store pieces of the packet header in memory, which could be based on user preference.
- Since, the goal is to analyze network communications, these values are stored and represented in their binary equivalent form.

- We define packet data as any element containing a non-zero value. If an element consists of all binary digit zeroes, then it shall not be considered as a candidate for Skyline computation.
- Storing packet header information for all packets traversing a live network would require a significant amount of storage space. Much more than would be available on a platform such as the NetFPGA. Therefore, this issue is beyond the scope of this work.

As an example, these preferences could be of interest to a network operator monitoring traffic flows. We utilize the preferences of minimum packet length and/or minimum number of packets exhibiting the same length value. These items are stored alongside the corresponding packets IP address. The IP address is utilized as an identifier and is not considered as criteria in computing the Skyline. In a real world deployment, the IP address could represent a network prefix which indicates the network of origin for packets exhibiting the user defined characteristics. This can also be helpful in identifying an attack site. Therefore, the stored information in our initial object represents the packets IP address, packet length, and number of similar reoccurring packets, which could indicate an attack.

In the proof of concept prototype, these values are statically defined and initialized by the "Assign Data" task as depicted in Figure 6. This task is called by the main module in the initial phase and defines the values for our initial object. Figure 8 depicts an example test case assigned to the initial object in our evaluation. Let each member of the object and Skyline window represent an element belonging to a list. Once data has been assigned, the main module initializes the Skyline window and copies the

first element of the object to the top of the Skyline. The tasks “Number Elements Skyline” and Number Elements Object” are then called which return the number of elements representing actual packet data in both lists. The values are subsequently used as loop counters to iterate through both the object and Skyline window. This signifies the end of the initial phase.

4.1.2 Skyline Phase

4.1.2.1 Start Process

The main module now transitions to the skyline phase. In the start process, the contents of both the object and Skyline are printed to the terminal through the “Print Object” and “Print Skyline” tasks. It is important to note that simulation directives, such as “$display”, are not synthesizable in final hardware designs. When a design is deployed to actual hardware, additional software must be written to utilize the hardware register system in order to read and print these values. Once the contents of both memory buffers are printed, the compare process of the skyline phase is initiated.

4.1.2.2 Compare Process

At the outset, the IP address, packet length, and occurrence value are stored as separate variables. A conditional loop compares every element belonging to object and Skyline window. Depending on the conditions met, a different series of events may transpire. Additionally, a task such as “Remove Element” may call a series of sub tasks.

Initially, only one element is contained within the Skyline window. This element is compared against all elements in the object until something better is found. An element from the initial object is added or removed to the Skyline window based on the following conditions and called tasks:

1. The Skyline values are less than those in the object. In this case, no operation is performed. The Skyline element is already better than what is being compared in the object.
2. The Skyline values are greater than those in the object. If true, remove the current Skyline element and add the new dominant object element to the Skyline. Since something better has been found, we should remove the inferior element and add the better element to the Skyline.
    a. Call “Remove Element” task
    b. Call “Element Exists” task
    c. Call “Add Element” task
3. The Skyline values are equal to those in the object. If this condition is met, then no operation is performed. It is likely that the element in the Skyline is being compared to itself in the object.
4. The element in the object is less than the Skyline element in at least one dimension. The object element does not totally dominate the Skyline element; therefore, the object element is incomparable. Nothing is removed from the Skyline. However, since the object element is better in at least one dimension, it should still be added to the Skyline.
    a. Call “Element Exists” task
    b. Call “Add Element” task

Figure 7 highlights the portion of code where the Skyline window is computed and these conditions are evaluated. Based on the condition met, different tasks and sub tasks may be called, which result in a different series of events. As an example, if

condition four is met, the “Element Exists” task will determine if the dominant object element is already in the Skyline window. If the element is not already present, then it should be added to the Skyline and the “Add Element” task is called. Otherwise, the object element is already in the Skyline, no operation need be performed. The “Add Element” task calls both “Number Elements Skyline” and “Element Position” which are used to determine how many elements are in the Skyline window, and where the dominated element exists in the Skyline. A series of conditions are then evaluated to determine where the dominant element shall be placed in the Skyline window.

If condition 2 is met, the dominated element in the Skyline will be removed before the new dominant element is added. In this case, the “Remove Element” task is called. This task first calls the “Element Position” sub task which determines where the element to be removed is located in the Skyline window. The element is then re-initialized to all binary zeros. Remove element then calls the “Move Element” subtask, which shifts other elements in the Skyline Window to a higher position in the list. This allows the Skyline computation to continue in the next iteration. The process for adding the dominant element to the Skyline is similar to condition 4. Once an iteration is complete, the loop counters are incremented and the process repeats until the final Skyline contains the minimal values from the object.

```
if (tempPktLen < objPktLen && tempOccur < objOccur)
  begin
    $display("Window element %d dominates obj element %d",loopCount,i);
    $display("Ignore current obj element");
    $display("");
  end

else if (tempPktLen > objPktLen && tempOccur > objOccur)
  begin
    memIndex = i;
    $display("Obj element %d dominated Window element %d",i,loopCount);
    $display("");
    $display("Removing Window Element %d Adding Obj %d",loopCount,i);
    removeElement(tempWindow[loopCount]);
    incompare = "False";
    elemExists(obj[i],elementExists);

      if(elementExists == "False") begin
        addElement(obj[memIndex],loopCount,i,incompare);
        printSkySummary();
        end
      else if(elementExists == "True") begin
        $display("Element Exists = %s",elementExists);
        $display("Dominant obj element already in Skyline, do nothing!");
      end
   end

else if(tempPktLen == objPktLen && tempOccur == objOccur) begin
    $display("Obj element %d is equal to Window element %d",i,loopCount);
  end

else if(tempPktLen > objPktLen || tempOccur > objOccur)
  begin
  elemExists(obj[i],elementExists);
  if(elementExists == "False") begin
    $display("Element %d in Obj was better in atleast one dimension",i);
    $display("Adding element to Skyline");
    incompare = "True";
    addElement(obj[i],loopCount,i,incompare);
    printSkySummary();
  end

 else if(elementExists == "True") begin
    $display("Element %d in Obj was better in atleast one dimension",i);
    $display("Dominant obj element already in Skyline, do nothing!");
  end
end

 else
   $display("Uh oh! None of the conditions are true");
```

Figure 7. Main Module Code for Skyline Computation

4.1.3 Finish Phase

Once complete the main module transitions to the finish phase and the final Skyline contents are printed to the terminal.

