Reconfigurable Processing Architectures for Stream Processing and Hybrid Computing

Janardhan Singaraju, Ph.D.

University of Connecticut, 2011

High performance computing systems are often inhibited by the performance of their storage system and their ability to deliver data. In stream processing of data, Active Storage Networks (ASN) provide an opportunity to optimize storage system and computational performance by offloading some computation to the network switch. Data processing in a distributed system often requires the data to be aggregated at a single client before performing the data operation. An implementation of this processing in the interconnection network which has the global view of the data could speed up the application. An ASN is based around an intelligent network switch that allows data processing to occur on data as it flows through the storage area network from storage nodes to client nodes. We propose an approach to perform transformation and reduction data operations in an intelligent network switch comprised of FPGAs. A low cost non blocking 2-dilated flattened butterfly interconnection network is chosen for prototype im-

plementation of ASN. Common data processing applications, namely data sort, data search, k-min/max and K-means clustering applications have been implemented on the switching elements of this network. The scalability of the ASN in performing data processing applications is evaluated by applying functional and data parallel techniques to the K-means clustering problem. The implementations show that the in-network processing in an ASN greatly improves performance.

In the other part of our work, we focus on providing operating system support for dynamic reconfiguration of FPGA to provide support for task offloading to FPGA. Operating system support for HW/SW co-design is in its infancy and faces several challenges before it could provide achievable benefits Some of the issues surrounding hybrid computing are resource management across heterogeneous multi-cores, data communication, recovery from errors etc. We have built a prototype reconfigurable system that can offload tasks from a processor to the reconfigurable core. We also developed several scheduling algorithms for resource allocation among HW and SW computing kernels and analyze the performance trade-offs of these algorithms.

Reconfigurable Processing Architectures for Stream Processing and Hybrid Computing

Janardhan Singaraju

A Dissertation

Submitted in Partial Fullfilment of the

Requirements for the Degree of

Doctor of Philosophy

at the

University of Connecticut

APPROVAL PAGE

Doctor of Philosophy Dissertation

Reconfigurable processing architectures for stream processing and hybrid computing

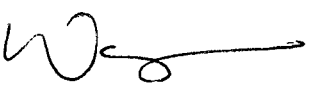
Presented by

Janardhan Singaraju, B.Tech. M.S.

Major Advisor

Dr John A. Chandy

Associate Advisor

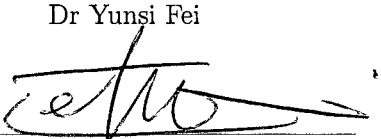


Dr Lei Wang

Associate Advisor

Dr Yunsi Fei

Associate Advisor



Dr Mohammad Tehranipoor

Associate Advisor

Dr Faquir Jain

University of Connecticut

September, 2011

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1

Introduction

1.1 Overview

1.2 Thesis Contributions

In this work, we implemented an Active Storage Network(ASN) on switches composed of NetFPGAs. The focus of this work is on the computational aspects of an ASN and how applications can take advantage of intelligent network switches to perform computations. A few key data processing applications namely data sort, data search, K-min/max, and K-means clustering have been implemented on the ASN. The performance improvements made by offloading some processing from storage node to the network are demonstrated by comparing the application computation time using an ASN with the computation time when it is performed with a normal switch. We also show several functional and data parallel techniques for data processing on the switching elements in ASN.

These techniques can be applied for large-scale data distributed across network. In particular, we show how these techniques applied to a K-means clustering algorithm scale with larger networks and improve the performance.

In order to support hardware implementations in the ASN, we need better scheduling mechanisms.We also propose to build a system that can utilize the hybrid architecture of an FPGA with the embedded processor. The embedded processors on a FPGA running an operating system can utilize the partial reconfiguration feature of the FPGA and dynamically offload function calls to the reconfigurable fabric on the FPGA. HW-SW co-synthesis requires advanced support of tools and resource allocation algorithms. We have developed several scheduling algorithms for dynamic task allocation on hybrid system and evaluate their performance. These algorithms are based on profile information of the application and can adapt to various workloads with different runtime requirements.

1.3 Outline

This thesis is organized into two parts.active storage networks and hybrid computing. Chapter 2 describes the topology and architecture of an ASN switch built on top of a NetFPGA development board. The application modules data search, data sort, Kmin/ Kmax and k-means clustering built on top of ASN switch are explained in detail in Chapter 3. The performance improvements made by processing applications over the network are also evaluated in this chapter .

Chapter 4 explains in detail the concept of hybrid computing and scheduling algorithms for task sharing between software and reconfigurable hardware. We conclude in chapter 5.2 with a discussion of future directions.

Chapter 2

Active Storage Networks

2.1 Introduction

Large scale data processing is heavily I/O dependent. Data must be retrieved from slow mechanical hard drives and then distributed across faster but still relatively slow (as compared to processors) networks. Congestion on a network can cause degradation, and as data sizes increase, the memory footprint of these data sets make data manipulation on a processor difficult. As a result, computation speed is often not the determinant in how fast a data processing application performs.

On the data processing side considerable amount of work has been performed in developing efficient external memory algorithms for large scale datasets that do not fit in main memory [1–3]. Pipelining techniques can be used to interleave computation and communication and improve performance [4]. These algorithms aim at optimizing the total number of I/O operations rather than the

computations. Standard template library in C++ uses efficient parallel IO techniques for common data processing tasks [5]. However, the theoretical limit of at-least one scan of the entire data set residing across the network is inevitable.

On the data storage side, there have been several approaches to the design of parallel I/O systems. Several commercial and academic parallel file systems have been developed using new Object Storage Device (OSD) techniques [6]. These systems allow dedicated data and control paths thus demonstrating aggregate throughput scalability for very large systems. However, these systems do not scale well when I/O requests are too small to stripe across multiple nodes or when applications do many meta-data operations such as file creations and deletions, fstats, and directory reads. Parallel I/O systems also do not perform well when a single client wants to read data from many storage nodes and perform a reduction operation like min-max or a search. In such cases, with high performance storage nodes, we can easily saturate the network connection to the client. Consider that if a storage node can deliver 1 Gb/s of data, 10 nodes can potentially deliver 10 Gb/s of data, thereby overwhelming a single 1 Gb/s network connection to a client. Thus, the client does not see the benefit of the parallel I/O.

Our approach to the problem is to attack the network contention where it exists. This forms the basis of our work for stream processing in an active storage network (ASN). An ASN is comprised of a smart network switch along with intelligence embedded in the I/O network. By embedding processing capabilities in the switching elements of the network, the storage node performance as well

as the computational performance of the parallel I/O systems can be enhanced. Processing capabilities in the switching element can also contribute to an optimized flow of data in the parallel I/O system and further enhance the overall performance of the system.

We demonstrate this idea of an ASN by performing data processing in an intelligent switching system which is built using FPGAs. We have designed a few data processing applications, namely data sort, data search, K-min/K-max,and K-means clustering and show the performance improvements made by offloading processing from storage node to the network. The performance improvements are shown by comparing the application computation time using an ASN and a normal switch.

Data processing kernels can be classified into two categories namely data reduction and data transformation kernels. Data reduction kernels take multiple data elements as input and condense the input to a set of smaller elements. eg. data summation, multiplication,aggregate queries etc. Data transformational kernels take multiple data elements and transforms them to an equivalent number of output elements. Examples of data transformation include sort, convolutions, image processing,etc.

While performing a reduction operation, an ASN switch can perform the operation at the switch level thereby eliminating some of the traffic in the network that could consume the client bandwidth. From the above mentioned example with 10 storage nodes, if the client wants to perform an operation to find the

minimum of a large set of keys stored across these storage nodes, an ASN switch with some intelligence could eliminate 90% of the non-minimum keys from each node which would be discarded at the client anyways, Thereby, the switch can send data to the client at full 1 Gb/s rate while it receives data from storage node at 10 Gb/s rate and the entire network can operate at maximum bandwidth.

ASNs have a similar concept to that of active disks [7]. In active disks, some of the computation is offloaded from the processor at the client to the disk. Previous work has demonstrated the effectiveness of this approach particularly with functions such as storage management, data mining, and multimedia [8]. However, the drawback of active disks in a distributed storage setting is that the data is striped across several storage nodes and each processor at the storage node can only see data residing at that node. Thus, any intelligence at the storage node can not operate on the entire set of data spread across storage nodes. For example, when doing a query in a database for the k items closest to a particular key, each of the m storage nodes will return the k closest items in its portion of the data. The requesting client must then sort through mk items to determine the k closest items overall. The overall computation is $O(n) + O(mk)$ where n is the number of data items per storage node.

In an ASN, the goal is to move intelligence to the network which has a better view of data than the individual storage node, thereby reducing the data transfer latencies and improving the network performance Processing ability on the network also eases some of the computational workload at the network client.

Most of the applications that operate on large sets of data require transforming the data from one form to another. Offloading data intensive parts of these applications to the network could ease client computing resources. It could also reduce network traffic as some of the data transfer operations that read and write data from client to the storage can be avoided. This further provides the impetus to embed intelligence in a networks.

Some other applications that could benefit from the idea of ASN include redundancy optimizations and file-system caching. Parallel I/O systems provide data protection through replication and parity across nodes in the cluster. With the use of an active storage network, the switch can offload parity computations from the client. Distributed file system performance often depends on aggressive caching to reduce network traffic. However, managing a cache across multiple clients can be problematic. ASNs offer an opportunity to present a global cache shared amongst all clients. By placing a centralized cache at the switch, we can free up memory at the client for other purposes. In this paper, we focus on the computational aspects of an ASN.n the k closest items in its portion of the data. The requesting client must then sort through mk items to determine the k closest items overall. The overall computation is $O(n) + O(mk)$ where n is the number of data items per storage node.

2.2 Related Work

Specialized hardware for application processing is common in data intensive and computation intensive applications. There has been significant work on search [9–12], sort [13–16], and K-means clustering [17–20] using reconfigurable hardware.

ASNs have a similar concept to that of active disks. In active disks, some of the computation is offloaded from the processor at the client to the disk. Previous work has demonstrated the effectiveness of this approach particularly with functions such as storage management, data mining, and multimedia [8]. However, the drawback of active disks in a distributed storage setting is that the data is striped across several storage nodes and each processor at the storage node can only see data residing at that node. Thus, any intelligence at the storage node can not operate on the entire set of data spread across storage nodes.

Performing computation in the network has been proposed in early supercomputer designs. For example, the NYU Ultra computer had a fetch-and-add instruction for use in multiprocessor synchronization primitives [21]. These networks, however, have fixed functional primitives and do not allow programmability as with our ASN architecture. Moreover, the primitives are very simple and targeted towards compute-node communications. The ASN, on the other hand, can implement complex functions and are intended for the primary data communication path - i.e. storage to compute node.

2.3 ASN topology and architecture

A critical component of an ASN is the network switch since the switch implements the data processing on data as it is is aggregated and distributed from multiple sources. Typically, custom silicon is used to build gigabit and multi-gigabit switches and these switches offer the best performance. FPGAs on the other hand provide an intermediate design point by offering maximum flexibility in the network processing while achieving high performance. For this reason we use a switch built using an FPGA. A basic network switch implemented in an FPGA is readily available on the NetFPGA board from Stanford [22]. We have modified the architecture to embed data sort, data search and K-means clustering capabilities in the network switch.

The selection of switching topology is also extremely important in an ASN. The topology has an impact on the flexibility of computations performable on ASN. Aspects like the type and nature of the interconnects used, latency, non-blocking, etc. are decided by the switching topology and thus have an important impact on the overall performance and scalability of the network. The NetFPGA, which is the building block of the ASN, has four Ethernet ports and two SATA ports. If we use a conventional Benes or Clos topology we can not make use of the SATA connection which effectively decreases the throughput of each switching node. In order to make use of the maximum 8Gb/s throughput of the NetFPGA board, we use interface modules developed for the SATA ports

[23] and a wide sense nonblocking switching topology named 2-dilated flattened butterfly [24–26].

2.4 ASN Switch

The ASN switch is built on top of a NetFPGA board provided by Stanford and Digilent to help building working prototypes of high-speed, hardware-accelerated networking systems. [22].

2.4.1 NetFPGA

2.4.1.1 NetFPGA architecture

The NetFPGA board is a PCI card, which contains a Virtex-II Pro (XC2VP50) FPGA, specifically designed for network applications. It has four 1 Gigabit/second Ethernet (GigE) interfaces and two SATA ports which makes it suitable to build a switching network. It also has four banks of locally-attached static and dynamic random access memory (SRAM and DRAM). The NetFPGA research group also provides the source code for the board so that it can be used as a hardware accelerated OpenFlow switch [27]. Figure 1 [22] shows the full resources available on the NetFPGA.

