A Hardware-Based Clustering Approach for Anomaly Detection

Khaled Labib and V. Rao Vemuri

Department of Applied Science, University of California, Davis, California, U.S.A.
Email: {kmlabib, rvemuri}@ucdavis.edu

Abstract

Several clustering methods have been developed for clustering network traffic in order to detect traffic anomalies and possible intrusions. Many of these methods are typically implemented in software. As a result they suffer performance limitations while processing real time traffic. This study presents a hardware implementation of the *k*-means clustering algorithm that is used to cluster network traffic. The implementation uses the Verilog hardware description language to build a circuit to read packet information from system memory and produce the output cluster assignments in a 32-bit register. After reset is applied, the circuit uses a state-machine that represents the *k*-means algorithm to process IP packets for a fixed number of iterations and then generates an interrupt to indicate that it had finished processing the data. The implementation is synthesized into a Field Programmable Gate Array in order to study the number of gates required for the implementation. The maximum achievable clock cycle without applying timing constraints is 40 MHz. To compare the performance of this implementation with a software-based implementation, a C version of the *k*-means algorithm is compiled, run and profiled with similar parameters of the hardware-based implementation. The results show that the performance of the hardware-based implementation is approximately 300 times faster than a software-based implementation.

1. Introduction

The field of anomaly detection in computer network security is rich with many methods that are devised and implemented to enable the detection of anomalous traffic and possible intrusions. Anomaly detection attempts to identify anomalies in the network traffic that suggest a possibility of an attack or intrusion that is taking place. This is achieved by establishing what the normal traffic patterns look like for a given network, and flagging out any variations in traffic from this norm.

At the core of most anomaly detection systems are a set of algorithms that attempt to cluster the input traffic data into a number of output clusters. This clustering process is needed to separate normal traffic from anomalous traffic, which in turn may include intrusive traffic. Several algorithms have been used including *k*-means [1], hierarchical clustering [2], Self-Organizing Maps [3] and Principal Component Analysis [4] as the core algorithms for anomaly detectors. However, these implementations suffer performance penalties as they are typically designed and implemented as software programs running on a host machine and clustering the packets as they arrive. There are several disadvantages to this software-based implementation. First, the performance of these software-based methods does not scale well to meet the requirements of real time processing. This performance limitation is true because of the limited processing power of the hosts running the software in terms of instructions per second (IPS) that can be executed by the Central Processing Units (CPU) in these hosts. When run as software programs, these algorithms are typically compiled from a high-level language to some object format that the host CPU executes. There is an upper limit on the number of IPS these CPUs can execute. Second, the processing of these algorithms places a burden on the host CPU running them, thereby hogging the host CPU bandwidth when processing other tasks. One implication of this fact is that it requires a dedicated host machine that serves as an anomaly detector node on every network segment in order not to burden others hosts running the main applications on that network segment which in turn increases the budget for building secure networks. Third, to meet the performance goals required for processing a large number of packets, the clock frequency for these CPUs must be increased, thereby increasing the cost of the systems and consuming more power as the frequency is increased. For example, performing full system monitoring

including detailed behavior such as memory references made by all applications and the operating system including network monitoring can be performed with slowdowns of roughly 10X [5].

To address some of the performance drawbacks of software-based clustering methods for network anomaly detection, this study presents a hardware-based clustering circuit based on the *k*-means clustering algorithm. The circuit is developed using synthesizable Verilog Hardware Description Language (HDL) [6]. The circuit can be viewed as a back-end hardware-assisting block that performs the clustering of network data once started and interrupts the host CPU when finished. Therefore it is designed to replace the functionality of a software-based *k*-means clustering algorithm by implementing the core function in hardware to attain much higher performance. This relieves the host CPU from performing the computational intensive task of clustering the data and allows it to handle other system tasks without hogging its bandwidth. The circuit consists of a clock and reset pins at the input and the cluster assignment and an interrupt pin at the output. The circuit is tested in a simulation environment by constructing a Verilog test bench that supplies the clock and reset. The circuit processes 32 packets at a time, and generates an output cluster assignment for each packet along with an interrupt. This generated interrupt indicates that the clustering process has finished and that the cluster assignments are ready to be read and further processed by the CPU.

After testing the circuit functionality, it is synthesized using an FPGA (Field Programmable Gate Array) flow to create a physical implementation. When synthesized with no timing constraints, the maximum clock frequency of 40 MHz can be achieved. Higher clock frequencies are achievable by applying timing constraints during the synthesis process on the expense of more gates and therefore more die area.

The rest of the paper is organized as follows: Section 2 presents related work in the field of anomaly detection using hardware accelerators. Section 3 presents the details of the circuit implementation in Verilog. Section 4 discusses the results obtained using the circuit in clustering test data and selected attack data from the 1998 DARPA intrusion detection data sets. Section 5 presents the synthesis results of the design in an FPGA flow. Section 6 compares the results obtained using this hardware implementation to a software implementation of the same algorithm in terms of performance. Finally, section 7 presents the conclusion and future work.