4.2 Data Structure Used for the proof of concept prototype Skyline Implementation

For the proof of concept prototype, we make the assumption that packets are traversing a network. These packets are perceived by the NetFPGA hardware as a stream of signals. These signals represent a sequence of binary digits. Using a hardware description language, we record the packets IP address and length as it traverses the NetFPGA. A counter in hardware calculates the number of packets from the same source exhibiting a similar length value. The parameters of packet length and occurrence serve as the basis for the Skyline computation. In this implementation, these values are statically defined and stored in a register. Since the goal is to simulate real time network transmissions, these values are also represented as a stream of binary digits.

Only data streams representing packet header data shall be considered for Skyline computation. The object and Skyline window are both defined as a register in memory with a vector width of 56 bits and an array dimension of four rows. An example test case is illustrated in Figure 8 which depicts how these values are stored and represented. Each row in both registers represents a distinct stream of data. Both the vector width and array dimensions are defined as a range of values. Each stream is stored by the most to least significant bits. The first 32 bits contain the IPv4 IP address. The next 16 bits represent the packet length and has a maximum value of 2^{16}. The final 8 bits represents the number of similar packets and has a maximum value of 2^8. Since, these values are statically defined we do not consider an occurrence value larger than 2^8.

```
//----------------------------------------------------------------------
// Test 2
//
// The contents are:
//                      IP Address    Length      Occurences
//
//              obj[0] = 192.168.1.1     5000       61
//              obj[1] = 192.168.1.2     5001       75
//              obj[2] = 192.168.1.1    12000      100
//              obj[3] = 192.168.1.2     6000      101
//----------------------------------------------------------------------

obj[0] = 56'b11000000101010000000000100000001000100111000100000111101;
obj[1] = 56'b11000000101010000000000100000001000010011100010010010011;
obj[2] = 56'b11000000101010000000000100000001001011101110000001100100;
obj[3] = 56'b11000000101010000000000100000001000010111011100000110010l;
```

Figure 8. Example Test Case

As previously stated, streams of data are represented as a series of binary digits representing signals received from the wire. In simulation, the absence of data in a register is represented as a series of X's; however, the actual value is unknown. This indicates an absence of a signal in a register. For sequential and combinational logic, it is difficult to test for the absence of a signal. Similar to a constructor in software; we solve this problem by initializing both registers to all binary zeros. Hence, we define packet data as any element within a register exhibiting a non-zero value. If an element consists of all binary zeros, then the contents of an element represent nothing.

These elements are not considered for Skyline computation. For every iteration, each parameter is stored in separate registers. This is accomplished by referencing the corresponding vector widths in the data stream. The individual registers are then used as the basis for the Skyline computation.

4.3 Results

To evaluate the hardware logic, five test cases were defined which can be viewed in Appendix C. Additionally, all permutations of each test case were evaluated by manipulating the array index values. As an example, Figure 9 exhibits the expected

output for each permutation of test case 2. The order in which each element is stored affects the order of comparisons in the Skyline computation. For each permutation, the resulting output is expected to be the same. This allows up to 24 different scenarios per test. As exhibited in Appendix C, the five test cases differ by the number of elements which belong in the Skyline. All permutations of each test case were evaluated resulting in a total of 120 distinct scenarios. The output of each scenario was validated against a Java implementation of Skyline BNL authored by a previous student Jeremy Cummins. The results indicate the proof of concept prototype implementation can accurately compute the Skyline for all 120 scenarios.

```
Contents of Skyline Window

Element 0: 11000000101010000000001000000010001001110001000001111 01
Element 1: 00000000000000000000000000000000000000000000000000000000000000000
Element 2: 00000000000000000000000000000000000000000000000000000000000000000
Element 3: 00000000000000000000000000000000000000000000000000000000000000000

Element          0

IP Address: c0a80101
Packet Length:  5000
Occurences:  61

Element          1

IP Address: 00000000
Packet Length:     0
Occurences:   0

Element          2

IP Address: 00000000
Packet Length:     0
Occurences:   0

Element          3

IP Address: 00000000
Packet Length:     0
Occurences:   0
```

Figure 9. Expected Output for Test 2

Chapter 5: Conclusion and Future Work

Adapting Skyline to an accelerated hardware platform such as the NetFPGA poses several challenges that range from capturing packet header data to implementing data processing algorithms for analyzing packets under multi-criteria. We present the first investigation towards understanding its complexity and describe our efforts towards adapting Skyline in the NetFPGA platform. We had two main challenges. The first challenge was describing Skyline in a state machine paradigm, which had not been done before. We successfully represented Skyline using a state machine for two dimension multi-criteria Skyline consisting of the packet length and number reoccurring IP addresses. State machines are a common paradigm used to represent solutions for accelerated hardware platform.

Our second challenge consisted in learning the hardware description language Verilog used by the NetFPGA. We successfully overcame this challenge by using Icarus Verilog. Icarus is a simpler programming environment that allows a novice programmer to grasp Verilog concepts. However, Icarus lacks the workbench required to evaluate the Skyline in hardware. This issue created a third challenge, as the evaluation of Skyline required a workbench where packets could be generated, captured and processed. We used the NetFPGA Network Interface Card (NIC) design and Skyline proof of concept prototype for particular cases of Skyline in the NetFPGA to evaluate our ideas.