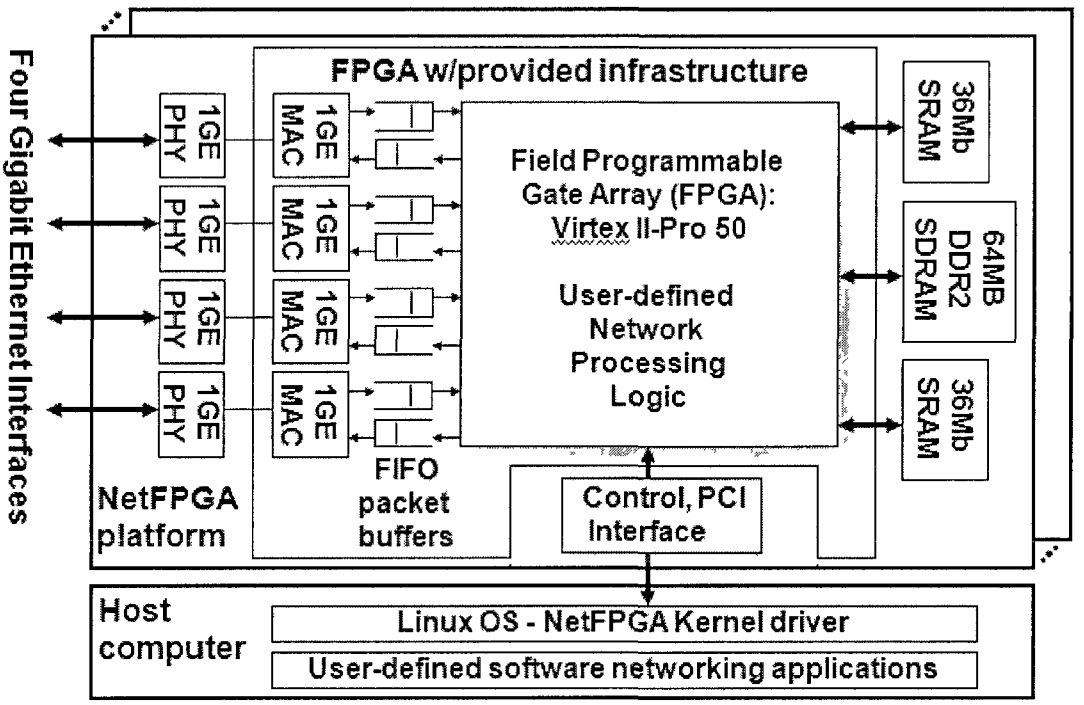


Fig. 1: NetFPGA Board Architecture

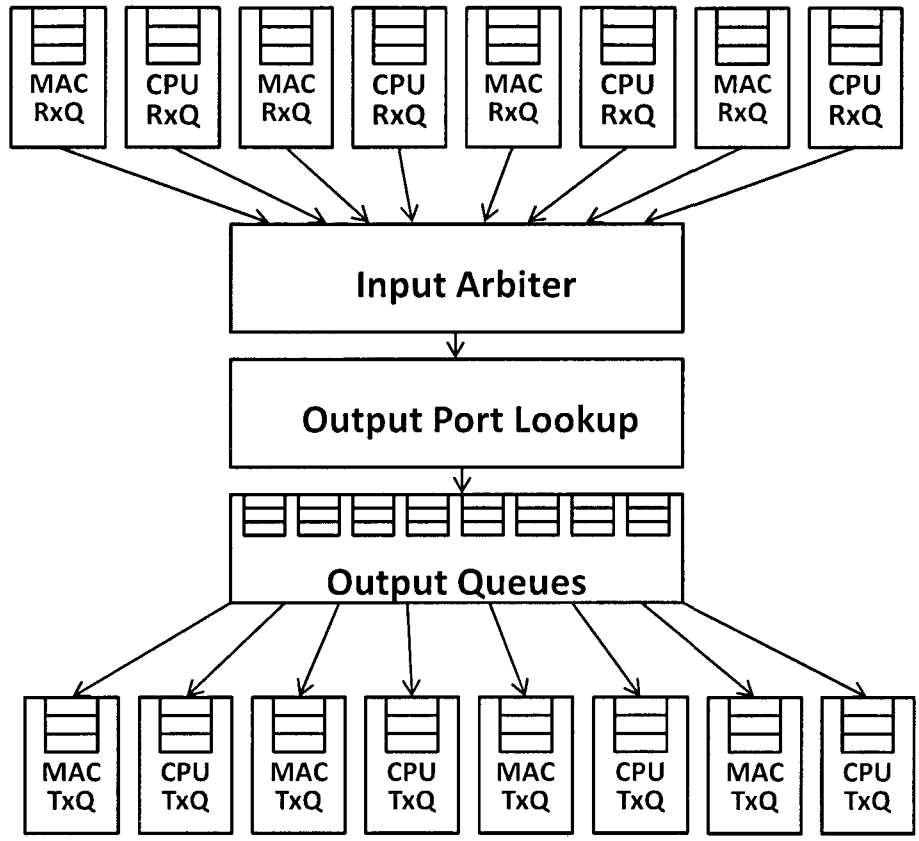


Fig. 2: NetFPGA Switch Pipeline

2.4.1.2 Reference switch pipeline

Figure 2 shows the design of the reference Ethernet switch provided in the NetFPGA package [22]. It is a five stage pipeline structure where each module communicates using a simple packet based synchronous FIFO push interface which makes it easy to add additional modules to the structure for the purpose of packet processing. The user data path is 64 bits wide and it is driven by a 125 MHz clock. Therefore, the switch can handle a maximum throughput of 8 Gb/s

- i.e. the hardware can process packets at line rate. Any packet through the switch should pass through all these stages as shown. The first stage is composed of a series of input queues that receive packets from the I/O ports. The MAC Rx queues and the CPU queues compose this stage. The CPU queues are the queues for the packets received from the PCI over DMA and these queues are not used in our design. The second stage of the reference switch process the queues in the first stage one by one in a round robin way. This input arbiter stage pulls a whole packet from one of the input queues and passes the packet on to the next stage for processing. The third stage output port look up, reads the packet header received from the previous stage and decides on the appropriate output queue to which this packet needs to be forwarded to. Finally the output queues stage puts the packets in appropriate destination output queues after which the packets are transmitted out through the Ethernet ports.

In the proposed switching topology, we are making use of the SATA port of the NetFPGA card for the interconnection between switching elements. The NetFPGA card supports two SATA ports. We are making use of the Aurora protocol for serial communication through SATA [28]. Aurora is a LogiCORE IP designed by Xilinx to enable easy implementation of the FPGA RocketIO transceivers while providing a light-weight user interface on top of which designers can build a serial link. It is a scalable, lightweight, link-layer protocol for high-speed serial communication. It also supports full duplex operation and flow control.

SATA with Aurora can give the maximum throughput if we drive the RocketIO transceivers using a differential clock network [29]. Using a single ended clock generator of 125MHz and Aurora, we can transmit two bytes in a single clock pulse at a maximum throughput of 2 Gb/s which is still twice that of gigabit Ethernet. Thus, each NetFPGA switching element will be able to provide maximum possible throughput of 8 Gb/s by making use of four 1 Gb/s Ethernet link and two 2 Gb/s SATA ports for the interconnection between switching blocks. The proposed ASN switch topology is designed to exploit the complete bandwidth of the SATA link and thus the bisection bandwidth of the switching network is maximized.

2.4.2 User data processing stage

For the purposes of an ASN, we have added an extra stage between the output-port-lookup and output queues stages in the reference switch. This is the stage where all the data processing applications are handled by a special hardware modules. These data processing modules can include reduction operations like min/max, average, sum, search, clustering, compression, etc. and transformational operations like FFTs and sort. The user data processing stage can have one or more of these modules as needed by the overall application. The user data processing stage receives data from the output-port-lookup stage as a stream of Ethernet packets. Figure 3 depicts the data processing units in this stage. Internally, the packets are examined to determine if the packets should be processed

by a user module. If so, these packets are sent to the appropriate application module.

2.4.2.1 Header parse logic

The nodes and the clients communicate using the user datagram protocol (UDP) protocol. UDP is a simple connectionless protocol built on IP that does not require prior handshaking before establishing a connection between the client and server. TCP, on the other hand, uses handshaking to establish a reliable connection. Which comes at the cost of additional header data and slower speed. Because of the extra processing required to handle TCP, we have chosen UDP as the transport protocol. Since UDP does not guarantee packet delivery, client applications using the ASN are must take separate measures to provide the reliability.

In a network setting, there will be several types of packets flowing through the network switch that may not be related to the user application. Some of these packets include packets from applications like HTTP, POP, SMTP, telnet and SSH etc. Not all packets in the switch need to be processed by the user data processing stage. These special packets can be identified by the UDP port number and a 16 bit magic header at the beginning of the UDP packet data. This header can be easily added by the client application. In addition to the magic header, an extra 32 bits of packet sequence number is added by the client application to assist them in providing a reliable service. This 6 bytes of extra

header information provide an intermediate solution between the highly reliable, slower TCP protocol and the less reliable but faster UDP protocol. The header parse logic verifies the packet header for the application port number and the magic header to ascertain that the packet is intended to be processed by the switch. The header parser logic also verifies if the incoming packet is in sequence with the previous stream of packets processed by it. In case of an out of sequence packet, it issues a request to the packet generator logic which in turn sends a request packet to the source requesting for the missing packet.

2.4.2.2 Packet generator

The packet generator uses the header information parsed by the header parse logic to construct these request packets. The packet generator is also responsible for constructing packets with the output data from the sort module or processed data from any other module implemented in the user data processing stage. The packet generator logic calculates the UDP checksum for the new data and updates the appropriate location in the header field of the packet. Once the packet is constructed, it is forwarded to the destination queues from where they are ultimately transmitted to the client/server.

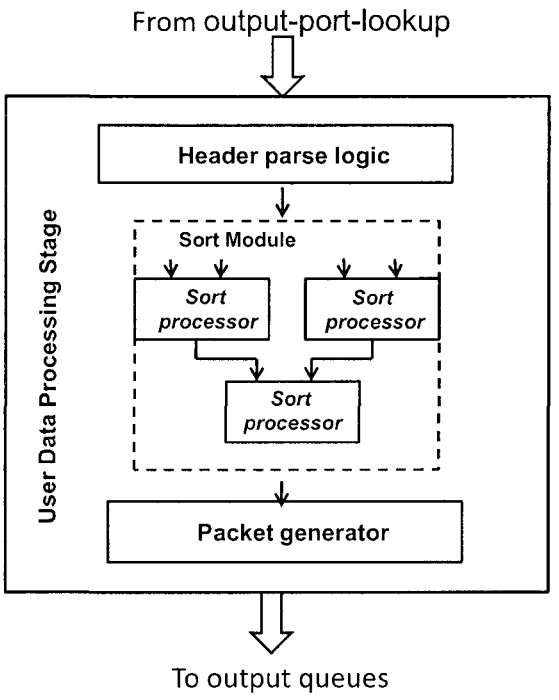


Fig. 3: User Data Processing stage containing the Sort module

2.4.3 2-dilated flattened butterfly

A critical choice in the design of an ASN switch is the topology of the switch. Some of the important design parameters are the number of interconnects, number of switching elements, overall latency, aggregate bandwidth, and whether it is non-blocking. The number of interconnects/links per switching element and the number of switching elements decides the total cost of the switch/topology network. Since we are using the NetFPGA as the switching element, cost will be primarily decided by the number of NetFPGA boards used. The nonblocking behavior is particularly important to insure that the switch can always deliver the maximum throughput, i.e. with a NxN switch with a per-port bandwidth

equal to B, the aggregate throughput should be NB regardless of the connections between ports. A 2-dilated flattened butterfly is a *wide-sense nonblocking (WSNB)* network that can connect from any idle input to any idle output without disturbing existing connections. It has a similar structure to a flattened butterfly with twice the bandwidth or with duplicated links. The structure of a 8x8 2-ary three dimensional 2-dilated flattened butterfly structure is shown in Figure 4. In this structure, interconnections in the first and second dimensions(vertical links) are implemented using 2Gb/s SATA links and all interconnections in the third dimension (horizontal links) are implemented using 1Gb/s Ethernet links. Each switching element shown in Figure 4 can be implemented using a NetFPGA board and it can deliver a maximum throughput of 8Gb/s since each switching element consists of four 1Gb/s and two 2Gb/s links. This 8x8 structure can be used as the basic building block to construct larger networks. The procedure to constructing larger flattened butterfly networks from smaller network is explained in [30].

In order to utilize the maximum throughput of 8GB/s provided by the NetFPGA and the low component cost of interconnects, we have used a 2-dilated flattened butterfly network as a backbone of ASN.

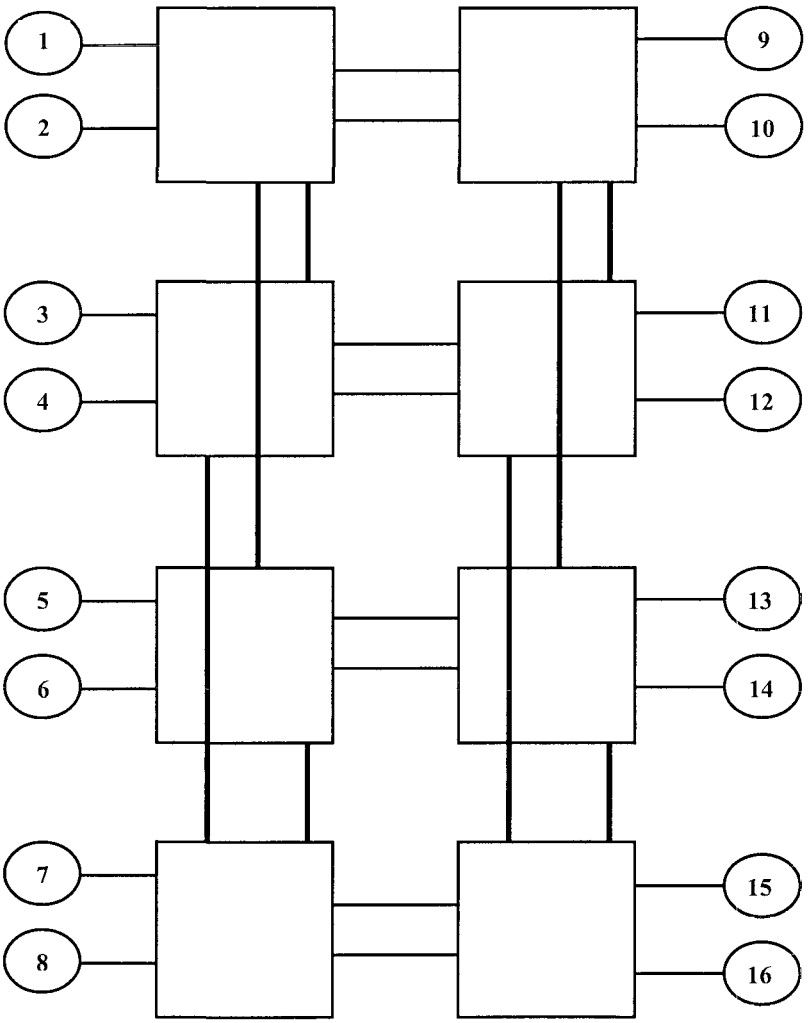


Fig. 4: 2-dilated flattened butterfly.

Chapter 3

Application Modules

ASNs can accelerate application performance in two specific classes of operations: reductions and transformations. Reductions take a large set of data and reduce it to a smaller set of data. Scientific applications often use similar reduction operations such as minimum, maximum, summation, etc. Transformational operations are another class of functions that take a large set of data and transform it into another large set of data. This transformation may be written back to storage or returned to the client. Video editing is composed of many transformational operations such as convolutions, filters, and compression which could benefit from an ASN.

We have implemented several reduction and transformation operations in the ASN including search, sort, k-min/max and k-means clustering. These applications are implemented in the user data processing stage of the network switch described above. The header parsing logic parses the Ethernet packets, strip the

packet headers and provides data in an appropriate format to the application module. The application modules see the stream of Ethernet packets coming in to the switch as a stream of application data from the header parse logic. The necessary reduction operations are performed and the processed data is forwarded to the packet generator where new packets are constructed and dispatched to the intended destination. In the remainder of this section, we describe four representative application kernels - namely search, sort, K-min/max and k-means clustering.

We have implemented data search, sort, and k-means clustering applications in an ASN switch built on top of a NetFPGA board. An 8×8 ASN network forming a 2-dilated flattened butterfly structure is built using 8 NetFPGA boards. The 8×8 switching network consists of 8 switching elements with each switching element having 2 input and 2 output ports. A single NetFPGA board with four ports can implement the switching element. We used a 8×1 subset of the network connecting 8 server nodes to a single client node, to evaluate the performance benefits of ASN switch for reduction and transformation operations. Figure 5 shows an implementation scheme for an 8×1 reduction/transformation operation and routing path from several servers to the client at node 9. The switching elements 1 to 4 perform the first level of application processing. As each of the switching elements are connected to two server nodes, they do not have the global view of the entire data. Hence a next level of processing is required. The switching elements 5 and 7 perform the final stage of processing

before transmitting data to the client at node 9. Switching elements 6 and 8 operate as pure switching elements and do not perform search or sort operations. In fact for search application, the second level processing in switching elements 5 and 7 is not necessary as the first level would be sufficient to search the client data from all the nodes. k-means clustering application however uses all the available switching elements for application processing.

In normal operation, the switching network acts as the interconnect between the servers and the client. The client would then have to locally perform any of the applications like search, sort, or k-means clustering on the accumulated data. However in an ASN topology, these applications are implemented in the switching network itself, thereby the client receives the processed results from the network.