2. Related Work

Software implementations of the *k*-means algorithm for anomaly detection exist in the literature [7]. However, there were no attempts to employ a hardware-based clustering algorithm for anomaly detection similar to the work reported in this study.

Nevertheless, few hardware implementations of the *k*-means algorithm have been used in the area of video and image processing.

Estlick et al [8] use algorithm level transforms to map the *k*-means algorithm into an FPGA and apply it to multi-spectral and hyper-spectral images having tens of hundreds of channels per pixel of data. They examine basically two algorithm level transforms. First, they studied using the Manhattan and Max distance measures that do not require multipliers. Second, they examined the effects of using single precision and truncated bit width in the algorithm. The algorithm is mapped to a reconfigurable hardware. Their implementation resulted in speedups by a factor of about 200 over a software implementation.

Filho et al [9] implemented a parameterized *k*-means algorithm for clustering hyper-spectral images in a hardware/software co-design approach. Two models, a software and a hardware/software co-design ones, have been implemented. Although the hardware component operates in 40MHz, being 12.5 times lesser than the software operating frequency (PC), the co-design implementation was approximately 2 times faster than software one.

While the above-cited works differ from this study primarily from an application point of view, there are other differences in terms of the hardware implementation details. The implementation done in this study

does not use co-design methodologies. The work done by Estlick et al focuses on the efficiency of implementation of the algorithm for handling a large number of pixel data using the VHDL language. While this approach may be suitable for clustering large amounts of data and where optimal implementation is sought, the implementation presented in this study deals with much less data for clustering, thereby lending itself to a simpler design that synthesizes directly from Verilog HDL. In hardware/software co-design a portion of the algorithm is implemented in software while the remaining computational intensive portion of the algorithm is implemented in hardware. A co-design approach has its merits and demerits but it eventually places some computational burden on the host CPU. This unnecessary burden is avoided using the simple structured implementation in this study.

3. Circuit Implementation

The *k*-means algorithm is a well-structured algorithm and therefore is suitable for implementation as a hardware circuit. The basic algorithm can be summarized in the following steps:

1) Take as input the number of clusters (*k*), the number of iterations to run and the input data set. In this study $k = 2$.
2) Create the initial *k* cluster centers by choosing *k* data points (records) randomly from the data set.
3) Calculate the arithmetic mean of each cluster.
4) Assign each record from the input data set to the nearest cluster using some distance measure like Euclidean or Manhattan distance measure. This study uses Manhattan distance measure for reasons explained below.
5) When all records are assigned, re-calculate the arithmetic mean of each cluster. This new arithmetic mean is the center of a new cluster.
6) Assign each record from the input data set to the nearest new cluster center.
7) Repeat steps 4 to 6 six above until stable clusters are formed or for a fixed number of iterations.

The circuit block diagram in Figure 1 shows the main components of the circuit that implements the *k*-means algorithm. The circuit has two input signals namely: clk (input clock) and reset. The clk signal supplies a clock of 40 MHz and the reset signal resets all internal state machines and registers. The outputs of the circuit are a 32-bit bus carrying cluster assignments and an interrupt pin that can be used to interrupt a host CPU to indicate that the clustering process is finished and that the cluster assignments are valid. Each of the 32 input packets is assigned a value of either zero or one indicating one of the two clusters it can be assigned. In addition the circuit reads the 32-input packet header information from a [192x32] bit memory array.

When the clock is running and after reset has been de-asserted, the main FSM (Finite State Machine) assigns the next state variable to one of the following values. This parameter declaration uses Verilog syntax where 3'b means a 3 bit binary value:

```
parameter   IDLE                = 3'b000,
            SELECT_CLUST_CENTERS = 3'b001,
            CALC_DIST_CENTER0   = 3'b010,
            CALC_DIST_CENTER1   = 3'b011,
            ASSIGN_ROW_TO_CLUST = 3'b100,
            CALC_NEW_CENTER     = 3'b101,
            DUMMY               = 3'b110,
            CALC_CENTER_AVG     = 3'b111;
```