Overall while software programming language environments over the past few decades offer considerable support to a programmer, hardware description languages used by accelerated hardware platforms have not [33]. This thesis contributes the first

study that seeks to bridge the gap of knowledge between the well-known databases multi-criteria solution Skyline and the NetFPGA.

Future work includes refining and further testing the proof of concept prototype of Skyline developed for NetFPGA. Also, we must consider evaluating and measuring the performance of the Skyline NetFPGA implementation with traces and real work network traffic. Finally, we would like to propose a novel Skyline algorithm that preprocesses data so that it can compute Skyline with less memory and processing requirements. This idea would focus on minimizing the number of Skyline packets compared by preprocessing the packets using a pivot or other similar ideas.

Appendix A

```
//----------------------------------------------------------------------
// Project: Skyline Block Nested Loop
// Author: Nathan Miller
// Date: 04/10/2013
// Filename: Skyline.v
// Description: Create an implementation of the Skyline
//                  Block Nested Loop algorithm in Verilog
//                  which could potentially translate to the NetFPGA.
//----------------------------------------------------------------------

module main;

`include "skytask.v"
`include "packetdata.v"

// Constants for case statement
parameter START = 1;
parameter COMPARE = 3;

// Variables for case statement
reg [7:0] skyline;

// Register for storing packet header information such as IP address,
// length, and the number of similar packets.
reg [55:0] obj[0:3];

// Store the IP Address, Packet Length, and number of Occurrences for the
// packet information received from the wire.
reg [31:0] objIP;
reg [15:0] objPktLen;
reg [7:0] objOccur;

// Temporary register used for comparing data
reg [55:0] tempWindow[0:3];

// Store the IP Address, Packet Length, and number of Occurrences for the
// packet information received from the wire.
reg [31:0] tempIP;
reg [15:0] tempPktLen;
reg [7:0] tempOccur;

// Variables for loop counters
integer i;
integer index;
```

```
integer iter;
integer loopCount = 0;
integer memIndex = 0;
integer add = 0;

// Variables for functions
parameter objSize = 4; // Constant defines depth of obj
integer counter = 0;
reg [79:0] messageA = "obj";
reg [79:0] messageB = "tempWindow";
reg [7:0] objNumElements;
reg [7:0] objElements = 0;
reg [7:0] nullVar = 0;
reg [7:0] addValue = 0;
reg [7:0] numElements = 0;
reg [79:0] objectCompared = 0;
reg [7:0] totalObj = 0;
reg [7:0] totalSky = 0;

reg [39:0] compareBegin = "False";
reg [39:0] firstCompare = "False";

reg [39:0] incompare = "False"; // Whether an element is incomparable
reg [39:0] endCompare = "False";

reg [39:0] elementExists = "False"; // Whether an element exists in the Skyline

 initial
   begin

   assignTestData(); // Assign the data for Skyline computation

   // Temporary window which is used for the Skyline computation
   tempWindow[0] = 56'b0; // All values are initialized to 56
   tempWindow[1] = 56'b0; // binary digits of 0
   tempWindow[2] = 56'b0;
   tempWindow[3] = 56'b0;

   //-------------------------------------------------------------------
   //
   // Initialize the first element of the Skyline window to the
   // first element of Obj
   //
   //-------------------------------------------------------------------

   tempWindow[0] = obj[0]; //Copy first element of Obj to the Temp window
```

```
//------------------------------------------------------------------
//
// Assign next case to START
//
//------------------------------------------------------------------

skyline = START; // Set the next case to START

//------------------------------------------------------------------
//
// Print the number of elements in Obj and tempWindow
//
//------------------------------------------------------------------

numSky(nullVar,numElements); // Get the number of elements in the Skyline
totalSky = numElements; // The result is used for the loop counter

$display("totalSky = %d", totalSky);
$display("Value of loopCount = %d", loopCount);

numObj(nullVar,numElements); // Get the number of elements in Obj
totalObj = numElements;

//------------------------------------------------------------------
//
// Begin Skyline Computation
//
//------------------------------------------------------------------

while (loopCount <= (totalSky - 1)) begin // Loop to initiate case statement
   $display("LoopCount before compare equals = %d",loopCount);
   $display("totalSky before compare equals = %d",totalSky);

//------------------------------------------------------------------
//
// Skyline Case Statement
//
//------------------------------------------------------------------

case(skyline)

START:begin

  $display("Begin START");

  //------------------------------------------------------------------
```

```
  //
  // Print the the detailed contents of Obj and tempWindow
  //
  //-----------------------------------------------------------------

  // Display all elements of obj
  printObjDetail();

  //Display all elements of tempWindow
  printSkylineDetail();

  //-----------------------------------------------------------------
  //
  // Begin comparision. Set next case to Compare
  //
  //-----------------------------------------------------------------

  skyline = COMPARE; // Set the next case to COMPARE
  $display("End START");

  //-----------------------------------------------------------------
  //
  // Set compareBegin and firstCompare to "True". This is to reset
  // the loopCounter in order to invoke the COMPARE case.
  // The first element of tempWindow will then be compared to all
  // elements in Obj
  //
  //-----------------------------------------------------------------

  compareBegin = "True"; // Skyline computation will now begin
  firstCompare = "True"; // This marks the first comparison

end // End of START case

COMPARE:begin

  //-----------------------------------------------------------------
  //
  // Begin COMPARE
  //
  //-----------------------------------------------------------------

  $display("");
  $display("Start COMPARE");
  $display("");
```

```
    //------------------------------------------------------------------
    //
    // Loop through Obj and compare against values
    // in tempWindow[loopCount] (The element in skyline to be compared)
    //
    //------------------------------------------------------------------

    for (i = 0; i < objSize; i = i + 1) // Compare all elements of obj
                                        // to those in tempWindow
      begin
        $display ("i = %d in for loop. This is the next obj element to be compared.", i); //
Display index value
        $display ("");

        //------------------------------------------------------------
        //
        // Dont compare anything if the tempWindow element
        // is not a packet
        //
        //------------------------------------------------------------

        if (tempWindow[loopCount] != 56'b0) begin

        //------------------------------------------------------------
        //
        // If true, proceed with Skyline computation
        //
        //------------------------------------------------------------