When implementing application processing in the network, the ASN network differentiates the application packets from normal Ethernet packets by a 16 bit magic number (0x2020) at the beginning of UDP packet data. The other packets in the network are simply routed according the routing scheme implemented. The ASN packet size is limited to 1512 bytes. NetFPGA currently does not support jumbo frame sizes but the architecture provided for the search, sort and k-means applications are scalable for bigger packets. With 14 byte Ethernet, 20 byte IP, 8 byte UDP and 2 byte ASN headers, the application data is limited to 1464 bytes per packet. The application data consists of multiple $key - value$ pairs. In the data search and sort applications, the *value* field is propagated after the necessary operation on the *key*. A smaller *value* would mean more *keys* per

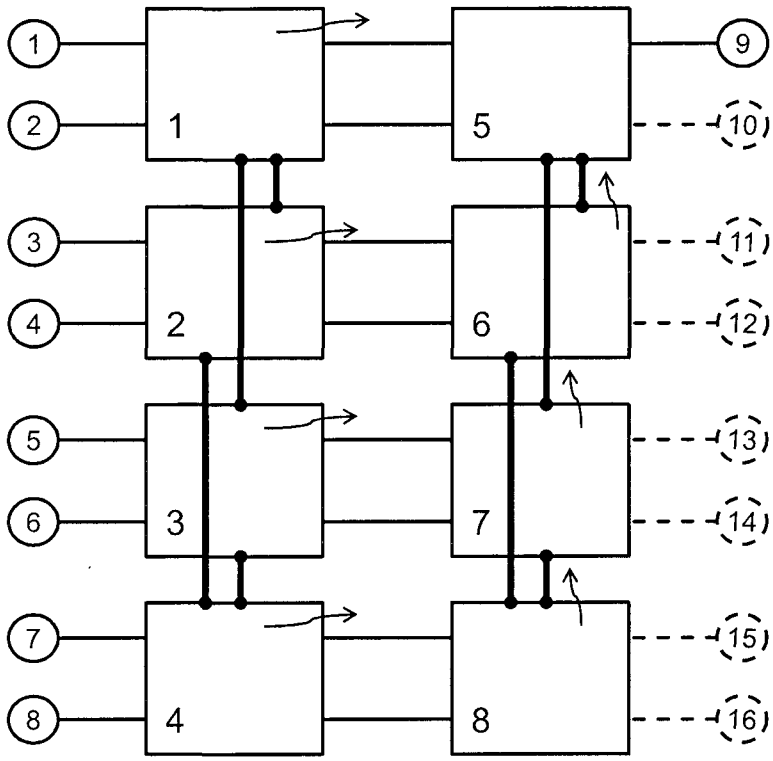


Fig. 5: An 8×1 routing scheme on a 8×8 2-dilated flattened butterfly network

packet and hence more comparisons. For this reason, the *value* field is kept to the minimum to reflect the worse case scenario.

The *keys* are generated at random with each key having 32 bytes of data. For our experiments, we used file sizes up to 4G bytes per server node. The client would then receive up to 24 GB from all the 8 servers. Each of the applications and the experimental results are described in the following sections.

3.1 Data Search

Search is a common application for data processing. Database search is performed on a $key - value$ pair, where a search for a specific string in the *key* field would return the matching $key - value$ pair. The core of this application is to look for specific string called *searchstring* henceforth, in all the *key*s in the system. search is a time consuming operation in a distributed system where all the $key - value$ pairs are spread across multiple storage nodes.

Search strategies can be broadly classified as linear, binary, hash and tree based search. Some of the string search algorithms include Rabin-Karp algorithm [31], KnuthMorrisPratt algorithm [32] and the Boyer-Moore algorithms [33]. Tree and hash based search algorithms are faster than linear search algorithms but they require the search space to be arranged in a tree or hash based data structures. Linear search strategy suits the network search scenario as data streams through the network from several storage nodes. We implemented a CAM based hardware search module in the network switch that can search for a specific *searchstring* in the incoming packet.

The search module constitutes of the character match array and the PE array logic blocks.

3.1.1 Character match array

The character match array contains a $k \times 1$ array of CAM blocks into which the k byte *searchstring* is loaded. Each of the CAM blocks contains an 8-bit memory to store a single ASCII character of the *searchstring*. A wider memory can support multi character comparisons such as the Unicode code character set. The CAM block also stores two additional bits to indicate if the character stored forms the beginning or the end of the *searchstring*. These extra bits are loaded in to the CAM array along with the *searchstring*. The CAM block compares the input character with the search character stored in the memory. The *key*s from the network are given to the CAM block a character at a time. This input character is applied simultaneously to all n columns in the array.The comparison results from all the CAM blocks are propagated to the PE array as character match signals M(i). The CAM evaluates a character match in a single cycle. However, the multi character string match is evaluated by the PE array in the subsequent stage.

3.1.2 PE array

The PE array is a $k \times 1$ array of processing elements (PE) that executes the algorithm shown in Figure 1. To perform the search operation, the *key* data from the header parse logic is given as input to the character match array, a character at a time. If the input character matches a column in the CAM array,

the match signal is set and passed onto the corresponding PE for that column. Each PE holds a binary value called a flag which indicates that the input *key* data matches the *searchstring* in the CAM up to this point. The PE flags are labeled PE(i). The flag is set for all the character matches at the beginning of the *searchstring*. The PE logic propagates the flag to the subsequent PEs if there is a corresponding match signal M(i) from the from the character match array. The flag is however dropped for a character mismatch. A search hit occurs if the flag propagates all the way to the end of the *searchstring*. Figure 6 shows an instance of search for the string "NCC" in action. The *searchstring* is stored in to the character match array and the *key* data from packets are applied as input to the character match array. In the first clock cycle (Not shown in the figure), a match on a character 'N' in the input set a flag in the PE array. In the subsequent cycle, the character match on 'C' moved the flag to the new position. In spite of the character matches on 'C' at multiple locations, the PE flag is set at a location that has a PE(i-1) set. In the final cycle the character match on the last character of the *searchstring* 'C' resulted in a search hit. The search method described above could search for any number of *searchstring*s loaded in to the character match array. After a search hit on a *key*, the corresponding $key-value$ pair from the incoming packet is propagated to the packet generator logic. The other $key-value$ pairs from the packet that did not match the *searchstring* can be dropped. The search process described here is a linear operation and can be done in $O(n)$ time.

This architecture supports search on multiple *searchstrings* stored in the CAM array. The array of the PE elements process all searches in parallel. A search on $\frac{k}{s}$ search strings each having a length s bytes can be evaluated in s cycles. A dynamic search on new *searchstrings* can be easily performed by replacing the contents of the CAM array with the required *searchstrings*.

```
1: if M(i) and wordbegin then
2:    PE(i) = 1
3: else if M(i) and PE(i-1) then
4:    PE(i) = 1
5:    PE(i-1) = 0
6:    if E(i) = 1 then
7:       searchhit = 1
8:    end if
9: else
10:   PE(i) = 0
11:   searchhit = 0
12: end if
```

Algorithm 1: PE Algorithm

3.1.3 Results: Search

In a network search implementation, The available resources on the Xilinx Virtex-II Pro FPGA limited the space available for the *searchstrings* to 4096 Bytes. We chose random sized *searchstringss* ranging up to a maximum of 32 bytes per *searchstring* for both the software and network search implementations. The software application for search is based on Aho-Corasick [34]

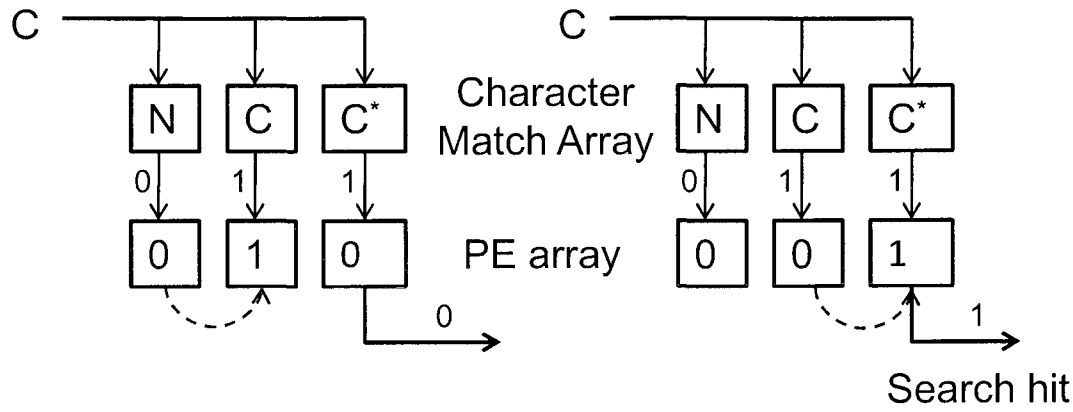


Fig. 6: Data Search

algorithm for dictionary searches. In a network search, the *searchstrings* are initially loaded only in to the switching elements 1- 4 of the ASN network 5 and the switching elements transmit the results of search operation to the client. The client receives only a portion of the server data that matches the *searchstrings*. for the measurements purposes, the server data is embedded with one extra reference packet of 50 bytes at the beginning of the data. these reference packets are not processed in the switching elements of ASN but are transferred to the client from the switch queue. The network search is timed at the client side once it receives these reference packets from all the servers. The results plotted in figure 7 show network search performing an average of 7 times faster than software search.

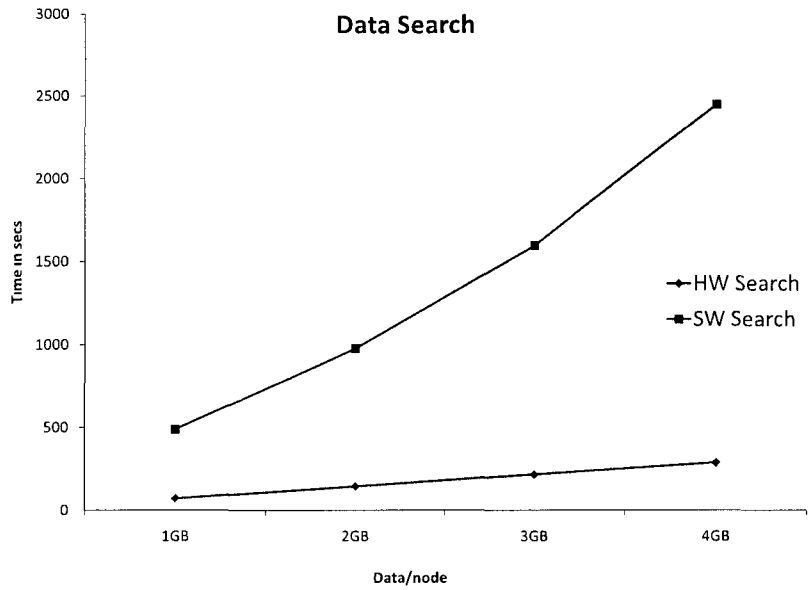


Fig. 7: Time to search 8GB-24GB of data

3.2 K-min/K-max

Min/Max is a data reduction operation to find the smallest or largest item in a given dataset; the k-min/ k-max problem is to get the first k corresponding min/max items. It is a restricted version of sort and has its applications in database and data mining systems. In a distributed setting, the data set is distributed across several storage nodes on a network. The traditional implementation of k-min/k-max requires the dataset to be retrieved from all the storage nodes to a single client before the k-min/k-max operation can be performed. An ASN implementation would benefit this application by overcoming the bottleneck associated with performing this reduction operation. It can also improve

the network performance by eliminating unnecessary data traffic in the network which otherwise would be needed.

The k-min/k-max user data processing module contains k buckets connected as shown in Figure 8. Each of the buckets consists of a register to store a data item and a logic function that executes an algorithm as described below. The *key* data from the header parse logic is given as input to the k-min/k-max module. The *key* data is of alpha-numeric in nature and can be of varying length having a maximum size up to the size of register in the bucket.

The registers in all the buckets are initialized to a lowest or a highest value possible for k-min and k-max operations respectively. These values are 0x00000000 00000000 and 0xFFFFFFFF FFFFFFF for a 64 bit register and are relative to the size of register. The *key* data from several storage nodes are given sequentially to the first bucket. The first bucket compares the *key* data to the register content at this bucket. For a k-min operation, if the data item is smaller than the register content, the register is updated with the *key* value and a value of 0xFFFFFFFF FFFFFFFF is forwarded to the subsequent bucket. In case the *key* is greater than the content of the register, it is simply forwarded to the next bucket. Similarly for a k-max operation, the register in a bucket is update with a new value if the *key* is greater than the register content. A value of 0x00000000 is propagated to the next bucket in this case. The *key* itself is forwarded to the next bucket if it is smaller than the register content. In essence, the buckets operate as a shift register maintaining a sorted list of the top k keys. At the end

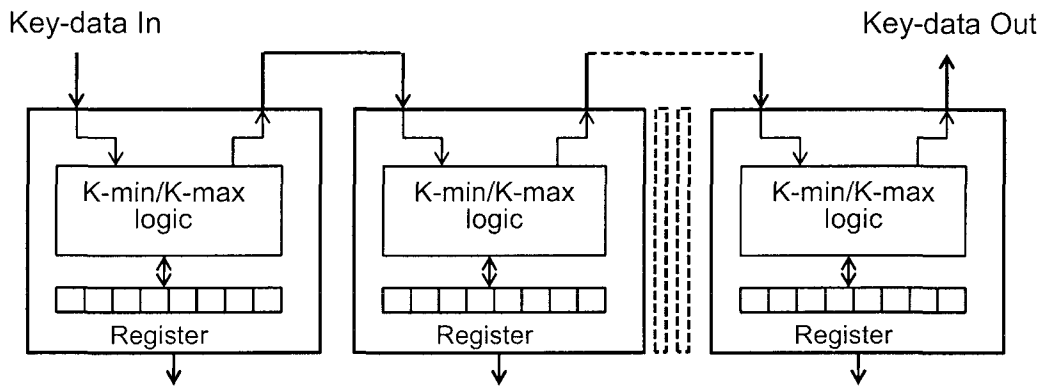


Fig. 8: k-min/k-max module

of the network packet data-stream, the k buckets contain the k-min items in a non-decreasing order and k-max items in a non- increasing order. These k items are then forwarded to the the packet generator where they will be assembled as new packets and dispatched to the final destination.

Each of the buckets process *key* data in parallel so the application runs at full line speed. The runtime for finding k-min or k-max *keys* from n *keys* is $O(n+k)$. Since typically $k \gg n$, the runtime is $O(n)$.

3.2.1 Results: K-Min/K-Max

In our experiments, we have set $k = 50$ to find 50 minimum and maximum values in the data for both software and network k-min/k-max implementations. For the network k-min/k-max implementation, the experiment is repeated for different client data sizes beginning from 800M to 4000M. A reference packet at the beginning of the data is used for experiment timing purposes. The k-min/k-max elements in the switch elements are flushed to the client after the last packet

from the server data. The network sort is timed at the client between the interval it receives the reference packet and the last packet from the k-min/k-max data. The results in Figure 15 show that the ASN k-min/k-max application runs 20 times faster than the corresponding software implementation.

3.3 Sort

Sorting is a common transformation application where data from multiple channels of I/O systems converge at a single node. By moving this process to the switch, the client node which is originally intended to perform this operation locally receives data in sorted order and can use its processing resources on other time consuming computations. Sorting is a common task performed in data handling applications like indexing, data mining and database management. The most common database query is some form of "*SELECT* ... from ... ORDER BY ..." that retrieves sorted data from the database such that it matches a set of criteria. With a normal SELECT and active disks alone, each disk will do the query independently and then return the data to the client. However, when doing an ordered query on n nodes, the m returned results from each of the node must be sorted before being returned to the client.

With the amount of data storage growing to the order of terabytes, data sorting often becomes the bottleneck in some applications thereby affecting their performance. In fact, a popular measure of server and I/O performance is the GraySort benchmark [5], which attempts to sort terabyte size key-value pairs.

Several FPGA based architectures for implementing sorting have been proposed [16,35]. However, data records residing on a storage over network have to be fetched in to the local storage at a host computer before they can be sorted. ASN can take benefit of this fact by implementing the sort process over the network. The FPGA based sort implemented in the ASN switch sorts data records as they stream the network and the host computer receives the sorted data.