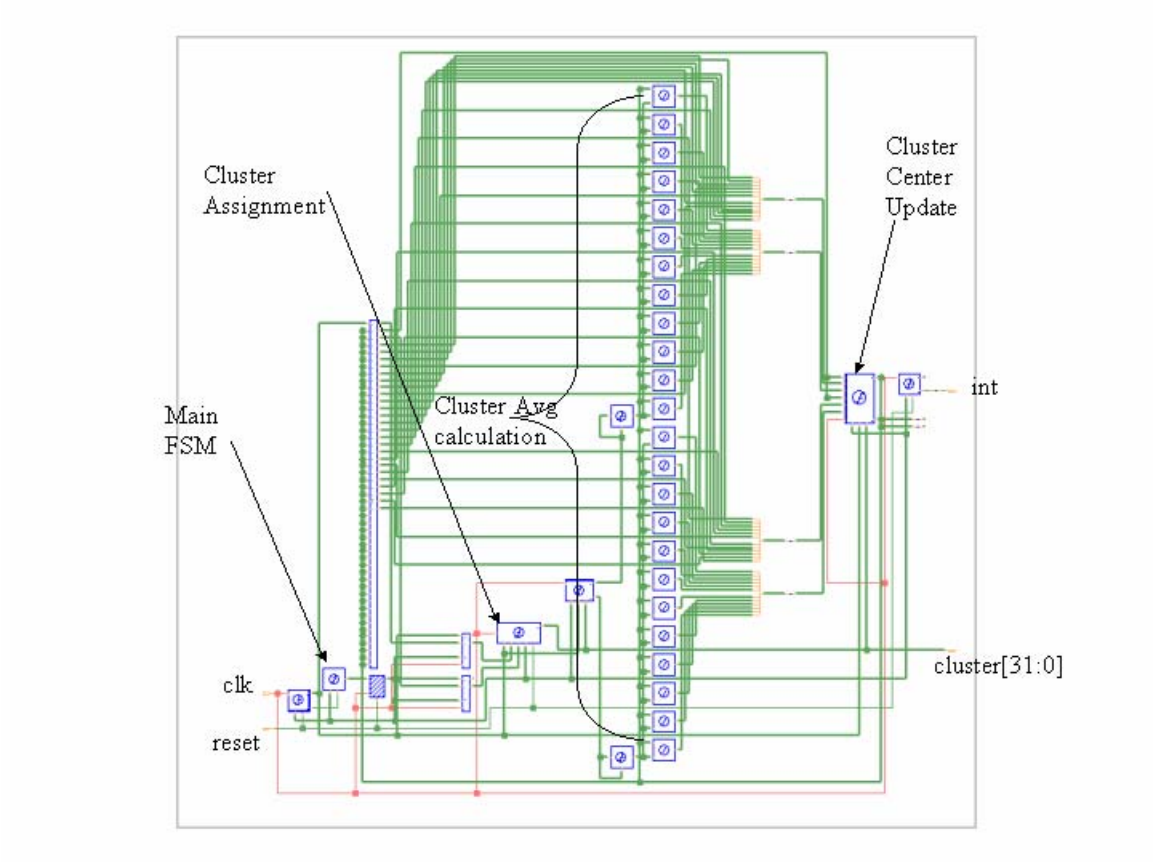

Figure 1: Block Level Diagram of the *k*-means circuit

The structure of the main FSM is depicted in Figure 2 which shows the different states the FSM can be in. At reset, the current state is assigned to `IDLE`. On the following positive clock edge after reset is de-asserted, the next state assigned is `SELECT_CLUST_CENTERS`. In this state, two data points are selected to form the input data to form the initial cluster centers and the next state is assigned to `CALC_DIST_CENTER0`. In this state the distance between all input data points and first cluster center (center 0) is calculated. Since it is difficult to implement the square root function in hardware (needed in Euclidean distance calculations), the Manhattan distance measure is used instead. The Manhattan distance, also known as the L_1-distance or City Block distance, is the distance between two points measured along axes at right angles. In a plane, the Manhattan distance between the point P_1 with coordinates (x_1, y_1) and the point P_2 at (x_2, y_2) is

$$| x_1 - x_2 | + | y_1 - y_2 |$$

Using the Manhattan distance yields less optimal clusters than the Euclidean distance but provides for a more efficient hardware circuit implementation. Theiler et al [10] examined the *k*-means method using Manhattan distance and Max distances as well as a linear combination of the two in a fixed-point hardware implementation. They could fit more distance-computation nodes on their chip, obtain a higher degree of parallelism and therefore faster performance but at the price of slightly less optimal clusters.

The `CALC_DIST_CENTER0` state takes 32 clock cycles before it advances to the next state. At each of the 32 clock cycles, the value of the row pointer of the input data array is advanced by one. The row pointer is incremented by one at each clock to point to the next row in the input memory array. Similar calculations are done when the FSM advances to state `CALC_DIST_CENTER1`.

In state `ASSIGN_ROW_TO_CLUST` each row of the input data is assigned to one of the two clusters based on its distance to the center of the cluster. In state `CALC_NEW_CENTER` the new cluster centers are calculated based on the assignments of each input row to one of the two clusters. In this state the sum of the clusters is calculated by adding up the values of the components of rows belonging to each center. After the new centers are calculated, the state machine advances to the DUMMY state which, as its name reflects, provides no calculations but is added as a staging state to ensure that the values of the flip flops of the previous state are stable. In state `CALC_CENTER_AVG` the arithmetic mean of each cluster center is calculated by dividing the cluster sums obtained in state `CALC_NEW_CENTER` by the number of rows in each cluster. After calculating the new cluster centers, the state machine jumps back to state `CALC_DIST_CENTER0` to re-calculate the distance of each input row to the new cluster centers. This process is repeated a number of times defined by the constant NUM_OF_ITERATIONS which is set to eight in this implementation. This value of eight reflects the typical default number of iterations in a software implementation of the algorithm. When the desired number of iterations is reached the circuit triggers an interrupt to indicate that the calculations are finished. In a full system implementation this interrupt would be directed to a host CPU that reads the clusters assignments output of the algorithm from a 32-bit register. Each of the 32 input packets would be assigned a value of either zero or one in this 32-bit register output. All packets that are similar would be clustered together in one of the two clusters.