        tempIP     = tempWindow[loopCount][55:24];
        tempPktLen = tempWindow[loopCount][23:8];
        tempOccur  = tempWindow[loopCount][7:0];

        objIP     = obj[i][55:24];
        objPktLen = obj[i][23:8];
        objOccur  = obj[i][7:0];

        //------------------------------------------------------------
        //
        // Display what is being compared
        //
        //------------------------------------------------------------

        $display("Next Compare");
        $display("TempWindow Element %d Obj Element %d",loopCount,i);
        $display("tempWindow = IP: %h PktLen: %d Occur:
```

```
%d",tempIP,tempPktLen,tempOccur);
        $display("");
        $display("obj = IP: %h PktLen: %d Occur: %d", objIP,objPktLen,objOccur);

        //------------------------------------------------------------
        //
        // If tempWindow element is less than Obj in both dimensions
        // then tempWindow element dominates. Ignore Obj element
        //
        //------------------------------------------------------------

        if (tempPktLen < objPktLen && tempOccur < objOccur)
          begin

            $display("Window element %d dominates obj element %d",loopCount,i);
            $display("Ignore current obj element");
            $display("");

          end
        //------------------------------------------------------------
        //
        // If tempWindow element is greater than Obj in both dimensions
        // then Obj element dominates. Discard tempWindow element
        // and add dominant Obj element to the Skyline (tempWindow)
        //
        //------------------------------------------------------------

        else if (tempPktLen > objPktLen && tempOccur > objOccur)
          begin

            memIndex = i;

            $display("Obj element %d dominated Window element %d",i,loopCount);
            $display("");
            $display("Removing Window Element %d Adding Obj %d",loopCount,i);

        //------------------------------------------------------------
        //
        // Remove the dominated element from the Skyline
        //
        //------------------------------------------------------------

            removeElement(tempWindow[loopCount]);
            incompare = "False"; // Dominant element is not incommparable
```

```
//-------------------------------------------------------------
//
// If the element doesn't already exist in the Skyline call
// addElement task. Otherwise the element already exists.
// Do nothing.
//
//-------------------------------------------------------------

     elemExists(obj[i],elementExists);

     if(elementExists == "False") begin

       addElement(obj[memIndex],loopCount,i,incompare);
       printSkySummary();

       end

     else if(elementExists == "True") begin

        $display("Element Exists = %s",elementExists);
        $display("Dominant obj element already in Skyline, do nothing!");
     end

  end

//-------------------------------------------------------------
//
// Condition if both elements in Obj and tempWindow are equal
//
//-------------------------------------------------------------

else if (tempPktLen == objPktLen && tempOccur == objOccur) begin

   $display("Obj element %d is equal to Window element %d",i,loopCount);
end

//-------------------------------------------------------------
//
// If element in Obj is better by atleast one dimension
// add to the Skyline anyways
//
//-------------------------------------------------------------

else if(tempPktLen > objPktLen || tempOccur > objOccur)
begin
```

```
//-------------------------------------------------------------
//
// If the element doesn't already exist in the Skyline call
// addElement task. Otherwise the element already exists.
// Do nothing.
//
//-------------------------------------------------------------

   elemExists(obj[i],elementExists);

   if(elementExists == "False") begin

      $display("Element %d in Obj was better in atleast one dimension",i);
      $display("Adding element to Skyline");
      incompare = "True"; //Dominant element is incomparable

      addElement(obj[i],loopCount,i,incompare);
      printSkySummary();

   end

  else if(elementExists == "True") begin

      $display("Element %d in Obj was better in atleast one dimension",i);
      $display("Dominant obj element already in Skyline, do nothing!");

   end

end

//-------------------------------------------------------------
//
// No conditions were matched. Display message
//
//-------------------------------------------------------------

else

    $display("Uh oh! None of the conditions are true");

//-------------------------------------------------------------
//
// After the first time through COMPARE set firstCompare to
// "False"
//
//-------------------------------------------------------------
```

```
        firstCompare = "False"; // First compare is no longer true

        end // End of if statement for packet existence

        //-------------------------------------------------------------
        //
        // If the element is not a packet. No comparison is performed.
        // Display message
        //
        //-------------------------------------------------------------

        else

          $display("Element %d of tempWindow is not a packet. No comparison
performed", loopCount);
          $display("");

    end // End of for loop. End of iteration.

        //-------------------------------------------------------------
        //
        // Display the value of totalSky (the condition to enter the
        // case statement)
        //
        //-------------------------------------------------------------

        numSky(nullVar,numElements);
        totalSky = numElements;

  end // end of COMPARE case

  endcase // End of case statement

  //-----------------------------------------------------------------
  //
  // Increment loopCount. Compare the next element in the
  // Skyline window
  //
  //-----------------------------------------------------------------

  loopCount = loopCount + 1; // Increment loop counter to compare next
                        // element in the Skyline
```

```
  //-----------------------------------------------------------------
  //
  // Restart loop counter if this is the first comparison
  //
  //-----------------------------------------------------------------

  if(compareBegin == "True" && firstCompare == "True")
    begin

      loopCount = 0;

    end

  //-----------------------------------------------------------------
  //
  // At the end of each iteration print the new summary results
  //
  //-----------------------------------------------------------------

  printSkySummary(); // Print the skyline after the iteration has ended

 end // End of while loop

  //-----------------------------------------------------------------
  //
  // COMPARE phase finished. Print final contents of Skyline
  //
  //-----------------------------------------------------------------

  $display("End COMPARE");
  $display("");
  $display("Start FINISH");
  $display("");
  $display("Skyline computation finished!");
  $display("");
  printSkylineDetail();

 end // End of module

//-----------------------------------------------------------------
//
// End module main
//
//-----------------------------------------------------------------

endmodule
```