Our assumption is that the active computation at each storage node will pre-sort its portion of the data and send the pre-sorted data to a requesting server. Normally, the server would then be responsible to merge sort the sorted data from the various storage nodes. With an ASN, the data will be sorted within the network. A tree based merge sort in the ASN can reduce the sort time from $mn \log m$ to $\log^2 mn$ where m is the number of nodes and n is the size of the data.

3.3.1 Sort module

We have implemented the data sort processing kernel as a sort module in the switch user data processing stage. The basic block of the sort module is the sort processor shown in figure 9. A sort processors takes two continuous data streams of $key - value$ pairs and merge sorts them by their key value. In general, p data streams can be sorted using $\log p$ levels of sort processors connected in a binary tree. In the NetFPGA we are limited to four 1 GigE interfaces thus the two stage binary tree can process four continuous data streams in parallel.

The significance of the sort processor is that it can sort data streams containing $key - value$ pairs of any arbitrary length. Since $key - value$ pairs in in a network could be of arbitrary length, a general purpose sort processor is an essential requirement. The sort processor fetches two 64 bit inputs $dataA$ and $dataB$, by issuing $reqA$ and $regB$ requests. If the comparison of the two inputs does not yield a result, i.e. if the keys for $dataA$ and $dataB$ are the same, the next 64 bits of $dataA$ and $dataB$ are fetched and the current $dataA$ and $dataB$ are stored in temporary storage where they wait for the next comparison. The temporary storage limits the maximum sort distance to which two equal $keys$ can be sorted. This value is flexible and is dependent on the resources available on the FPGA. We allocated 64 bytes for the temporary storage. If the comparison yields a result, the smaller of the two inputs is sent as sorted output to an output buffer and the appropriate request signals are set to fetch new set of data. In some cases where the two keys are equal, the comparison does not yield a result even after the end of the $key - value$ pair is reached. The sort processor identifies this case and clears the temporary storage results to the output buffer. The sorted output in this case could have the equal $key - value$ pairs in any order. While the data streams through, the $key - value$ pair might end anywhere on the 64 bit boundary. The sort element recognizes the end of the key character and limits the sorted output till the end of that $key - value$ pair. The sort resumes from the beginning of the next key and this process continues on till there is no more data left to be sorted.

The output-port-lookup stage in the reference pipeline stores the packets in SRAM where they wait on the sort module to operate. The sort module contains three state machines operating in parallel. The read state machine issues read requests to fetch data from SRAM. The fetched data is buffered in data store FIFOs from where the sort element described above gets its data.The read state machine is analogous to instruction pre-fetch cycle in a CPU. The sort processor issues requests for dataA and dataB as and when required. prefetching the dataA and dataB in to the FIFOs greatly improves the performance as the requests from sort processor are handled immediately and the wait time between issuing request and fetching data is reduced. Once the sort operation is completed, the sorted output in the output buffer is written back to the SRAM by the write state machine.

The sort processor compares k sorted *keys* from two packets yielding $2k$ sorted keys at the output. In the worst case there will be k comparisons, with each comparison a constant time operation. The merge will take k cycles to complete. In general, a packet with a b byte payload would take $\frac{b}{k}$ cycles for the merge operation. The run time for the sort is the same as a software sort. However the significance of network sort lies is in reducing the deficiencies of a parallel I/O systems in a distributed network setting.

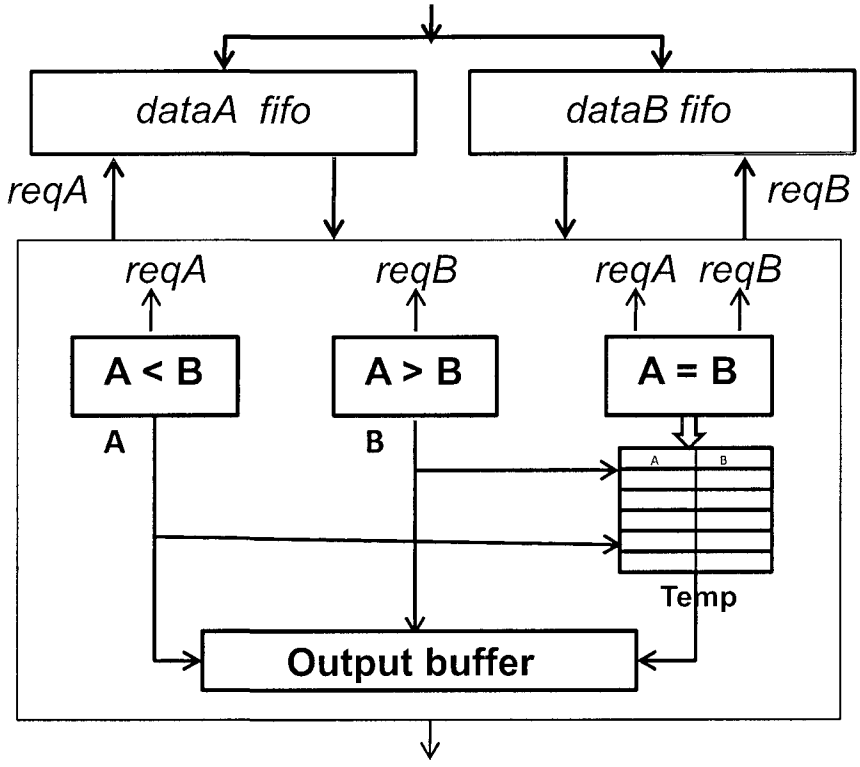


Fig. 9: Sort processor module

3.3.2 Results: Data sort

Network sort is implemented on an NetFPGA based network switch. The base NetFPGA switch design on Xilinx Virtex II pro consumed 15572 slices taking up 65% of the FPGA slices. The inclusion of the network sort unit accounted for a mere 4% addition of the slices thus consuming 16329 slices of the total 23616 slices amounting to 69% of the FPGA slices. The FIFO based network sort switch also consumed 169 RAMB16s block RAM blocks compared to the 157 RAMB16s blocks in the base switch design.

In the NetFPGA we are limited to four 1 GigE interfaces thus can support only four directly connected machines. All the machines run Linux on a dual core AMD Opteron 1.8Ghz processor. The data that is to be merged is distributed across these four storage servers. In a typical network setting, the servers would accumulate the data to a single client machine. A software application running on this client would merge the accumulated data. However in a hardware based network sort setting, the storage servers send their data to a network switch where the merge is handled. A client-server C program handles all the data transfer operations across the network, and a C software application handles the software sort. The software application implements the best known k-way merge sort algorithm and all the C programs are compiled in gcc with the maximum optimization levels.

The server programs packs the Ethernet packets with a 16 bit magic header 0x2020 if they are to be sorted on the network switch. The Ethernet packets without this header are treated as normal packets and are transmitted without any processing. Figure 10 shows the packet header format for the network sort processing. An 0x09 byte is used to separate $key - value$ pairs within a packet.

The test data is randomly generated with a varying number of *key* sizes per packet. For our experiments, we used a file size of 1 Gig bytes per storage server node. In a $key - value$ pair comparison, *keys* are compared and the *value* field is propagated after the *key* field. A smaller *value* size would mean more number of *keys* per packet and hence more comparisons. On the other hand, a higher *value* size would result in less number of *keys* per packet and hence fewer comparisons. To reflect the worst case scenario, the test data is generated to contain *key* field only.

Packets with *key* sizes of 8, 16, 32 and 64 bytes are generated and transmitted by the servers to the host. Two different test cases with packet sizes limited to about 1512 bytes and 112 bytes were used. The 48 byte header for each of the packet would limit the pay-load to approximately 1464 and 64 bytes respectively. The current NetFPGA architecture does not support jumbo frame packets but the sort architecture discussed above is easily portable to a NetFPGA version that supports jumbo frame packets.

The resulting merged and sorted data can then be forwarded to either of two locations: 1) to a single host that stores all the sorted data or 2) distributed

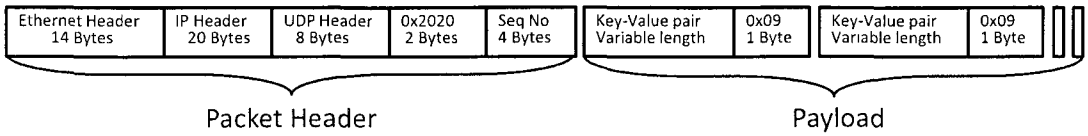


Fig. 10: *Packet Header Format.*

back to the original servers where they store the globally sorted data in a virtual striped manner. Each of these methods has its own merits. In a single host scenario, one of the machines is used as both the host and server. The servers act as storage node that sends its sorted data to the host and the host receives the data from all four servers. We present these two scenarios separately below.

3.3.2.1 Single host

In this section, we evaluate the single host scenario where data is sent to a single host and the resulting sorted data is stored locally at the host. Figures 11 and 12 show the throughputs of a plain network, network sort and the software sort for 1512 byte and 112 byte packets. The throughputs for the plain network and network sort are calculated based on the number of packets received at the host from the network switch. For a network switch with data processing capabilities, the network transfer time and the hardware sort time effect the throughput. The throughput for the regular network switch reflects the network transfer time only. The throughput for the software sort is calculated based on the number of packets sorted by the software sort application at the host as it receives data from the server. The network transfer time and the software sort

time influence this throughput. In a software sort, we do not measure the time to write the sorted data to disk as our host node did not have sufficient memory buffers to handle that much volume of data. Note that the FPGA network sort does not suffer from this problem.

Figure 11 shows for a large packet with about 1464 bytes of payload data and with smaller key size of 8 bytes, the network sort is little over 5 times faster than the software sort. As the key size increases, the number of keys per packet is reduced, and as a result, the average number of comparisons per packet is reduced, thus reducing the overall software sort time. This effect is reflected in the reduction of speedup to a value of 2 times for the *key* size of 64 bytes. Note that the network sort performance for different *key* sizes is almost the same as network performance. This is due to the fact that the worst case performance of hardware sort on the network switch is the same as the network performance and any improvements in hardware sort performance is negated by the slow network performance Thus, in other words, the sort almost comes for free since it is being done in line with the data transfer. These results are further supported by the results from a smaller sized packet as shown in Figure 12. For a 64 bytes of payload data, the improvements in the speedup are similar to that of a large packet. With a *key* size of 8 bytes, the network sort performed about 3 times to a software sort and for a larger *key* size of 64 bytes, the network sort performed almost two times as faster as software sort. The results also indicated that the 1 Gbps I/O resources at the single host side are maxed out and the clients can

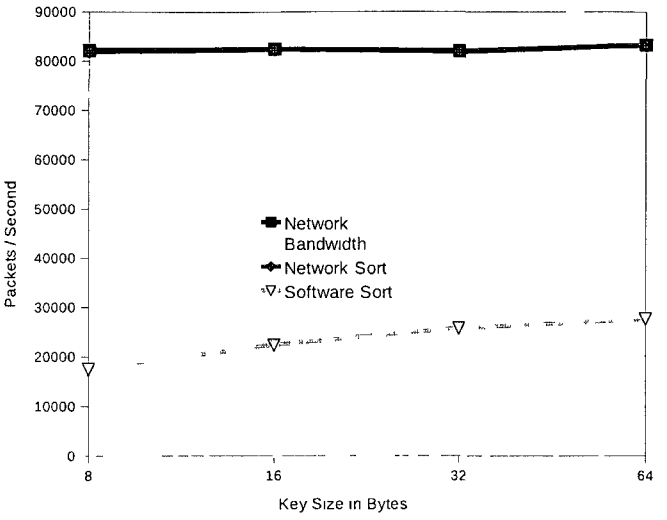


Fig. 11: *Bandwidth comparison for 1512 byte packets.*

not send data any faster in spite of their I/O resources running below their peak abilities at around 0.25 Gbps.

3.3.2.2 Distributed sort

Storing data on a single host suffers from poor write performance as all the sorted data is written to a single storage disk. Often, the sorted data is instead stored back to the storage servers from where the unsorted data was read. This is performed by a sort operation accompanied by a write operation to multiple storage nodes. The network sort topology can improve this sort performance by taking advantage of the parallel I/O to distribute the sorted data back to the server nodes. The network sort in this case sorts data from multiple nodes in the

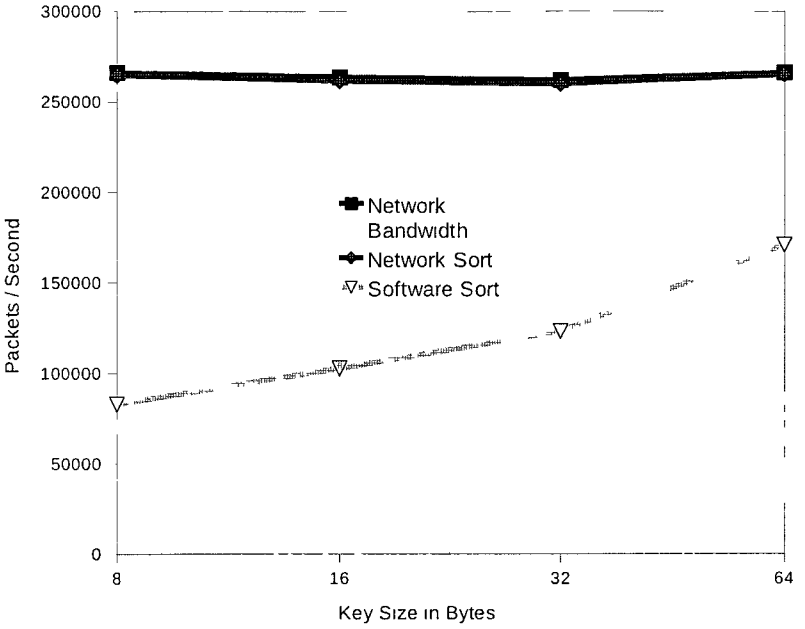


Fig. 12: *Bandwidth comparison for 112 byte packets.*

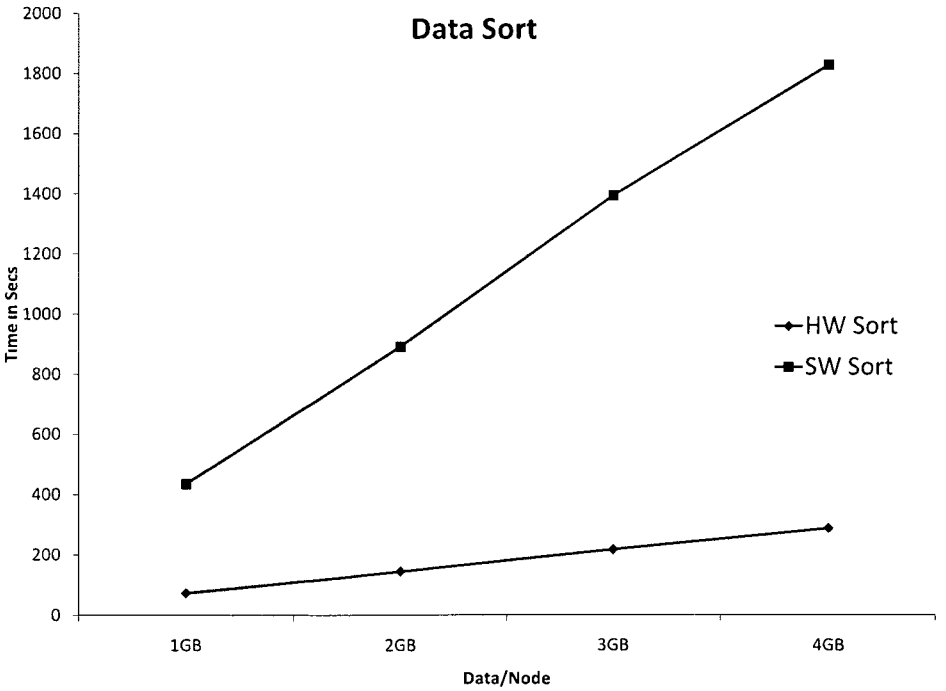


Fig. 13: Time to sort 8GB-24GB of data

network switch and once the data is sorted, it is distributed back to the storage servers instead of aggregating at a single host.