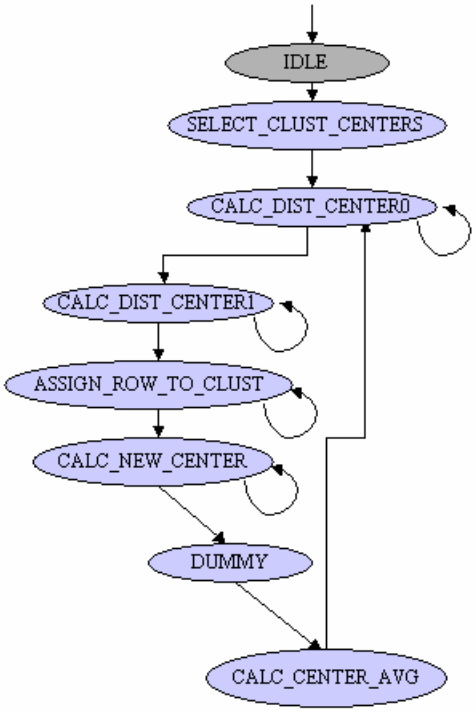

Figure 2: Main Finite State Machine structure for the design

4. Results

The circuit described in Section 3 was tested using both synthetic test data and real network traffic data. The synthetic test data was used during the development phase of the circuit to create a simple test bench where the input data rows contain numbers that can easily be manipulated by the *k*-means implementation

and later studied. After the algorithm was stable, real network traffic data was tested using attack data sets from the 1998 DARPA Intrusion Detection Evaluation data sets [11]. These data sets represent four attack types namely: Smurf, Neptune, IPsweep and Portsweep.

Smurf attacks, also known as directed broadcast attacks, are a popular form of denial-of-service packet floods. Smurf attacks rely on directed broadcast to create a flood of traffic for a victim. The attacker sends a ping packet to the broadcast address for some network on the Internet that will accept and respond to directed broadcast messages, known as the Smurf amplifier. These are typically mis-configured hosts that allow the translation of broadcast Internet Protocol (IP) addresses to broadcast Medium Access Control (MAC) addresses. The attacker uses a spoofed source address of the victim. For Example, if there are 30 hosts connected to the Smurf amplifier, the attacker can cause 30 packets to be sent to the victim by sending a single packet to the Smurf amplifier [12].

Neptune attacks can make memory resources too full for a victim by sending a TCP packet requesting to initiate a TCP session. This packet is part of a three-way handshake that is needed to establish a TCP connection between two hosts. The SYN flag on this packet is set to indicate that a new connection is to be established. This packet includes a spoofed source address, such that the victim is not able to finish the handshake but had allocated an amount of system memory for this connection. After sending many of these packets, the victim eventually runs out of memory resources.

IPsweep and Portsweep, as their names suggest, sweep through IP addresses and port numbers for a victim network and host respectively looking for open ports that could potentially be used later in an attack.

Most of the current inexpensive home and small-office routers in addition to high-end ones employ mechanisms to detect the selected attacks and a dozen other common attacks, many of which exist in the DARPA data sets. These attacks are becoming simple to detect using stateful packet inspection (SPI) techniques. The use of the selected attacks in this study is mainly to demonstrate the process of clustering for anomaly detection where the clustering engine attempts to explore the relationships within the multidimensional input packet data.

A Verilog test bench is constructed to provide the clock, assertion and de-assertion of reset and to initialize the 144x32 bit array of input data.

4.1. Input Data Format

The following data format was used to create the input feature vectors representing data to be clustered. The same format was used for both synthetic test data and selected DARPA test data. To create the four sets of test data from the DARPA data sets, the data sets were preprocessed by extracting the IP packet header information to create feature vectors. The resulting feature vectors were used as input to the circuit implementing the *k*-means algorithm. The feature vector chosen has the following format:

48 bit	12 bit	48 bit	12 bit	12 bit	12 bit
SIPx	SPort	DIPx	Dport	Prot	Plen

Where

- SIP*x* = Source IP address nibble, where *x* = [1-4]. Four nibbles constitute the full source IP address
- SPort = Source Port number
- DIP*x* = Destination IP address nibble, where *x* = [1-4]. Four nibbles constitute the full destination IP address
- DPort = Destination Port number
- Prot = Protocol type: TCP, UDP or ICMP
- PLen = Packet length in bytes

This format represents the IP packet header information. Each feature vector has 12 components. Each component is represented by 12-bits. Therefore, each input row is represented by a 144 bit value. A total of

32 rows are processed at a time by the circuit. The IP source and destination addresses are broken down to their network and host addresses to enable the analysis of all types of network addresses.