Appendix B

```
//----------------------------------------------------------------
// Project: Skyline Block Nested Loop
// Author: Nathan Miller
// Date: 04/10/2013
// Filename: skytask.v
// Description: This file provides all the supporting methods for
//                      the Skyline computation.
//----------------------------------------------------------------

//----------------------------------------------------------------------
//
// This task calls obj_getsize() to determine the number of packets
// in obj
//
//----------------------------------------------------------------------

task numObj; // Returns the number of packets in an array

input testvar;
output [7:0] objElements;

integer a;

begin

objElements = 0;
counter = 0;

   // Loop passes each element of obj. Determines how many elements
   // (Packets) are in the memory buffer.
   for (a = 0; a < objSize; a = a + 1) // Loop passes each element of object
     begin
       obj_getsize(obj[a],addValue);
        objElements = objElements + addValue;
         //$display("Total Elements obj = %d", objElements);
         //$display("");
     end

end
endtask
```

```
//-----------------------------------------------------------------------
//
// This task calls obj_getsize() to determine the number of packets
// in tempWindow
//
//-----------------------------------------------------------------------

task numSky; // Returns the number of packets in an array

input testvar;
output [7:0] objElements;

integer b;

begin

objElements = 0;
counter = 0;

   // Loop passes each element of tempWindow. Determines how many elements
   // (Packets) are in the memory buffer.
   for (b = 0; b < objSize; b = b + 1) // Loop passes each element of object
     begin
        obj_getsize(tempWindow[b],addValue);
        objElements = objElements + addValue;
        //$display("Total Elements tempWindow = %d", objElements);
        //$display("");
     end

end
endtask

//-----------------------------------------------------------------------
//
// This task determines the number of elements in an object (Array)
// and whether the current element is a packet.
//
//-----------------------------------------------------------------------

task obj_getsize; // This task determines if an element is a packet. If
                  // so add 1 to the total number of elements otherwise
                  // add nothing.

input [55:0] a; // Element from object passed as argument
inout [7:0] addValue; // Value to be added to number of elements in object
```

```
begin

//$display("In task obj_getsize");
//$display("Element value: %h", a);

      // If there is data to be read and the array is not out of
      // bounds then add 1 to the number of elements.
      if (a != 56'b0 && counter < objSize)
        begin
          addValue = 1;
          counter = counter + 1;
        end

       // If there is no data in the element then don't add anything
       // to the number of elements.
       if(a == 56'b0 && counter < objSize)
        begin
          addValue = 0;
        end

end
endtask

//-------------------------------------------------------------------
//
// This task finds the current position of an element in the Skyline
//
//-------------------------------------------------------------------

task elemPos; // Scan the array to determine if a packet exists. If so
              // return its index value.

inout [55:0] a; // Check the object for the existence of this element.

output [7:0] objPos; // If the element exists this is its position
                     // in the stack.
integer c;

begin
   for(c = 0; c < objSize; c = c + 1)
      begin
```

```
        //-------------------------------------------------------------
        //
        // Loop through tempWindow to find the corresponding value
        // passed by Obj. If a match is found return the index value
        //
        //-------------------------------------------------------------
        if (a == tempWindow[c])
          begin

          objPos = c;

          end

      end
end
endtask

//-----------------------------------------------------------------------
//
// This task removes an element from the Skyline
//
//-----------------------------------------------------------------------

task removeElement;

input [55:0] a; // Remove element from the Skyline.

begin

elemPos(a,index); // Find the index value of the element in the Skyline

  //-------------------------------------------------------------
  //
  // If the value passed is equal to the value at the index of
  // tempWindow then initialize the element to all binary 0's
  //
  //-------------------------------------------------------------
  if (a == tempWindow[index]) // If the object from obj exists then
                              // discard the current value. Set the
                              // value to all zeroes
    begin

      tempWindow[index] = 56'b0;
      add = add - 1;
      $display("Add after remove = %d",add);
```

```
    end

  if(index == 0 && tempWindow[index+1] == 56'b0)

    begin

      add = 0;
      tempWindow[0] = obj[1];
      $display("The top skyline element was removed");

      if(obj[1] == obj[i]) begin

        $display("The next obj element %d has been added to the top of the Skyline",i);
        $display("Continue comparisons");

      end

    end

    moveElement(); // Shift elements in the skyline to a higher position
                   // in the list

end
endtask

//-------------------------------------------------------------------
//
// This task adds an element to the Skyline
//
//-------------------------------------------------------------------

task addElement;

input [55:0] a; // Element to be added to the Skyline.
input [7:0] loopCount; // Skyline element to be replaced
input [7:0] posElement;
input [39:0] incompare; // If the element was incomparable or not

reg [7:0] e;

begin

e = 0;
elemPos(tempWindow[loopCount],e);
numSky(nullVar,numElements);
add = numElements - 1;
```

```
add = add + 1;
$display("After adding add = %d",add);

    //-----------------------------------------------------------------
    //
    // Add the incomparable element to the Skyline
    //
    //-----------------------------------------------------------------

    if (incompare == "True" && e == loopCount) begin

      //-----------------------------------------------------------
      //
      // If the next index is larger than the boundaries of the array
      // then add the element to the current position in tempWindow
      //
      //-----------------------------------------------------------

      if (add >= objSize) begin

        tempWindow[e] = obj[posElement];
        $display("");
        $display("Added dominant element to %d",index);
        $display("");
        incompare = "False";

      end

      //-----------------------------------------------------------
      //
      // If the next index value is less than the boundaries of the
      // array then add the incomparable element to the next index
      // of tempWindow
      //
      //-----------------------------------------------------------

      else if (add <= objSize) begin

        tempWindow[add] = obj[posElement];
        $display("");
        $display("Added dominate element to %d",(add));
        $display("");
        incompare = "False";

      end
```

```
    end

    //-------------------------------------------------------------
    //
    // The element to be added is not incomparable. Add the element
    // to the current index of tempWindow
    //
    //-------------------------------------------------------------

    else if (incompare == "False" && e == loopCount)
      begin

        tempWindow[e] = obj[posElement];
        $display("");
        $display("Added dominant element to %d",(e));
        $display("");

      end

end
endtask

//---------------------------------------------------------------------
//
// This task prints the contents of obj
//
//---------------------------------------------------------------------

task printObjDetail;

integer f;

begin

    // Display all elements of obj
    $display("Begin START");
    $display("");
    $display("Contents of OBJ");
    $display("");
    $display("Element 0: %b",obj[0]);
    $display("Element 1: %b",obj[1]);
    $display("Element 2: %b",obj[2]);
    $display("Element 3: %b",obj[3]);
    $display("");
```

```
    for(f = 0; f < objSize; f = f + 1)
      begin

        objIP     = obj[f][55:24];
        objPktLen = obj[f][23:8];
        objOccur   = obj[f][7:0];

        $display("Element %d", f);
        $display("");
        $display("IP Address: %h", objIP);
        $display("Packet Length: %d", objPktLen);
        $display("Occurences: %d", objOccur);
        $display("");

      end

end
endtask

//---------------------------------------------------------------------
//
// This task prints the contents of the Skyline
//
//---------------------------------------------------------------------

task printSkylineDetail;

integer g;

begin

      $display("Contents of Skyline Window");
      $display("");
      $display("Element 0: %b",tempWindow[0]);
      $display("Element 1: %b",tempWindow[1]);
      $display("Element 2: %b",tempWindow[2]);
      $display("Element 3: %b",tempWindow[3]);
      $display("");

    for(g = 0; g < objSize; g = g + 1)
      begin

        tempIP     = tempWindow[g][55:24];
        tempPktLen = tempWindow[g][23:8];
        tempOccur   = tempWindow[g][7:0];
```

```
            $display("Element %d", g);
            $display("");
            $display("IP Address: %h", tempIP);
            $display("Packet Length: %d", tempPktLen);
            $display("Occurences: %d", tempOccur);
            $display("");

        end

end
endtask

//-------------------------------------------------------------------
//
// This task prints the contents of the Skyline
//
//-------------------------------------------------------------------

task printObjSummary;

begin

    // Display all elements of obj
    $display("Contents of OBJ");
    $display("");
    $display("Element 0: %b",obj[0]);
    $display("Element 1: %b",obj[1]);
    $display("Element 2: %b",obj[2]);
    $display("Element 3: %b",obj[3]);
    $display("");

end
endtask

//-------------------------------------------------------------------
//
// This task prints the contents of the Skyline
//
//-------------------------------------------------------------------

task printSkySummary;

begin

    // Display all elements of obj
    $display("Contents of Skyline Window");
```

```
    $display("");
    $display("Element 0: %b",tempWindow[0]);
    $display("Element 1: %b",tempWindow[1]);
    $display("Element 2: %b",tempWindow[2]);
    $display("Element 3: %b",tempWindow[3]);
    $display("");

end
endtask

//-----------------------------------------------------------------------
//
// Determine if an element in the Skyline already exists
//
//-----------------------------------------------------------------------

task elemExists;

input [55:0] a; // Determine if this element exists in the Skyline

output [39:0] elementExists;

integer h;

begin

for (h = 0; h < objSize; h = h + 1) begin
   if (a == tempWindow[h]) begin

      elementExists = "True";
      h = 5;

      end


    else if(a != tempWindow[h]) begin

      elementExists = "False";

     end

end

end
endtask
```

```
//----------------------------------------------------------------------
//
// If an element has been removed from the Skyline. The remaining
// elements shall be moved to a higher position in the list.
//
//----------------------------------------------------------------------

task moveElement;

integer t;

begin

for(t = 0; t < objSize; t = t + 1)
  begin

    if(tempWindow[t] == 56'b0 && tempWindow[t-1] != 56'b0 && tempWindow[t+1]
!= 56'b0 && t < objSize) begin

      tempWindow[t] = tempWindow[t+1];
      tempWindow[t+1] = 56'b0;
      $display("Moved tempWindow element from %d to %d",(t+1),t);

    end

    else if(tempWindow[t] == 56'b0 && tempWindow[t+1] != 56'b0 && (t-1) < 0) begin

      tempWindow[t] = tempWindow[t+1];
      tempWindow[t+1] = 56'b0;
      $display("Moved tempWindow element from %d to %d",(t+1),t);

    end

  end

end
endtask
```