We ran two experiments. First, a software sort and distribute, where server nodes send presorted data over a normal switch to a host which then merge sorts the data and then sends the data back to the server nodes for storage. Secondly a FPGA-based network sort and distribute where server nodes send presorted data to the FPGA switch which merge sorts the data and send the data back to the server nodes for storage. The difference is that the FPGA-based network sort does not require a host to accumulate sorted data, thus removing the bottleneck from the switch to the host.

Figure 14 shows the performance improvements made by employing the network FPGA sort architecture for distributing sorted data back to the servers as opposed to a single host based software sort and distribute. The *key* sizes in the test data are limited to 32 bytes. The graphs shows the time to accumulate sorted data on a single host. The results show the network sort and distribute outperforming software sort and distribute in a single server topology. Network sort and distribute ran 13 times faster.

While both software and hardware sorts grow linearly, the hardware sort can do so at a much faster rate because of its ability to handle Gigabit network transmissions at line rate. The results indicate that when the network is operating at full bandwidth, the software application could not keep up with the rate at which it receives packets and the server utilization was at maximum. The Linux

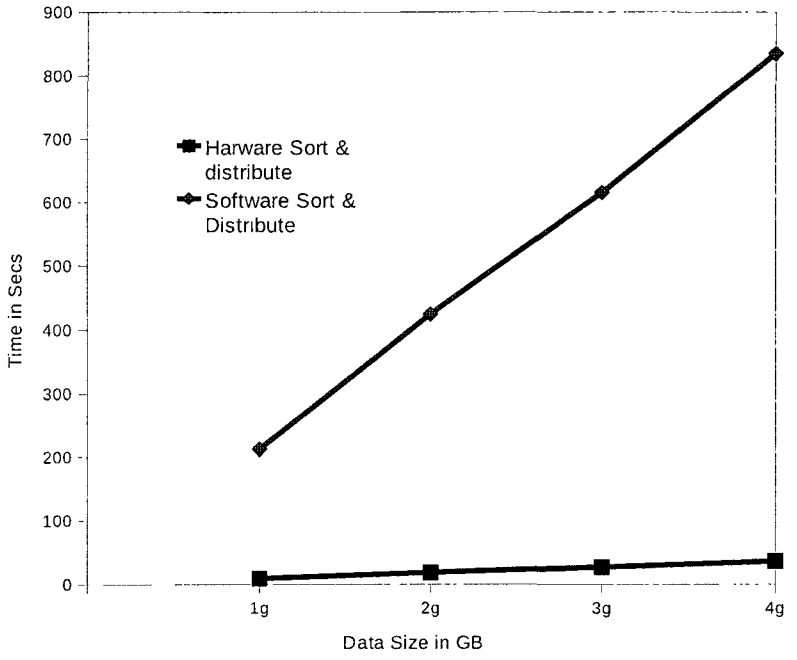


Fig. 14: *Sort and distribute performance.* Data size shown is the data size per node.

command of time was executed to see the amount of time spent by the host in running the user application. With a software implementation of sort, the host spent an average of 9 times the time spent with an FPGA network implementation of sort. This further makes the case for hardware sort over the network.

The sort unit in the current implementation operates on four incoming packets at a time. This limit is due to the number of ports available on the NetFPGA board. However, the design is scalable and it could be easily extended to support any number of ports. Multiple NetFPGA boards can also be connected in a tree-like network to increase the scalability.

Network sort is implemented on a $8X8$ network switch for sorting data spread across 8 storage nodes. The software application implements the best known k-way merge sort algorithm in C. The network sort is timed at the client beginning the first server packet received at the client. The results in 13 show network sort outperforming software sort with a speed up of over 11 times.

The higher performance of the hardware sort is due to the network topology in a distributed storage setting and the reduction operation being performed. The server is connected to the network with a 1 Gbps connection, the client can send data to the server at a rate of 1/4 Gbps each. The sort application running in the switch on a 125 Mhz clock could process a single byte of data per clock cycle in the worst case scenario and therefore a yielding a worst case throughput of 1 Gbps which is the maximum rate at which the server can receive the data. Irrespective of the improvements in the software application for the sort, the software sort under performs the hardware sort due to the network transfer delay associated associated in this setting.

3.4 K-means clustering

Clustering algorithms are prevalent in image processing, statistical data analysis, data compaction, and pattern recognition applications. Clustering algorithms try to group a given dataset in to a set of clusters such that all the elements belonging to a cluster are closely related with the least possible distance across the data items. Such a cluster assignment is an NP hard problem.

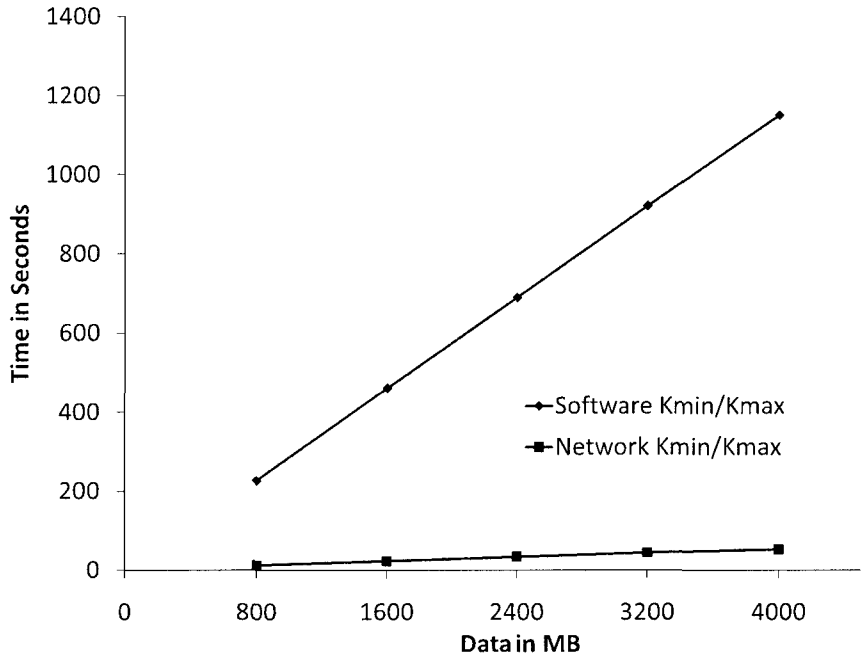


Fig. 15: Time to find 50 min and max keys in 800-4000MB of data

K-means data clustering is a popular clustering algorithm that performs clustering based on Euclidean distance calculation across data points. It is used in a wide variety of applications including data mining. The K-means algorithm is compute intensive due to the iterative nature of the algorithm and the higher degree of complexity involved with floating point computations. The amount of data and computations involved make real time data processing virtually impossible even on high end workstations.

3.5 K-Means Data Clustering Algorithm

Given N multidimensional data points each having D dimensions, the clustering algorithm performs data grouping to k clusters as shown in algorithm 2.

Input: Set of N, D-dimensional data points, Number of clusters K
Output: N, D-dimensional data points assigned to K clusters, K cluster centers

- Initialize the cluster centers C_{i_k} with first k data points or a heuristic

repeat
  foreach *data point* X_i *in* N **do**
    foreach *cluster* C_j *in* K **do**
      - Calculate the Euclidean distances $\Delta_k = \sum_{j=1}^{D} |X_{ij} - C_{i_kj}|^2 \; \forall k \in K$
      - Calculate $\Delta_{k_{min}}$ and assign X_i to $C_{i_{kmin}}$

    end
  end
  Update cluster centroids
until *Iteration count* $\geq$ *a fixed constant* | *Crossover count* $< \delta$;

Algorithm 2: Kmeans clustering algorithm

The algorithm starts by selecting k initial cluster centroids by a heuristic or sampling method. The Euclidean distances are computed between each multidimensional data point and all the cluster centroids. Each data point is then assigned to the cluster with the least Euclidean distance. After all data points have been assigned a cluster, new cluster centroids are calculated by taking an average of the data points in the new clusters. The algorithm proceeds with the

next iteration and terminates if the number of points crossing over to a new cluster at the end of the iteration is below a threshold or after a sufficient number of iterations have passed.

The most time consuming part of the computation is the Euclidean distance calculation between the data point and the cluster center. For all data points, we must calculate the distance between the data point and all the K cluster centroids. Moreover, these calculations are performed with floating point numbers. However, these calculations do not have any data dependencies across clusters and hence can be performed in parallel.

The Lloyds K means clustering algorithm differs from other clustering algorithms in the objective function chosen for minimization which is the square of the mean distance between the data points. Other variations of the algorithms include minimizing the sum of distances to the nearest center [36] and minimizing the geometric distances to the center [37]. In a hardware implementation, the floating point operations take a varied number of cycles for completion depending on the type of implementation. They are also expensive in hardware due to the amount of area occupied on the chip/FPGA. For this reason initial hardware implementations have tried to reduce the cost of Euclidean distance calculations by replacing them with Manhattan distance and max distances [17,38] which would require fewer floating point units. This resulted in a slight decrease in accuracy which was outweighed by the saved space. Other implementations have taken a hardware/software approach where the cluster assignment part of the algorithm

was implemented in software [39,17]. We implemented the original k means algorithm thus preserving the accuracy. Our implementation is targeted for large scale data in a distributed storage system by performing computations in the a network switch as the data flows from the storage nodes to the computation node.

The primary K-means clustering components are the *cluster*, *kmin*, and *cluster_update* units. These units implement Lloyd's algorithm for optimal cluster assignment. A *cluster* computes the distances across the cluster centroid and the given data point. It is implemented with floating point subtracter, multiplier and adder units as shown in Fig. 16. The cluster centroid is stored in a d wide register in the *cluster* unit. It is a volatile data point that might change after every iteration, but remains constant within each iteration. An iteration constituting a single transfer of data set from the storage node to the hardware, concludes by updating the cluster register with the new cluster centroids calculated by the *cluster_update* unit. The first K data points received by the hardware are stored in the k cluster registers. These k data points act as the initial cluster centroids for the algorithm. The rate of convergence of the algorithm depends on the first k data points received by the hardware. The data FIFO s in the user data processing stage receive the remaining set of data points. An array of k cluster units concurrently fetch data points from the data FIFO s and calculate the distances to their corresponding data centers in parallel.

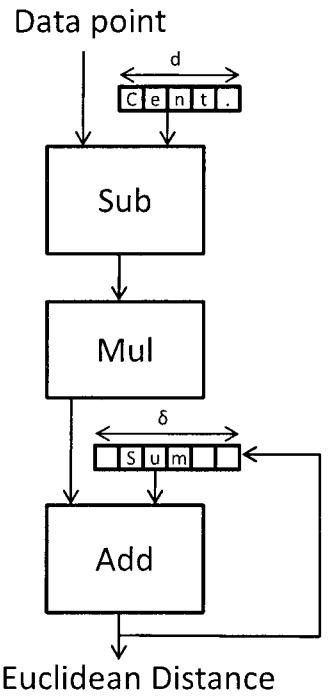


Fig. 16: K-means cluster unit

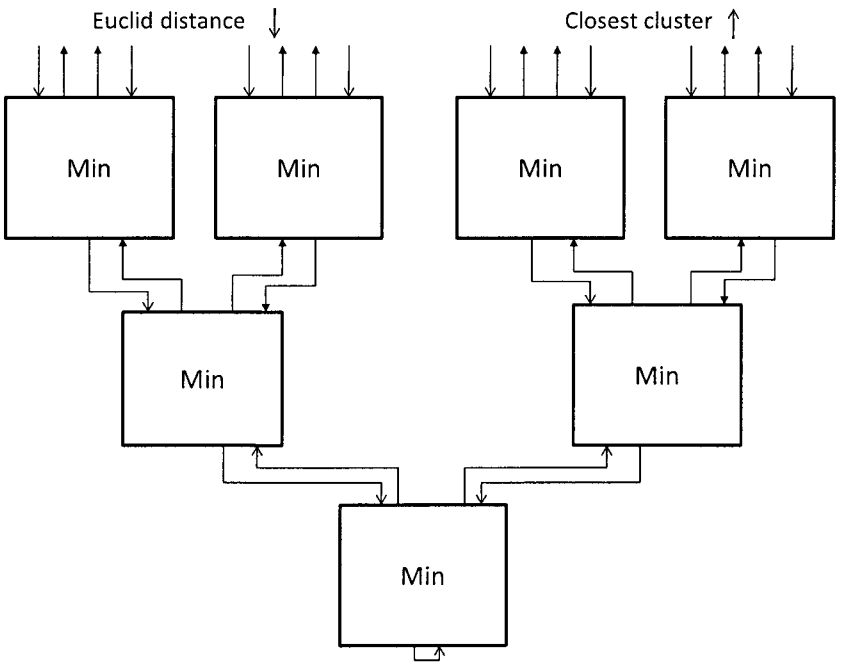


Fig. 17: Kmin unit

The Euclidean distances computed from the k clusters are presented to the $Kmin$ unit where the closest cluster is computed. The $Kmin$ unit has a binary tree structure as shown in Fig. 17 with $\log K^*$ stages where K^* is the extension of k to the nearest power of 2. The $Kmin$ unit compares the distances with one another determining the minimum in the last stage of the binary tree. Once the smallest value is found, the *cluster* unit yielding the smallest value is decoded by propagating the pointer to the top of the binary tree. The cluster center having the minimum of the Euclidean distances is the closest cluster. The given data point is then assigned to the new cluster center by updating the embedded header field of the data point. The header field for each data point contains $\log k$ bits at minimum for cluster assignment. The data point header field can also include additional 32 bits for the Euclidean distance with the assigned cluster centroid. The *cluster_update* unit is responsible for header update operations. Any changes in the cluster assignments of the data point will also increment the crossover count register in the *cluster_update* unit. A data point is latched in the data FIFO until the cluster assignment, after which it is removed from the data FIFO s and and sent to the packet generator where multiple such points are assembled in to new packets and transmitted to the output queues.

At the end of the iteration, the cluster centers required for the subsequent iteration are calculated by the *cluster_update* unit. The new cluster centers for the k clusters are determined by the average of the data points per cluster.

Cluster_update unit calculates the sum of all the data points assigned to a cluster. Since at most a single data point flows through the hardware pipeline, a single *cluster_update* unit would be sufficient to maintain information of all the k clusters. The cluster sum per iteration is calculated by adding the current data point with the previous sum of a given cluster. Dividing the individual cluster sum by the total number of elements assigned per cluster at the end of a iteration would give the cluster centroids useful for the subsequent iteration. The crossover count register is updated for every data point assignment to a different cluster than it previously belonged to. It is reset after the completion of a iteration. The crossover count register and the iteration count register are verified to terminate the algorithm if the crossover count is less than a threshold value, it means that there are only a few data points that were assigned to new clusters in the current iteration and current assignment is closer to the optimal assignment. The *cluster_update* unit contains a floating point adder, divider, K d-dimensional registers to hold the sum and k registers to hold the count of number of elements per cluster as shown in Fig. 18.