Figure 3 shows the output cluster assignments using the synthetic test data as input. The input data in this case consists of interleaved data where similar data rows appeared in every other row of the 32 input rows. It can be seen, using signal “cluster” in the figure, that the circuit clustered the output as 0x55555555 which is a 32-bit value with zeros and ones alternating in the sequence. This simply means that the first row was assigned to cluster one, the second row to cluster zero, the third row to cluster one, and so on.

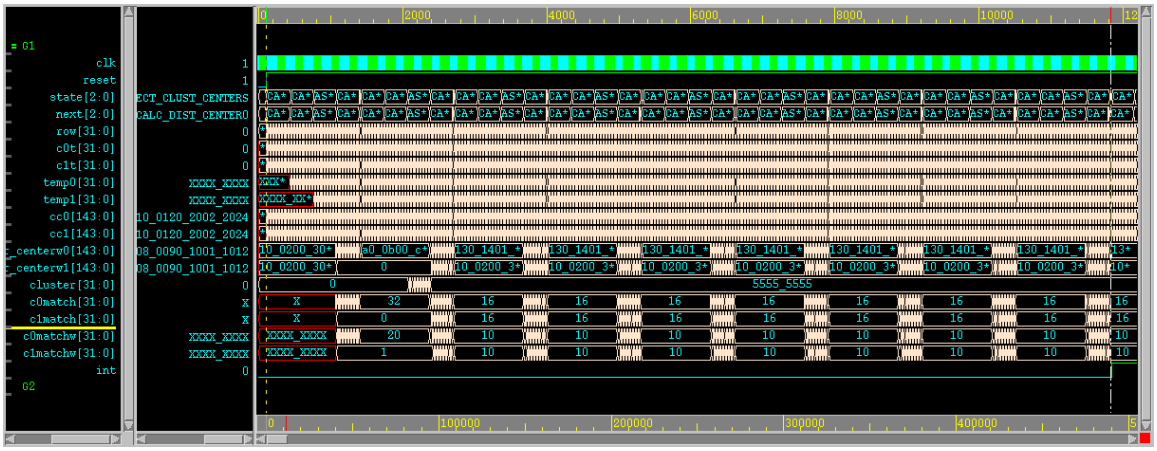

Figure 3: Signal Waveforms for Synthetic Input Data

In the above figure, the final cluster assignment is shown by the signal “cluster” (The third signal above the yellow marker on the left part of the figure). The final value of this signal when the interrupt was generated by signal “int” is 0x55555555 indicating the 32 cluster assignments of the input rows. Signals “c0match” and “c1match” indicate the number of input rows assignments to clusters zero and one respectively. The final values of these signals were both 16, indicating that both clusters zero and one had 16 members each, totaling to 32 input rows.

Figure 4 shows output cluster assignment for the Smurf attack data. The 32 input rows for this data set consist of normal traffic for the first 20 rows after which the Smurf attack packets start for 12 additional rows.

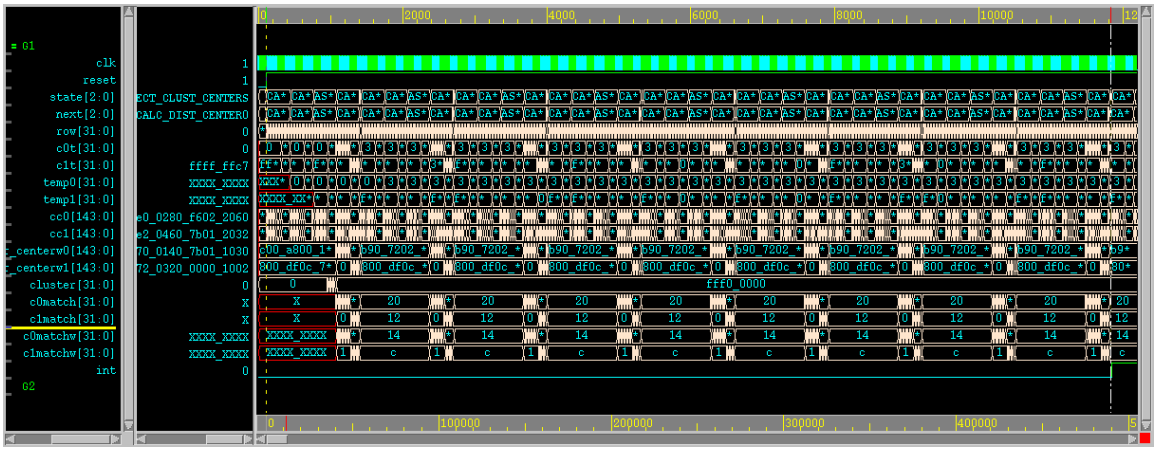

Figure 4: Signal Waveform for Smurf Attack Data

In the above figure, the final cluster assignment is shown by signal “cluster” with a value of 0xfff00000. This value represents the first 20 packets (rows) of normal traffic as being assigned to cluster zero, while the last 12 packets (rows) of Smurf attack traffic as being assigned to cluster one. In addition, the final values of signals “c0match” and “c1match” when the interrupt is generated are 20 and 12 respectively.