Appendix C

```
//----------------------------------------------------------------
// Project: Skyline Block Nested Loop
// Author: Nathan Miller
// Date: 04/10/2013
// Filename: packetdata.v
// Description: Defines the data used for Skyline computation.
//----------------------------------------------------------------

task assignTestData;

begin

  //----------------------------------------------------------------
  // Uncomment the // in front of the variable assignment to use
  // the test condition
  //
  // Test 1
  //
  //
  // Statically assign the IP address, packet length,
  // and occurence values in binary. Pkt_Len is 16 bits wide,
  // occur is 8 bits. These values are stored in binary
  // to represent the signals being received from the wire.
  //
  // The contents are:
  //                      IP Address    Length      Occurences
  //
  //              obj[0] = 192.168.1.1     1200       70
  //              obj[1] = 192.168.1.2       15       75
  //              obj[2] = 192.168.1.1    10000       60
  //              obj[3] = 192.168.1.2       17       90
  //
  // The Final Skyline:       192.168.1.1     1200       70
  //                          192.168.1.2       15       75
  //                          192.168.1.1    10000       60
  //----------------------------------------------------------------

  obj[0] = 56'b11000000101010000000000100000001000001001011000001000110;
  obj[1] = 56'b11000000101010000000000100000001000000000000111101001011;
  obj[2] = 56'b11000000101010000000000100000001001001110001000000111100;
  obj[3] = 56'b11000000101010000000000100000001000000000001000101011010;
```

```
//-------------------------------------------------------------------
// Test 2
//
// The contents are:
//                      IP Address    Length      Occurences
//
//              obj[0] = 192.168.1.1     5000        61
//              obj[1] = 192.168.1.2     5001        75
//              obj[2] = 192.168.1.1    12000       100
//              obj[3] = 192.168.1.2     6000       101
//
// The Final Skyline:       192.168.1.1     5000        61
//-------------------------------------------------------------------

//obj[0] = 56'b11000000101010000000000100000001000100111000100000111101;
//obj[1] = 56'b11000000101010000000000100000001000010011100010010100101 1;
//obj[2] = 56'b11000000101010000000000100000001001011101110000001100100;
//obj[3] = 56'b11000000101010000000000100000001000010111011100000110010 1;

//-------------------------------------------------------------------
// Test 3
//
// The contents are:
//                      IP Address    Length      Occurences
//
//              obj[0] = 192.168.1.1     5000        80
//              obj[1] = 192.168.1.2     5001        75
//              obj[2] = 192.168.1.1    12000       100
//              obj[3] = 192.168.1.2     6000       101
//
// The Final Skyline:       192.168.1.1     5000        80
//                          192.168.1.2     5001        75
//-------------------------------------------------------------------

//obj[0] = 56'b11000000101010000000000100000001000100111000100001010000;
//obj[1] = 56'b11000000101010000000000100000001000010011100010010100101 1;
//obj[2] = 56'b11000000101010000000000100000001001011101110000001100100;
//obj[3] = 56'b11000000101010000000000100000001000010111011100000110010 1;
```

```
//----------------------------------------------------------------
// Test 4
//
// The contents are:
//                      IP Address    Length      Occurences
//
//              obj[0] = 192.168.1.1     5000        80
//              obj[1] = 192.168.1.2     5001        75
//              obj[2] = 192.168.1.1    12000         60
//              obj[3] = 192.168.1.2     6000        50
//
// The Final Skyline:       192.168.1.1     5000        80
//                          192.168.1.2     5001        75
//                          192.168.1.2     6000        50
//----------------------------------------------------------------

//obj[0] = 56'b11000000101010000000000100000001000100111000100001010000;
//obj[1] = 56'b11000000101010000000000100000001000010011100010010010011;
//obj[2] = 56'b11000000101010000000000100000001001011101110000000111100;
//obj[3] = 56'b11000000101010000000000100000001000010111011100000011001O;

//----------------------------------------------------------------
// Test 5
//
// The contents are:
//                      IP Address    Length      Occurences
//
//              obj[0] = 192.168.1.1     5000        80
//              obj[1] = 192.168.1.2     5001        75
//              obj[2] = 192.168.1.1     6100        40
//              obj[3] = 192.168.1.2     6000        50
//
// The Final Skyline:       192.168.1.1     5000        80
//                          192.168.1.2     5001        75
//                          192.168.1.1     6100        40
//                          192.168.1.2     6000        50
//----------------------------------------------------------------

//obj[0] = 56'b11000000101010000000000100000001000100111000100001010000;
//obj[1] = 56'b11000000101010000000000100000001000010011100010010010011;
//obj[2] = 56'b11000000101010000000000100000001000101111101010000101000;
//obj[3] = 56'b11000000101010000000000100000001000010111011100000011001O;

end
endtask
```