3.5.1 Optimization

Pipelined floating point add/subtract units on an FPGA have a latency anywhere from a single cycle to a 13 cycles for each operation. Similarly the multiplication takes one to eight cycles. On an FPGA, floating point implementations have a trade off between the area and latency. The lower the latency,

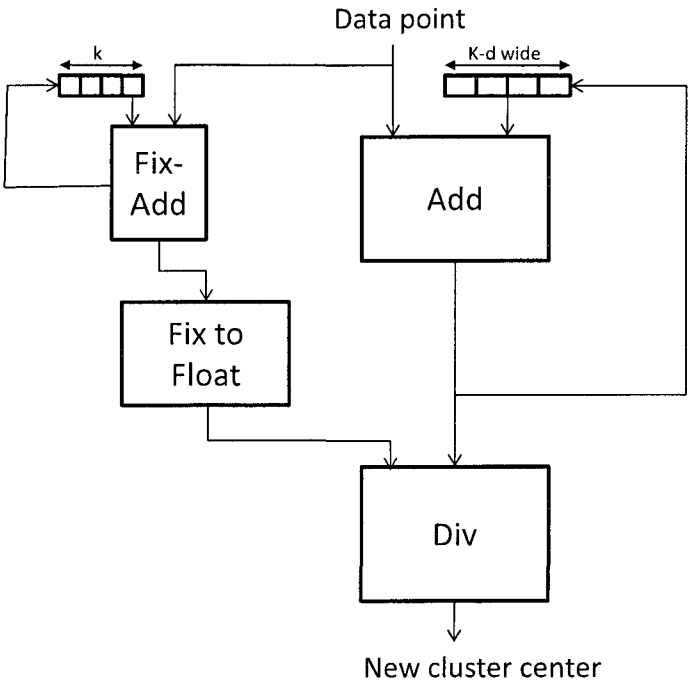


Fig. 18: Cluster update unit

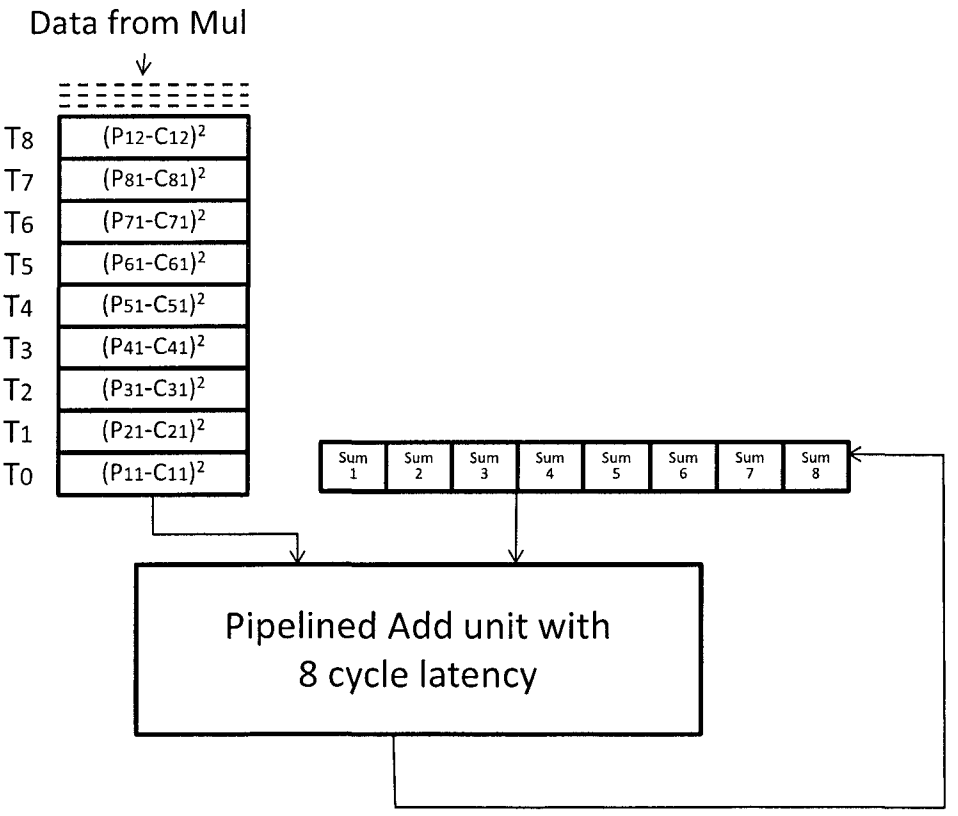


Fig. 19: Data interleave optimization

the larger the cycle time and the area. Euclidean distance calculations of a data point is data dependent with all the dimensions in the given point. For this reason, a straightforward implementation with a floating point adder unit would stall every dimension of the data point for the duration of the latency in the adder circuit. An optimization strategy would be to interleave the dimensions of different points together in such a way that the latency is hidden by performing calculations of unrelated data points before the results for the initial calculations are ready. An interleaving optimization for hiding eight cycle latency is shown in Fig. 19. Data point P_{ij} (where i is the point number and j the dimension), sent to the *cluster* unit shown in Fig. 16 is interleaved in such a way that , $i \ mod \ \delta$ (floating point add latency) varies from 1 to $\delta \ \forall \ j$. The floating point add unit in Fig. 19 receives the interleaved $|P_{ij} - C_{ij}|^2$ from the floating point multiplication unit in Fig. 16. The result of the partial sum $\sum |P_{ij} - C_{ij}|^2$ for a dimension j of the data point i will be ready after δ cycles. This result will be ready just in time for the add unit to perform another partial sum $\sum |P_{ij+1} - C_{ij+1}|^2$. Interleaved data required for this type of optimization could be easily generated at the data source or with the help of reorder buffers on the FPGA.

3.5.2 Parallelization techniques

The K-means algorithm on a single FPGA exhibits *cluster* level parallelism by performing Euclidean distance calculations belonging to all the *cluster* units

in parallel. For higher values of k, the amount of resources available on a single FPGA will not be sufficient to fit all k *cluster* units. For such cases, the number of cluster units on the FPGA is reduced to a fraction of k cluster units in a single step. The remaining *cluster* units are computed in multi steps with $\frac{k}{n}$ *cluster* units per step. The original *Kmin* unit is modified to find the minimum of Euclidean distances from of all the steps. The latency associated in the multi step approach could degrade the performance but in a worst case when $n = k$, the multi step approach becomes the sequential algorithm. In a multi FPGA environment, the additional FPGA resources can be used to implement the parallel algorithm in two ways. One way is to distribute data to every FPGA and perform cluster assignment in parallel. The other approach would be to distribute the *cluster* units to each of the FPGAs and perform Euclidean distance calculations in parallel. These two approaches are explained in the following sections.

3.5.3 Data level parallelism

In a multi-FPGA system, each FPGA can operate on separate data points thereby performing cluster assignments in parallel. Data level parallelism on a multi FPGA system is possible if each of the FPGAs implement all k *cluster* units. The data is uniformly partitioned across all the FPGAs, and each data partition is sent to separate boards where each calculate the Euclidean distances

and cluster assignments in parallel. Each FPGA board calculates the local cluster centroids based on the portioned data. At the end of each iteration, the local cluster centroids are broadcast to the other boards. Upon receiving this broadcast information along with the local centroid information which was broadcast earlier, new centroids at each board are calculated as the average of the received centroids with the local centroid. Upon cluster update, each board sends a cluster update acknowledgment indicating its readiness for the subsequent iteration. This process proceeds till the iteration termination criterion of number of iterations or sub threshold crossover count. Data level parallelism extracted this way would have higher performance but with the limitation on the number of clusters that can be fitted on each of the FPGA. A multi step approach with only a fraction of the total k clusters per board in each step can be utilized for higher values of k.

3.5.4 Cluster level parallelism

Another approach in extracting parallelism on a multi FPGA system is to partition the k clusters across the FPGAs. This approach is intended for higher values of k. Each FPGA operates in parallel on the entire data set calculating the distances between data points and the FPGA's subset of k *cluster* units. The local minimum of the Euclidean distances at each FPGA is broadcast to the other FPGAs where the global minimum is computed. The global minimum is used in resolving cluster assignment for the incoming data point. Since the data point is

applied simultaneously to all the FPGA boards, the global minimum calculated at each FPGA board should result in the same cluster assignments for the data point at all the FPGAs.

3.5.5 Results: K-means clustering

Several sets of experiments were run to explore parallelism on hardware with a single FPGA board and with a combination of multiple FPGA boards. The sequential and parallel versions of the software algorithm [40] were run to evaluate the performance improvements made by implementing k means algorithm on a network switch. All the machines used in the setup run Linux on a dual core AMD Opteron 1.8Ghz processor with a 40GB SATA drive and a gigabit NIC. A client-server program in C handles the data transfer operations. Test data up to 1GB is generated at random as an input to the algorithm and is distributed across several hosts for parallel software and hardware implementations. The time for data partitioning is not timed in all the experimental results. All the host machines are connected through an FPGA based network switch in a low cost 2-dilated flattened butterfly network [41].

The base NetFPGA switch design on Xilinx Virtex II Pro consumed 15572 slices taking up 65% of the available FPGA slices. Due to the limited resources available on the FPGA, we were able to fit up to a maximum of eight cluster units. For clusters over eight, we used the multi step approach discussed in 3.5.2

3.5.5.1 Single FPGA

In a single host scenario, one of the machines is used as both the host and server with the NetFPGA board as a co-processor. Data from the client is sent to the NetFPGA through a gigabit interface. The NetFPGA performs the k means clustering assignment and returns the data back to the host. These iterations are continued till the termination at which time the NetFPGA returns a packet to the host signaling termination. Results for a single host system in Fig. 20 show the speedup of the hardware implementation over software implementation for various cluster sizes and dimensions. The results show speed up 2-10 times that of the software implementation. As can be seen, since the FPGA can calculate several clusters in parallel, the speedup improved as we increase the number of clusters. The runtime per iteration of the algorithm in hardware roughly remained the same with a slight increasing trend towards the end for different number of clusters. However the software runtime software grew non-linearly after 8 clusters. This resulted in the superlinear speedups for cluster sizes beyond 8. The FPGA has a limit of 8 clusters, so for 16 and 32 clusters, we used a multi step approach. However for higher number of clusters, the multistep approach adds a sequential bottleneck in hardware. Thus for even higher number of clusters, the speedup may not grow linearly. The hardware implementation should still outperform the software algorithm. The results also show speedups independent of the number of dimensions.

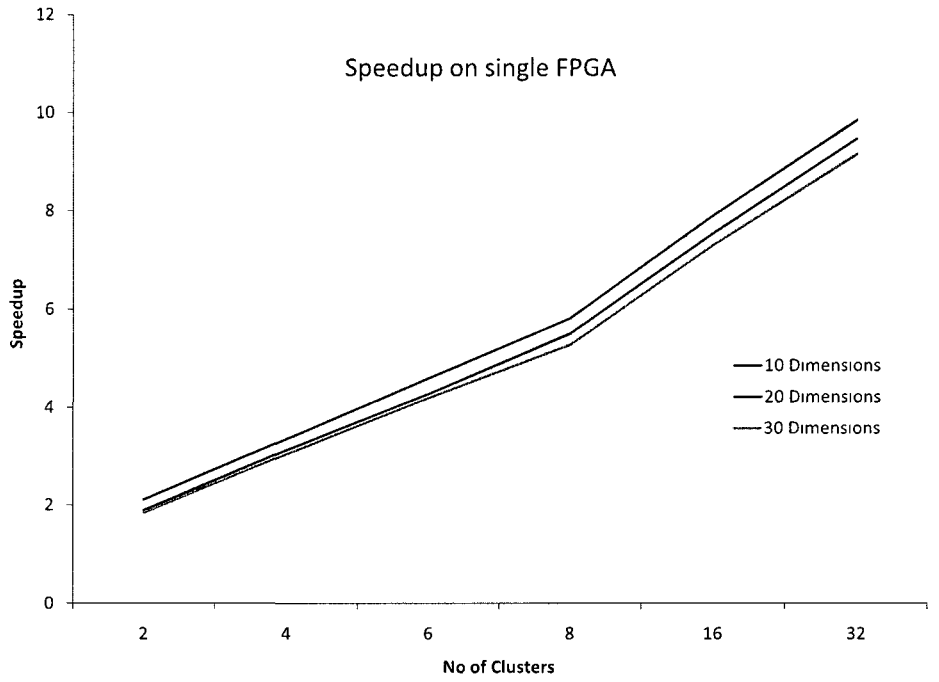


Fig. 20: Speedup on a single FPGA.

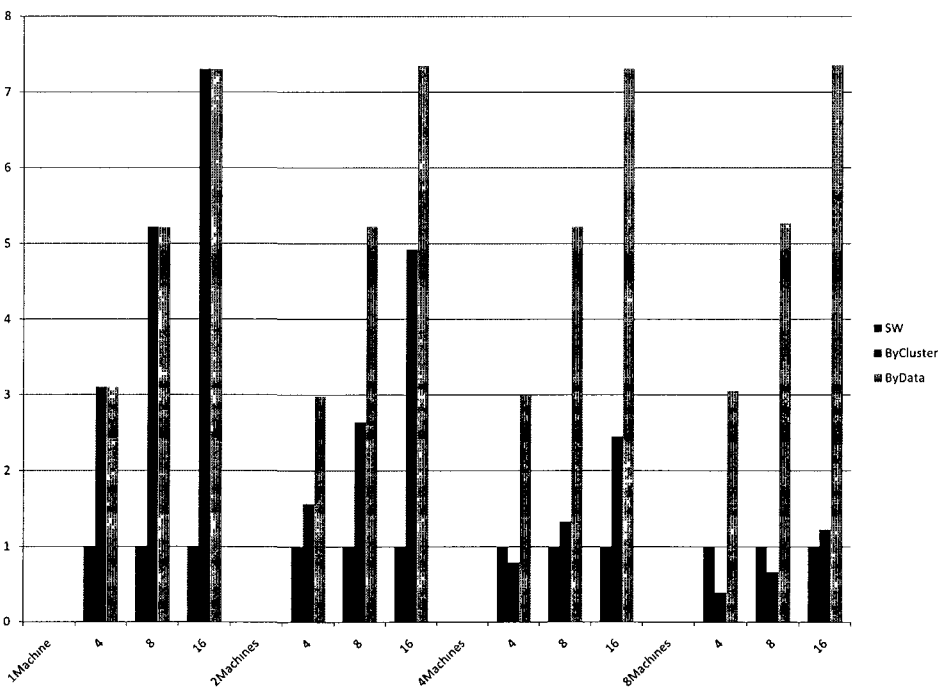


Fig. 21: Speedup on multiple FPGAs.

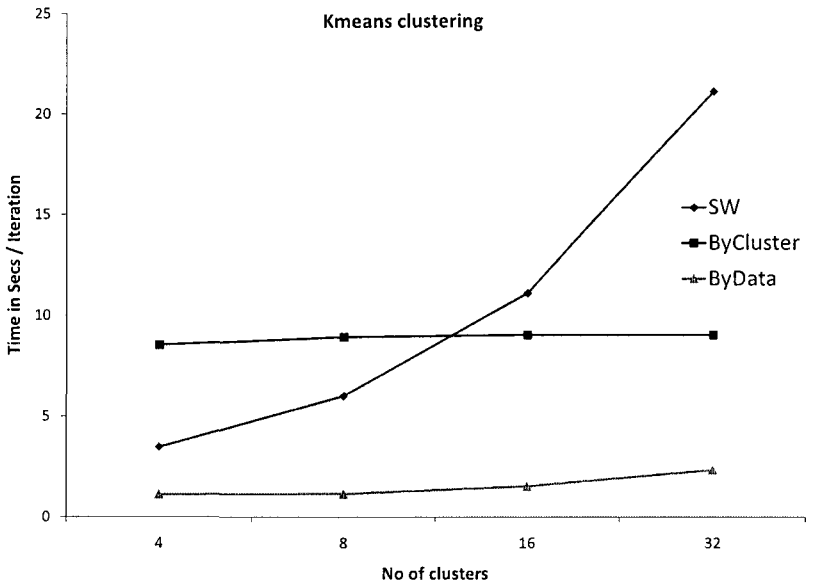


Fig. 22: Run time per iteration of K-means algorithm

3.5.6 Multiple FPGAs

We connected multiple FPGA boards in a 2-dilated flattened butterfly network to operate in parallel to implement the K-means algorithm. Sets of experiments with two, four and eight FPGA boards were conducted. An equivalent number of hosts act as storage nodes. A client server program on the hosts perform the data transfer operations on the partitioned data in the same manner as on a single FPGA system.