The above results suggests that the circuit is able to cluster the input data correctly by applying both synthetic input data and real packet data extracted from the DARPA data sets.

Similar results were obtained using Neptune and IPsweep data sets. The data sets have a similar structure as the Smurf data set where first 20 rows included normal traffic while the remaining 12 rows include traces of each attack respectively. The results of using these sets are shown in Figure 5 and Figure 6.

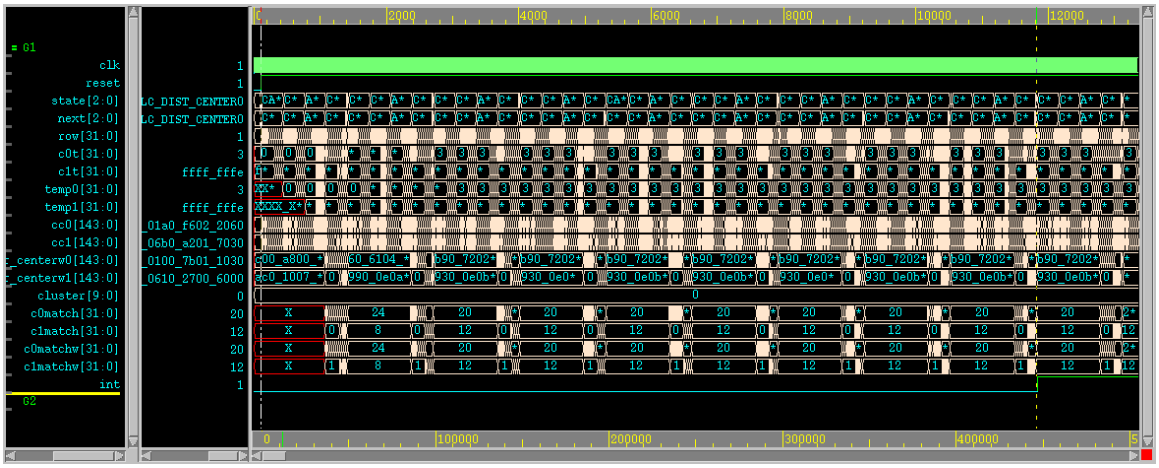

Figure 5: Signal Waveform for Neptune Attack Data

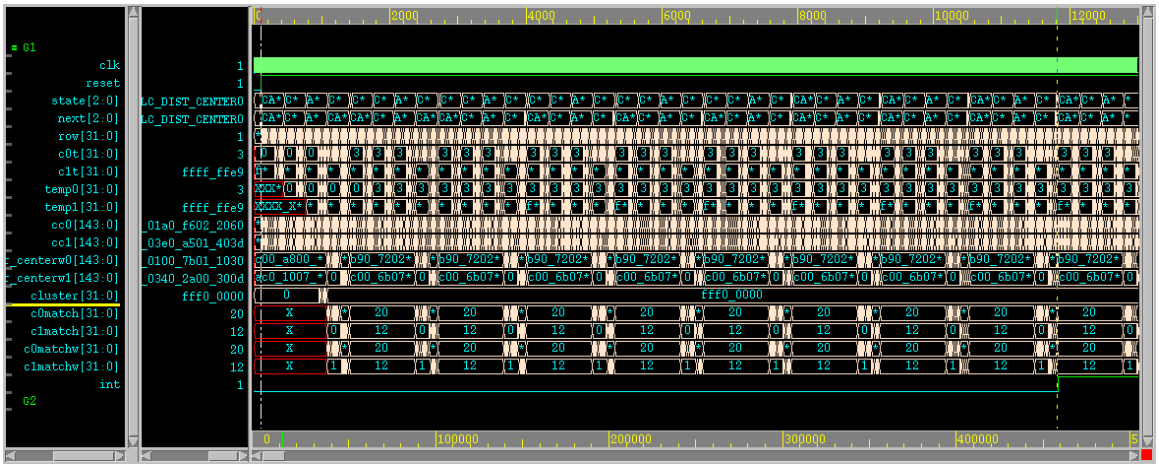

Figure 6: Signal Waveform for IPsweep Attack Data

For the Portsweep data set, the results were slightly different. Even though similar format as previous data sets was used, the results indicate that the first 27 rows were assigned to one cluster while the remaining 5 rows were assigned to a different cluster. These results are shown in Figure 7.