Bibliography

[1] P. Godfrey, R. Shipley, and J. Gryz, “Algorithms and analyses for maximal vector computation,” The VLDB Journal, vol. 16, no. 1, pp. 5–28, Sep. 2006.

[2] D. Papadias, Y. Tao, G. Fu, and B. Seeger, “An optimal and progressive algorithm for skyline queries,” In Proceedings of the 2003 ACM SIGMOD international conference on Management of data, 2003, p. 467.

[3] V. Chandola, A. Banerjee, and V. Kumar, “Anomaly detection: A survey,” ACM Computing Surveys, vol. 41, no. 3, pp. 1–58, Jul. 2009.

[4] M. Thottan, G. Liu, and C. Ji, “Anomaly Detection Approaches for Communication Networks,” in Algorithms for Next Generation Networks, G. Cormode and M. Thottan, Eds. London: Springer London, 2010, pp. 239–261.

[5] J. Rose, A. El Gamal, and A. Sangiovanni-Vincentelli, “Architecture of field-programmable gate arrays,” Proceedings of the IEEE, vol. 81, no. 7, pp. 1013–1029, Jul. 1993.

[6] N. Telle, W. Luk, and R. C. C. Cheung, “Customising Hardware Designs for Elliptic Curve Cryptography,” in Computer Systems: Architectures, Modeling, and Simulation, vol. 3133, A. D. Pimentel and S. Vassiliadis, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 274–283.

[7] W. John and T. Olovsson, “Detection of malicious traffic on back-bone links via packet header analysis,” Campus-Wide Information Systems, vol. 25, no. 5, pp. 342–358, 2008.

[8] G. Stitt, F. Vahid, and S. Nematbakhsh, “Energy savings and speedups from partitioning critical software loops to hardware in embedded systems,” ACM Transactions on Embedded Computing Systems, vol. 3, no. 1, pp. 218–232, Feb. 2004.