Equivalent parallel versions of the software program were run to compare the performance. During software implementations the the host nodes act as processing nodes and are connected in the same network as in the hardware implementations. The software algorithm utilizes MPICh2 for message parsing.

The results for the speedup due to the two parallelism techniques in hardware compared to the parallel software algorithm are shown in Fig. 21

Parallelism by data

Data is distributed across the n hosts and the k cluster computations are replicated on the multiple FPGAs. Speed up grew linearly with increase in clusters. Results show a speedup of around 9 times over software implementation [40]. As the data is distributed across multiple nodes, the bandwidth scales with the number of FPGAs and the speed over parallel software implementation remained constant with increase in FPGAs.

Parallelism by cluster

The K cluster computations are distributed across the multiple FPGAs and the data is replicated across all the storage nodes. The results show an increase in speedup with the number of clusters. The speedup however declined with increasing the number of FPGAs. This is because the bandwidth does not scale as we increase the number of FPGAs. The bandwidth of data coming in to the multi FPGA system remained a constant 1 Gbps for all the FPGAs. The benefit of cluster level parallelism will be evident as we increase the number of clusters. Figure 23 shows the runtime per iteration of execution for different k. The results show the data parallelism outperforming cluster level parallelism and a parallel

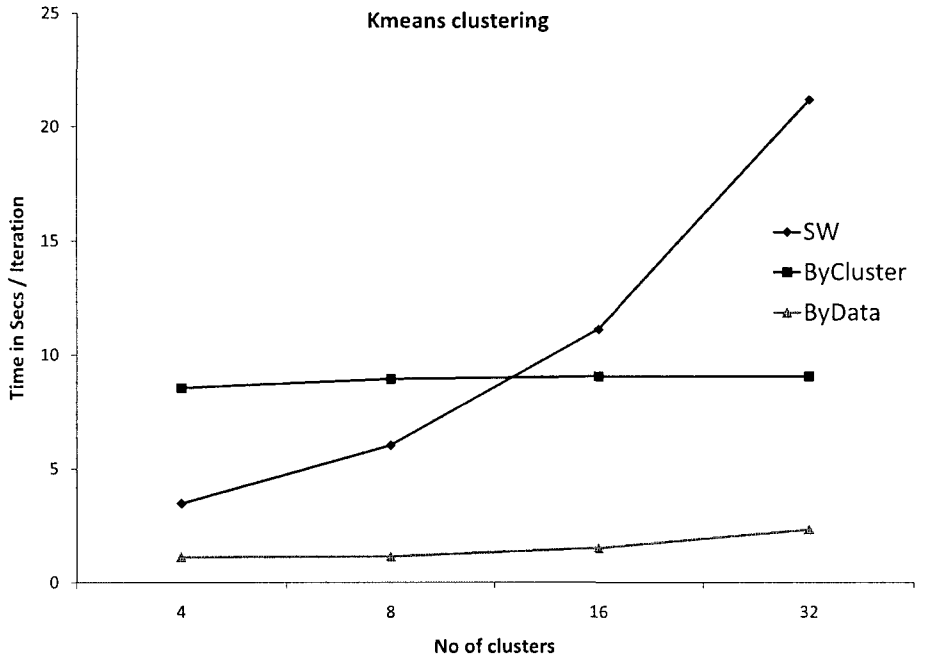


Fig. 23: Run time per iteration of K-means algorithm

implementation in software based on work in [40]. However for larger values of k the cluster level parallelism should catch up with data level parallelism.

Data level parallelism has better performance compared to cluster level parallelism for lower values of k. One of the issues with data level parallelism is the limit on the maximum number of clusters (k) that can be accommodated on the FPGA. A maximum of eight clusters units can fit on the XC2VP50 FPGA, and above eight clusters the multi step approach will be used. For a given set of available FPGA resources, cluster level parallelism offers maximum choice in terms of the number of cluster units that can be computed. For an an eight

FPGA network, using cluster level parallelism a maximum of 64 clusters can be computed before having to resort to the multi step approach.

Chapter 4

Hybrid Computing

4.1 Introduction

Microprocessor design in recent years has hit a clock cycle roadblock that has changed the design direction towards scaling the number of processor cores on the chip. Chip multiprocessors with 2-8 cores have become common in the commodity processor market. Intel has already demonstrated 80 cores in a 45 nm process [42]. Graphic processors have been even more aggressive as evidenced by NVidias recent GeForce 480 GPU with 480 cores on a 40 nm process. With processing technology expected to reach 4nm in the coming decade, hundreds of cores on a chip will be common. The technology road map pushes the limits of computation power - However, finding commodity applications that can fully utilize the hardware potential will be difficult. The difficulty with this future road map is how end users are supposed to use these processors. There are few data intensive and compute intensive parallel applications that can use these large

numbers of processors. Large-scale applications like genome sequencing, and data mining, earth simulation etc require the full computation power of several hundreds of processing cores. For regular desktop applications, the additional processing power could be overkill. In current multi-core CPUs, the primary usage of the extra cores has been to support multitasking - i.e. the many processes that run in the background such as virus checking, indexing,de- fragmentation, searching, downloading, etc. However, multitasking is unlikely to scale past about 8. The challenge is to develop applications that can go beyond multitasking and use parallelism to utilize the cores.

Decades of research have shown that parallelism is difficult to find in typical applications whether by hand coding or automated compiler techniques. The only applications that easily scale to 100s or 1000s of cores or those that can be decomposed into independent tasks or those that operate on independent sets of data. Examples of such mainstream applications include image editing, rendering, and search. However, overall, the set of applications that are easily parallelizable is limited. Therefore, going past 8-16 cores is unlikely to provide benefits to most application workloads.

One option to adequately use the silicon area is hybrid multi-core architectures. GPUs and Cell Broadband engines are some good examples of heterogeneous multi-cores where data streaming operations and data processing operations are handled in separate cores. The Cell Broadband Engine is a hybrid architecture where a POWER CPU is augmented with 8 Synergistic Processing

Elements that can handle various types of data streaming operations [43]. Other hybrid architectures like GPUs with hundreds of processing cores are gaining prominence. With 1000s of available cores, these architectures still do not take full advantage of the available transistors. We propose a reconfigurable hybrid architecture that merges traditional CPUs, SIMD-style GPU cores, and finally, a reconfigurable hardware fabric similar to that found in current FPGAs. Reconfigurable hardware can extract maximum parallelism out of a task as hardware structures that can perform bit level parallel operations on data can be easily built. Moreover the custom tailored design can ensure data parallel operations at a wider range of granularity without the context switching overhead as in multi-core processors. Reconfigurable devices have the best computational density for bit-level and integer operations while traditional CPUs do best for floating point operations [44,45]. A hybrid architecture integrating the reconfigurable fabric with processor can provide flexibility in implementing parallel applications and provide better utilization of resources on the chip. The computational cores can offload tasks to the reconfigurable cores on the hybrid system where several parallelization strategies can be explored. Such a framework presents several different ways of handling the task in hardware, software and a combination of both. The performance benefits of these various implementation depend on the available resources in the hybrid core, the amount of time a task spends on reconfigurable core, the various parallelization techniques used in implementing a task on the reconfigurable core etc. These are some of the factors that are to be considered

while offloading tasks in a hybrid system. Several resource allocation strategies for low power, performance, efficient resource utilization catering to the needs of application have to be developed. The remaining part of our work focuses on these aspects.

The integration of reconfigurable logic with CPUs is a well known idea and reconfigurable computing has been an active research area for many years. The primary characteristic of reconfigurable computing is the integration of a microprocessor with hardware that can be reprogrammed on the fly. Reconfigurable hardware include PALs, PLAs, smart memories, and field- programmable gate arrays (FPGAs). FPGAs provide the greatest amount of flexibility with programmable logic blocks connected by a sea of programmable interconnect. FPGAs have long been used to accelerate a variety of applications including network intrusion detection [11,46,47], multimedia [48,49], numerical algorithms [50], and encryption [51,52].

4.2 Reconfigurable processing architecture

There have been several reconfigurable computing architectures proposed by the research community and these architectures fall into two general classes - functional unit based and co-processor based. Functional unit type architectures are targeted toward fine grain computational optimizations, whereas co-processor architectures are more appropriate for large task offloading. In the context of many-core architectures, we feel the co-processor architecture is most appropriate.

Functional unit based reconfigurable computing takes a microprocessor and integrates the reconfigurable hardware as a functional unit within the microprocessor. Examples of these architectures include Chimaera [53], PRISC [54], and OneChip [55,56]. These reconfigurable functional units (RFU) execute custom instructions to provide speedups of short instruction sequences. The advantage of the RFU design is that the tight coupling with the processor core allows fast access to processor registers. With co-processor based designs, the reconfigurable hardware is distinct from the main processor core, in that it does not participate in the pipeline. However, it may use some processor functionality such as memory access or data caching. Examples of co-processor based research reconfigurable platforms include Garp [52,57] PipeRench [48], DISC [58], and PRISM [59]. Co-processor architectures have become common in commercial high performance computing systems such as the Cray XD1 and SGI RC100. In a variation of the co-processor architecture, Xilinx and Altera have both introduced FPGAs with embedded processor cores. Because the reconfigurable co-processor is separated from the processor core, sending data from the processor to the co-processor will typically incur additional latency. As a result, the algorithms that can be moved to the reconfigurable hardware must be large enough to amortize the data communication cost. This usually precludes the instruction level type kernels possible with RFUs. The advantage of a co-processor, though, is that it is easy to integrate with existing processor cores.

Reconfigurable computing architectures require both the design tool support and the operating system support in developing applications that can run in a hybrid system. These applications also require libraries of reconfigurable hardware kernels. Traditional software development depends on the availability of compilers that translate high level languages such as C, C++, or Java to machine language or intermediate byte-code. Linkers then link the compiler generated object code with appropriate libraries that implement either operating system functionality or commonly used functions. Operating system functions include such features as I/O, process manipulation, and memory allocation. Commonly used functions include higher levels of I/O such as streaming and stdio, GUI libraries, string manipulation, math functions, and others. For C programmers, the standard repository for these functions is the libc library and C++ programmers are familiar with the STL library. In a software application, the operating system resolves function calls to these libraries at runtime. In a co-processor based hybrid architecture, the processing core running the operating system would dynamically offload tasks in a running process to either HW or SW. The operating system must also dynamically choose between multiple implementations of various cores depending on the availability of resources on HW, performance/power issues etc.

4.3 Related work

The integration of reconfigurable logic with CPUs is a well known idea and reconfigurable computing has been an active research area for many years. Most of the work on SW/HW co design such as OpenFPGA [60,61] and PFIF [62] provide standardized software APIs to hardware IP cores. The RCLib set of cores has been developed towards a set of standard interface guidelines [63]. These APIs provide an interface between software and hardware thereby allowing software to call hardware functions by specifying mechanisms to pass data to and from the hardware. The APIs also allow software to work with hardware and allow hardware designers to develop portable IP core libraries. However, there are no mechanisms to manage the loading and unloading of cores, include software implementations or support multiple hardware implementations. Our work aim at providing at providing operating system support for task offloading

4.4 Operating system support

To provide generic operating system support, a shared library approach at offloading tasks from software to hardware is used. Operating systems use libraries to provide application a collection of frequently used functions. There are two types of libraries namely static and dynamic libraries. As the name implicates, static libraries are used at compile time by the compiler and linker to generate an executable. An executable with static linkage would contain all the

code for different function calls in itself. An executable with static linkage would therefore run faster as there is no need of resolving function calls. A dynamic shared library contains the code for different function calls which are resolved by the loader at runtime. A compiler using dynamic shared libraries will generate executable with a smaller executable size. However at runtime, the loader is responsible for function look up. The advantage of dynamic libraries is that common function that are used again and again need not reside in memory multiple times. A compiler compiles source code for an application typically written in higher level languages like C, C++ or Java, to a machine readable object code. The object code contain many references to function calls that are unresolved. The linker resolves the unresolved function calls to the functions available in shared libraries. Figure 24 shows the steps involved in generating an executable. A loader resolves these function calls to actual function implementations during the runtime. Various applications can use the functions in the shared library during the runtime. The disadvantage of dynamically shared library is that function look-up is slow. We extend the mechanism of dynamic shared library support for offloading task from software to hardware. A scheduler is implemented for task offloading to hardware as shown in figure 25. The scheduler makes a decision on implementing a given task in HW or SW and chooses the appropriate shared library to implement a given function. If a decision is made to run a task in HW, scheduler pick the appropriate function from the hardware shared library.

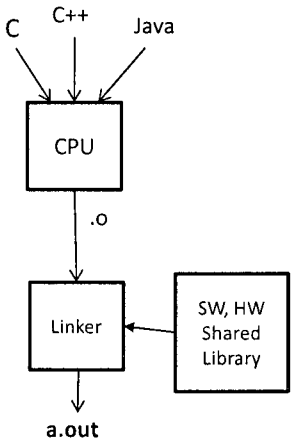


Fig. 24: Linking process

If the scheduler finds that a given task cannot be implemented in HW, the given function is implemented from the appropriate software shared library.

Shared library approaches need an extensive support of shared libraries that run in hardware. These libraries also provide a way to configure the hardware and move data between HW and SW.

On an FPGA, the hardware functions are implemented in higher level hardware descriptive languages such as Verilog, VHDL,etc.. These functions are synthesized to bit streams that can be easily programed to the configuration memory of FPGA. Generally when configuring an FPGA, the entire configuration is reloaded with a new bit stream. Recent techniques in FPGA's allow partially writing the configuration memory keeping the other parts of configuration memory intact. This feature allows runtime reconfiguration of FPGA. Some parts of the FPGA can be dynamically reconfigured while the other parts of FPGA are

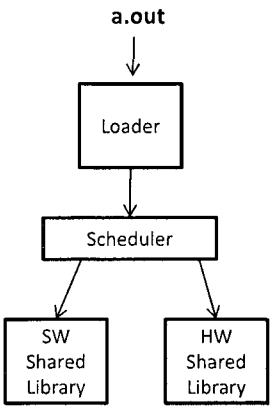


Fig. 25: Runtime process

still in operation. We explore this partial reconfiguration of FPGA combining the shared library feature of an operating system and the partial reconfiguration feature of FPGA, we implemented a prototype hardware system on a Xilinx Virtex 2P FPGA as shown in Figure 26. The Virtex 2P FPGA contains embedded Power-PC processor running a Linux 2.6 kernel with shared library support. A prototype wrapper hardware library was built. This wrapper can reconfigure an FPGA and return control back to the processor. It is also responsible for data transfer from hardware to the software. Though not implemented, we envision a DMA type of transaction for data transfer between hardware and software. A software application running on this processor can partially reconfigured parts of FPGA with the required hardware function of user choice. The prototype hardware implementation contains several partially reconfigurable regions for the user as shown in the Figure 26. The embedded processor utilizes the internal

configuration access port (ICAP) for partial reconfiguration. As an example let function A need to be executed in hardware. This function has a software implementation given by a shared library say libA.so.1. If we need to offload this task to hardware, The functional reference to the the function A need to be resolved to the hardware instead of libA.so.1. Before a hardware implementation, the FPGA need to be properly configured with the bit-files corresponding to the function A. A wrapper shared library libwrapA.so.1 is implemented for hardware configuration. This library has the same function declarations as that of the software library libA.so.1. We put the hardware wrapper library in a higher precedent library path such that the loader program searches this wrapper library first for functionA. When executing a binary, the loader, thus, resolves the function call in the binary to the function declared in the wrapper library instead of the software library.