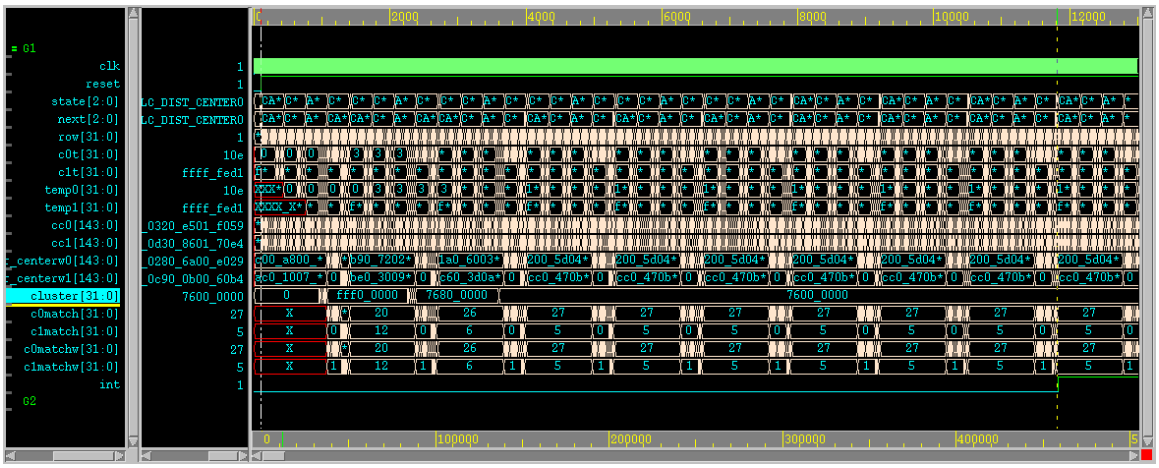

Figure 7: Signal Waveform for Portsweep Attack Data

The above Portsweep results can be explained by the small size of the input data set where the clustering process maybe impaired by the small number of observations. In a different experiment with the Portsweep data set the number of rows containing the attack data were increased to 16. In this case the algorithm was able to correctly classify the data. Another approach would be to increase the number of observations that the algorithm operates on from 32 observations to possibly 64. The primary disadvantage of such approach would be the drastic increase in the size of the design in terms of gate count, where the size of the design will roughly double.

4.2. Circuit Execution Time

The clk signal is a free running clock that is synchronously used in the design. Careful examination of Figure 4 reveals the time it took the algorithm to finish all iterations. The time is shown in nano-seconds in the top portion of the figure. When the interrupt is generated (white marker on the right) the time indicates that 11,800 nano-seconds have elapsed indicating the total time it took to finish the clustering process.

5. Design Synthesis

The circuit described in section 3 was synthesized using an FPGA synthesis flow to determine the details of the physical implementation and to ensure that the design is synthesizable.

The Xilinx ISE (Integrated Synthesis Environment) was used to synthesize the design. In addition the Xilinx Project Navigator was used to load and control the synthesis parameters. The following data was obtained as part of the design synthesis process:

Finite State Machines (FSMs)	**1**
RAMs	**2**
32x32-bit single-port distributed RAM	2
Registers	36
1-bit register	34
144-bit register	2
Counters	**2**
32-bit up counter	1
16-bit up counter	1
Multiplexers	**2**
1-bit 32-to-1 multiplexer	1
144-bit 32-to-1 multiplexer	1
Adders/Subtractors	**70**
12-bit adder	24

14-bit subtractor	24
14-bit adder	2
15-bit adder	4
16-bit adder	8
18-bit adder	2
17-bit adder	6
Comparators	**2**
32-bit comparator lessequal	1
32-bit comparator greater	1

Table 1 : Design Synthesis Results

As shown in Table 1, a FSM is synthesized which controls the different states described in section 3. The two 32 x 32 bit RAM represent the memory required to hold the distance of each row to each of the two cluster centers. The 34 1-bit registers hold the cluster assignment for each row. The two 144-bit registers each holds the cluster center values at any time during the process of clustering. The counters are used to hold incrementing row values since each row is processed in a single clock cycle. The multiplexers, adders and subtractors are used in calculating the Manhattan distance measures and the averages. Finally, the comparators are used to compare the distance values when assigning a cluster to each row.

The total equivalent gate count for the design is 58,484 gates. To compare the size of this circuit to other components of a modern system-on-a-chip (SoC), a typical 32-bit Reduced Instruction Set Computer (RISC) CPU is roughly 800,000 gates, where a PCI controller is roughly 150,000 gates. A small component such as a Timer/Counter which features an interval timer, a pulse generator and a watch-dog timer is roughly 5000 gates. Therefore the size of the *k*-means design is relatively large given that it serves as a hardware-assist circuit. In addition, to configure the *k*-means design to process more than 32 packets at a time will cause the circuit to grow even bigger in terms of number of gates, making it expensive to manufacture for mass production.

6. Comparison between Hardware and Software Implementations

In order to compare the results obtained by the hardware implementation of the *k*-means algorithm to a software implementation in terms of performance, a software implementation was created, run and profiled. The basic *k*-means algorithm code in C was configured to process 32-packet data and was run on a Sun Ultra2 200MHZ machine running Solaris 5 operating system. A generic GCC compiler was used to compile the C code with a –gprof option to create a profile of the code. The number of iterations of the algorithm was set to NUM_OF_ITERATIONS to mimic that of the hardware implementation. Next the algorithm was run to process the input data and the following data was collected in the output profile:

% Time	**Cumulative Seconds**	**Self Seconds**	**Self Calls**	**Total us/call**	**us/call**	**Name**
75.00	0.003	0.003	8	37.500	37.500	CalculateMidPoints
25.00	0.004	0.001	8	12.500	12.500	CalculateDistances
0.00	0.004	0.000	1	0.00	0.00	Initialize
0.00	0.004	0.000	1	0.00	4000.00	Kmeans
0.00	0.004	0.000	1	0.00	4000.00	Main

Table 2 : Results of profiling the k-means implementation in C

Where:

- **% Time**: the percentage of the total running time of the program used by this function.
- **Cumulative Seconds**: a running sum of the number of seconds accounted for by this function and those listed above it.
- **Self Seconds**: the number of seconds accounted for by this function alone.