[9] I. Kuon, R. Tessier, and J. Rose, “FPGA Architecture: Survey and Challenges,” Foundations and Trends® in Electronic Design Automation, vol. 2, no. 2, pp. 135–253, 2007.

[10] U. Farooq, Z. Marrakchi, and H. Mehrez, “FPGA Architectures: An Overview,” in Tree-based Heterogeneous FPGA Architectures, New York, NY: Springer New York, 2012, pp. 7–48.

[11] A. Covington, “Guide • NetFPGA/netfpga Wiki • GitHub,” NetFPGA Wiki, Dec-2012. [Online]. Available: https://github.com/NetFPGA/netfpga/wiki/Guide. [Accessed: 30-Mar-2013].

[12] A. Covington, “InstallSoftware • NetFPGA/netfpga Wiki • GitHub,” NetFPGA Wiki, 2012. [Online]. Available: https://github.com/NetFPGA/netfpga/wiki/InstallSoftware#wiki-Install_CAD_Tools. [Accessed: 31-Mar-2013].

[13] G. Singh, F. Masseglia, C. Fiot, A. Marascu, and P. Poncelet, “Mining Common Outliers for Intrusion Detection,” in Advances in Knowledge Discovery and Management, vol. 292, F. Guillet, G. Ritschard, D. A. Zighed, and H. Briand, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 217–234.

[14] A. Goodney, S. Narayan, V. Bhandwalkar, and Y. H. Cho, “Pattern Based Packet Filtering using NetFPGA in DETER infrastructure,” 1st Asia NetFPGA Developers Workshop, Daejeon, Korea, June 14, 2010.

[15] Mentor Graphics, “ModelSim - Advanced Simulation and Debugging.” [Online]. Available: http://model.com/. [Accessed: 31-Mar-2013].

[16] G. Gibb, J. W. Lockwood, J. Naous, P. Hartke, and N. McKeown, “NetFPGA -- An Open Platform for Teaching How to Build Gigabit-Rate Network Switches and Routers,” IEEE Transactions on Education, vol. 51, no. 3, pp. 364–369, Aug. 2008.

[17] Mahoney, M., P. K. Chan, "PHAD: Packet Header Anomaly Detection for Identifying Hostile Network Traffic", Florida Tech. technical report 2001-04, http://cs.fit.edu/~tr/

[18] D. Seo, H. Lee, and A. Perrig, “PFS: Probabilistic filter scheduling against distributed denial-of-service attacks,” IEEE 36th Conference on Local Computer Networks (LCN), 2011, pp. 9–17.

[19] A. D. Sarma, A. Lall, D. Nanongkai, and J. Xu, “Randomized Multi-pass Streaming Skyline Algorithms,” PVLDB, vol. 2, no. 1, pp. 85–96, 2009.

[20] M. Goncalves and M.-E. Vidal, “Reaching the Top of the Skyline: An Efficient Indexed Algorithm for Top-k Skyline Queries,” in Database and Expert Systems Applications, vol. 5690, S. S. Bhowmick, J. Küng, and R. Wagner, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 471–485.

[21] T. J. Todman, G. A. Constantinides, S. J. E. Wilton, O. Mencer, W. Luk, and P. Y. K. Cheung, “Reconfigurable computing: architectures and design methods,” IEEE Proceedings - Computers and Digital Techniques, vol. 152, no. 2, p. 193, 2005.

[22] D. Kossman, F. Ramsak, and S. Rost, “Shooting Stars in the Sky: An Online Algorithm for Skyline Queries,” in Proceedings of 28th International Conference on Very Large Data Bases, Hong Kong, 2002, pp. 275–286.

[23] Xilinx, “Software and Design Tools,” ISE Design Suite. [Online]. Available: http://www.xilinx.com/products/design-tools/ise-design-suite/index.htm. [Accessed: 31-Mar-2013].

[24] N. Weaver, V. Paxson, and J. M. Gonzalez, “The shunt: an FPGA-based accelerator for network intrusion prevention,” in Proceedings of the 2007 ACM/SIGDA 15 International Symposium on Field Programmable Gate Arrays, 2007, pp. 199–206.

[25] S. Borzsony, D. Kossmann, and K. Stocker, “The Skyline Operator,” In Proceedings of 17th International Conference on Data Engineering, 2001, pp. 421–430.

[26]“NetFPGA - NetFPGA.” [Online]. Available: http://netfpga.org/index.html. [Accessed: 07-Apr-2013].

[27]“Xilinx University Program : Donation Program.” [Online]. Available: http://www.xilinx.com/university/donation/index.htm. [Accessed: 07-Apr-2013].

[28] G. R. Smith, FPGAs 101: Everything you need to know to get started. Amsterdam ; Boston: Newnes, 2010, pp. 43-54.

[29] I. Kuon and J. Rose, “Measuring the Gap Between FPGAs and ASICs,” IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 26, no. 2, pp. 203–215, Feb. 2007.

[30] U. A. Sandhu , S. Haider , S. Naseer , and O. U. Ateeb , “A Survey of Intrusion Detection & Prevention Techniques”, IPCSIT vol.16, IACSIT Press, Singapore 2011.

[31] M. Goncalves, D. Torres, and G. Perera, “An Evaluation Algorithm for Location Based Skyline Queries,” 2nd Workshop on Recommender Systems meets Databases in conjunction with the 23rd DEXA Conference, September 3 - 7, 2012, Vienna, Austria.

[32] R. Mueller, J. Teubner, and G. Alonso, “Streams on wires: A Query Compiler for FPGAs,” Proceedings of VLDB Endowment, vol. 2, no 1, pp 229-240, 2009.

[33] D. Bacon, R. Rabbah, and S. Shukla, “FPGA Programming for the Masses,” ACM Queue, vol 2, pages 40-53, 2013.