- **Self Calls**: the number of times this function was invoked, if this function is profiled, else blank.
- **Total us/call**: the average number of microseconds spent in this function and its descendents per call, if this function is profiled, else blank.
- **us/call**: the average number of microseconds spent in this function per call, if this function is profiled, else blank.
- **Name**: the name of the function.

As shown in Table 2 above, the C implementation of the *k*-means algorithm spent most of the time in two functions, namely "calculateMidPoints" and "calculateDistances". The total amount of time spent in both functions is 4000 microseconds. The hardware implementation of the same algorithms can process the same number of packets and iterations in a total of 11.8 microseconds at 40MHZ clock speed. The hardware implementation is over 300 times faster than the software implementation leading to a much greater performance when processing real time packets.

7. Summary and Future Work

This study presented a hardware-based implementation of the *k*-means algorithm that is used in network anomaly detection to cluster network packets. The circuit is designed to replace the functionality of a software-based *k*-means algorithm by implementing the core function in hardware to attain much higher performance. The circuit consists of a clock and reset pins at its input and a 32-bit cluster value and an interrupt pin at the output. A Finite State Machine controls the transition of the different states in the circuit. The circuit can process 32 packets at a time. The processing is initiated by asserting the reset signal and is completed by the assertion of an interrupt. The design is synthesized to run at a clock frequency of 40 MHZ at which it processes 32 packets in 11.8 microseconds. The performance of the design is compared with a software-based implementation of the *k*-means algorithm with similar parameters and was found to be over 300 times faster. Future work includes optimizing the implementation to yield a smaller die area and enhance its capability to process a larger number of packets at a time.

8. References

1. Shi Zhong, Taghi M. Khoshgoftaar, and Naeem Seliya. Evaluating Clustering Techniques for Unsupervised Network Intrusion Detection. *International Journal of Reliability, Quality, and Safety Engineering*, 2005.
2. Khaled Labib, V. Rao Vemuri, "Application of Exploratory Multivariate Analysis for Network Security", CRC Press, 2005
3. Rhodes B., Mahaffey J., Cannady J., "Multiple Self-Organizing Maps for Intrusion Detection". *Proceedings of the NISSC 2000 conference*, Baltimore M.D. 2000.
4. Shah H., Undercoffer J., Joshi A., "Fuzzy Clustering for Intrusion Detection". *FUZZ-IEEE*, 2003
5. David L. Oppenheimer and Margaret R. Martonosi., "Performance Signatures: A Mechanism for Intrusion Detection". *Proceedings of the 1997 IEEE Information Survivability Workshop*, 1997.
6. Samir Palnitkar, "Verilog HDL : A Guide to Digital Design and Synthesis". Prentice Hall, 1996.
7. Shi Zhong, Taghi M. Khoshgoftaar, and Naeem Seliya, "Evaluating Clustering Techniques for Network Intrusion Detection". In *10th ISSAT Int. Conf. on Reliability and Quality Design*, pp. 149-155. Las Vegas, Nevada, USA. August 2004.
8. Mike Estlick, Miriam Leeser, James Theiler, John J. Szymanski, "Algorithmic Transformations in the Implementation of K-means Clustering on Reconfigurable Hardware". International Symposium on Field Programmable Gate Arrays. Proceedings of the 2001 ACM/SIGDA ninth international symposium on Field programmable gate arrays.
9. Abel Guilhermino da S. Filho, Alejandro C. Frery , Cristiano Coêlho de Araújo , Haglay Alice, Jorge Cerqueira, Juliana A. Loureiro, Manoel Eusebio de Lima , Maria das Graças S. Oliveira , Michelle Matos Horta, "HYPERSPECTRAL IMAGES CLUSTERING ON RECONFIGURABLE HARDWARE USING THE K-MEANS ALGORITHM". 16th Symposium on Integrated Circuits and Systems Design (SBCCI'03), September 08 - 11, 2003, São Paulo, Brazil.

10. James Theiler, Leeser M., Michael Estlick, and John J. Szymanski, *"Design Issues for Hardware Implementation of an Algorithm for Segmenting Hyperspectral Imagery,"* Image Spectrometry VI, Proceedings of SPIE Vol 4132, July 2000, pp 99-106.
11. DARPA Intrusion Detection Evaluation Project: http://www.ll.mit.edu/IST/ideval/
12. Skoudis E., *"Counter Hack: A Step-by-Step Guide to Computer Attacks and Effective Defenses"*. Prentice Hall Inc., 2002