4.5 Scheduling algorithm

Specialized structures in hardware can accelerate specific tasks in hardware over their software counterparts. However, in an operating system running several tasks, accelerating all the tasks in hardware is not possible due to limitations in the amount of space available in reconfigurable hardware. To achieve a global optimal runtime, a select few tasks have to be run in the hardware. The remaining tasks have to be run in software. A decision on implementing the tasks in hardware or software can be made by an operating system. For a normal process

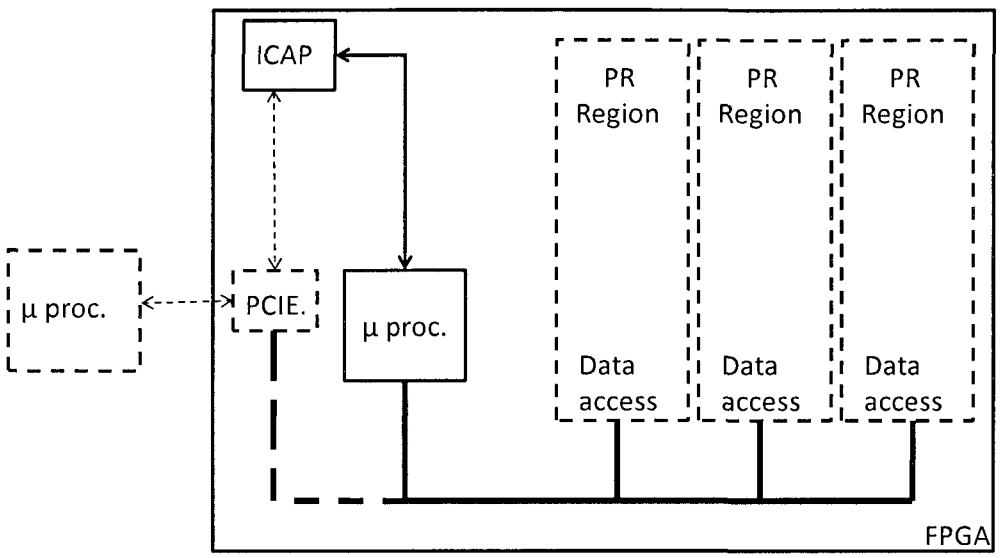


Fig. 26: Prototype hybrid system

executable, a standard operating system will load the executable into memory, pass program control over to the new process, and then manage the scheduling of the process. In the same way, reconfigurable aware OS loads executables, begins the execution, and then manages the scheduling of the task. A scheduling algorithm can make this decision statically or at runtime to accelerate overall application performance in a system. A superior reconfigurable scheduling algorithm relies on scores of hardware libraries available for accelerating tasks. These libraries include several versions of hardware bit-streams with different space, speedup requirement for a same a processing unit. Inclusion of one version of the library over the other in the reconfigurable hardware would have a different

impact on the overall run time of an application. A significant decision that the OS must make is whether to 1) keep the existing configuration and let the new function run in software. 2) remove the existing configuration and let the new function run in hardware thereby consigning the old function back to software or 3) use smaller configurations of both functions. This question must be answered with several metrics in mind including performance of each thread, priority of each thread, CPU load of each thread, and energy usage.

Our initial approach to this problem is to treat this as a resource allocation problem both in time and space. Time being the amount of time that the thread is expected to require use of the shared reconfigurable resource. Space being the amount of reconfigurable resources being used. This allocation problem is distinct from configuration scheduling in FPGAs where a large circuit must be partitioned and scheduled over time to fit in the available space. The allocation problem that we are looking at is similar to job scheduling on a compute cluster except for the constraint that not all allocations are viable. For example, on a compute cluster, for the most part it does not matter which nodes are assigned to a task. However, in a reconfigurable array it is critical that the rows of the array are assigned to match the available implementations for the desired function. Thus, the allocated rows must be contiguous. As mentioned, resource and task allocation in the context of job scheduling has been well studied [64–66]. We will draw on this existing work and heuristics specific to allocation problem. The primary objective is execution time. Thus, the allocation must ensure that the

highest priority thread finishes first. This requires estimates of computation time for each thread, and the runtime for each implementation of the function - both hardware and software. These estimates can be provided as part of the scheduling process. Thus, the parameters of interest are:

- T_{i0} the expected overall run time for thread i assuming a software implementation.
- f_{ij} fraction of time that thread i spends in computation kernel j assuming a software implementation.
- t_{jk} the execution time of the kth implementation of computation kernel j where implementation, 0 is the software implementation.
- L_{jk} the space resources of the kth implementation of computation kernel j.
- l_{jk} time to load the kth implementation of computation kernel j into the hardware.
- t_u time to unload the existing configurations and its state.

T_{i0} is the baseline execution time for thread i, and the use of hardware can reduce this time. The execution time for a thread is:

$$T_i = T_{i0} - \sum_{j=1}^{J} f_{ij} Ti0 + \sum_{j=1}^{J} \left(\frac{f_{ij} T_{i0}}{t_{j0}} t_{jk'} + l_{jk'} \right) \quad (1)$$

where J is the number of computational kernels used by a thread and k' is the selected implementation of a kernel function. The objective is to minimize

T_i for all threads given assigned priorities and choosing implementations and hardware allocations to meet this goal. A simple algorithm would be to iteratively start with the highest priority thread and minimize its T_i and then minimize the next priority thread until no more threads can be minimized. If T_i is relatively short, this might be appropriate. If, however, T_i is very long but f_{ij} is small, it might make sense to swap out the hardware function for a different thread depending on l_{jk}. The objective function must also take into account the unload time, t_u, and its relationship to the improvement in Ti because if the unload time is too long, it may not be worth reconfiguring the hardware. The objective function can be altered to account for energy usage if being used in a power constrained environment. For example, hardware implementations will typically have higher power requirements, but since the execution time is much faster, the overall energy usage may be lower. Thus, constraints on the allocation and decision problem include overall energy usage and peak power utilization. Using a shared library approach we implemented a scheduler shown in figure 25. The scheduler makes a runtime decision on implementing a given task in hardware or software. We also provide two scheduling algorithms for task allocation problem.

4.5.1 Static scheduling

Static scheduling makes a decision on implementing a given task in HW before an application begins execution . The algorithm relies on profiling information about a given application. This information include the total runtime of

an application in software, the number of functions executed and the amount of time spent in each of the functions. This information is used in formulating a cost function in determining the scheduling process. The cost function used by the scheduling algorithm is given by equation 1. The cost function typically has a negative value. Adding a new task k can alter the total runtime of an application given by equation 2.

Static scheduling precedes by ordering all the tasks by their increasing order of the cost function given in equation 2. This ordering process has a complexity of $O(n \log n)$ where n is the total number of tasks available for hardware implementation. Once the tasks are ordered, the sorted tasks are filled in the hardware. Due to the space limitation on hardware, a subset k of the total tasks can be filled in hardware. the remaining $k - n$ tasks have to executed in software. The tasks that are filled in the hardware stay in hardware for the runtime of the application or a new static scheduling process is initiated. During the runtime of an application, the job of the scheduler is to check weather a given task is available in hardware and is not running. A table of running tasks can be formulated while sorting the tasks and filling the hardware. During runtime, the table lookup can be done in constant time.

$$f_{ik} T_{i0} \left(\frac{1}{Speedup_k} - 1 \right) + l_k \tag{2}$$

The advantage of static scheduling is that the runtime decision making process has a complexity of $O(1)$. However, when the total task list is large or when the chosen tasks in HW run at the end of application, static scheduling looses out further opportunities of running an application in hardware. Dynamic scheduling provides an alternative to this problem.

4.5.2 Dynamic scheduling

Dynamic scheduling offers a flexible option of choosing the tasks at runtime. The cost function used for dynamic scheduling is the same as for static scheduling given in equation 2. Just as in static scheduling, profiling information about the total runtime of the application and the frequencies of functions executed are used in making runtime decisions on implementing the functions in hardware.

The dynamic scheduling algorithm proceeds by filling the hardware with tasks as they are encountered in the application. While filling the tasks in hardware, a table of the running tasks and total cost function of all the tasks in hardware is noted in a table. The tasks are filled in the hardware as long as there is space available. Once all the hardware resources have been used new tasks are executed in hardware if adding the new task would lower the overall runtime. For a task to be replaced in the hardware, the running task should have higher cost function than the task that replaces it. Choosing the right task to remove from the hardware is tricky. Another requirement for removing a task is that the space occupied by the tasks in hardware should match the new task. If removing

one task does not provide sufficient space for the new task to be put in hardware, multiple tasks from hardware have to be removed. Tasks that are currently not running in hardware are preferred over the tasks that are currently running in hardware.

The runtime for each schedule decision in $O(k)$, where k is the number of tasks in hardware. The advantage of dynamic scheduling is that when the number of tasks are more, dynamic scheduling can provide better utilization of hardware than static scheduling.

4.6 Results

A scheduler for static and dynamic task allocation is written in C++. The scheduler is compiled as a shared library with all necessary functions for a given workload. We ran our scheduler on a GEMS simulator [67]. GEMS is a set of Ruby and Opal modules built on top of Virtutech's SIMICS [68] cycle-accurate simulator. The ruby and opal modules provide accurate simulator modules for modeling the timing of the memory system and microprocessor. The simulator ran a Linux 2.6 kernel on Sun sunfire 6500 machine with two ultraSPARCII processors at 168 MHz. The Linux kernel is augmented with all the GNU tool support. We chose a LLR function for a target tracking application [69] as a work load. The work load accounted for over 95% of the total target tracking application. The workload characteristics of the LLR function is given in the table 1 . The table shows various kernels required in the given workload and

kernel	Area occupied	Speedup	Active percent
SIN	8%	12	0.6
COS	8%	11	1.6
ArcTAN	8%	12	3.0
SQRT	1%	2	2.0
EXP [70]	8%	13	91.6
LOG [71]	10%	73	1

Table 1: Workload characteristics of an LLR search function

the percentage of time spent on each kernel. The table also contains hardware implementation results of area occupied speed up over corresponding software counterparts. These kernels are double precision floating point units based on cordic algorithms. Cordic floating point cores were implemented in hardware for SIN, COS, ATAN, and SQRT functions. The results for hardware implementation of EXP and LOG cores were taken from work done by Pottathuparambil et al. [70] and Zhou et al. [71], respectively. The hardware is modeled to run a Virtex 5 XC5VLX50T FPGA running at 100 MHz.

The results for scheduling implementation are shown in Figure 27. The figure show the number of cycles taken to run the given workload on software, and on hardware with static and dynamic scheduling. When running a software implementations, the kernels in table 1 were run from libmath software shared library. The hardware results were accurately modeled based on the results from the table 1. The results show that static scheduling run around 6 times faster than the software implementation. The dynamic scheduling yielded a 4 times speed up

over the software runtime. The reduction in the speedup for dynamic reconfiguration is due to the overhead in task lookup. Since EXP function accounted for about 92% of the total workload, the speedup due to hardware implementation of the work load, ideally should have been close to 13 times. However, the overhead in shared library lookup and the load time of FPGA resulted in reduction of speedup to 6 times. The Virtex 5 XC5VLX50T FPGA has a load time of 439136 cycles. Shared library lookup and static scheduling brought down the speedup of EXP function to about 7 times. The workload ran for about 26512188068 cycles. The total overhead for static scheduling was 2379298929 cycles and for dynamic scheduling, the overhead was 4523905107 cycles. A special application written for task offloading could improve the speedup but a shared library approach provides a general purpose and easily portable way of going forward to offload tasks to reconfigurable hardware.

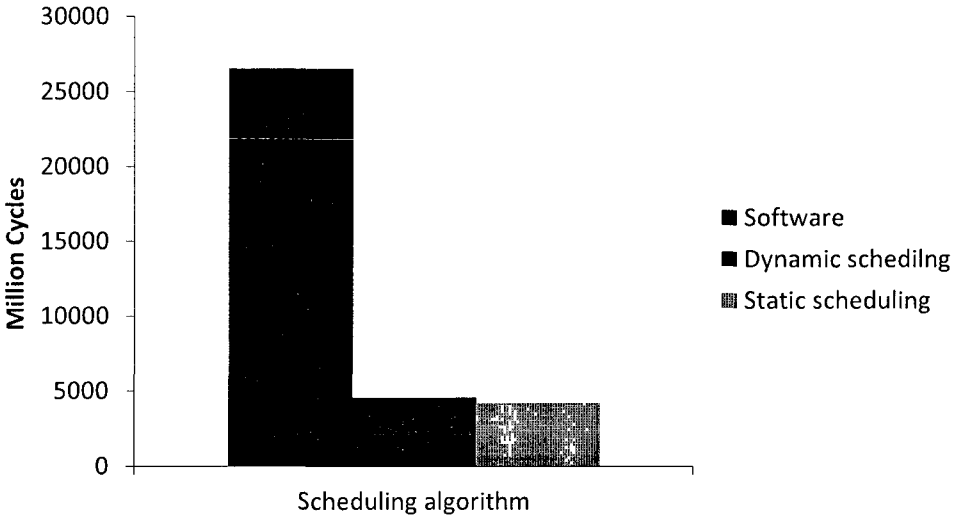


Fig. 27: Scheduling algorithm runtime results

Chapter 5

Conclusions

We implemented several data processing kernels for data search, data sort, Kmin/Kmax and K-means clustering in an ASN built on a 2-dilated flattened butterfly network. These processing kernels were implemented on an active switch built on a NetFPGA development board. We evaluated the performance benifits of application processing in ASN and show how data reduction and transformation applications can take advantage of the idea of ASN. In our other work, we focused on managing multiple hardware kernels on reconfigurable hardware. We built a prototype hybrid computing framework on a Virtex-2P FPGA using the partial reconfiguration feature of the FPGA. We provided shared library support for Linux 2.6 kernel capable of dynamically offloading tasks to the Virtex 2P FPGA. We also developed static and dynamic scheduling algorithms for task management between HW and SW.

5.1 Outcome and Significance

By implementing stream processing in an ASN, we hope to provide a novel method of application processing for large scale data in storage networks. Several large scale data processing applications in the fields of Image/Video processing, data mining, file system optimizations etc can accelerate processing performance by the concept of ASN. Offloading computations from the client to the network can yield several benefits. Network processing can speedup application processing considerably. Hardware in the network can accelerate applications by exploring parallelism in the application through special structures built for the specific application.These oprimizations are not possible in software implementation.The client side computation burden to keep up in pace with the I/O can be eased. Application processing in network can take control of the flow of data in network therby reduce I/O deficiencies while performing transformation and reduction operations. Advancements in parallel I/O push the bottleneck in computation from I/O to the compute nodes, our work can strike a balance between the two by providing I/O resources with computation abilities. On the hybrid computing front, our work would facilitate operating system with dynamic loading support for offloading computations to reconfigurable hardware. Hybrid computing is still in its infancy and has several roadblocks before it becomes mainstream. One of the issues with hybrid computing is the lack of tools support for handling tasks between hardware and software. We hope to provide a step forward in this

area by providing a shared library approach at offloading tasks to reconfigurable hardware. We also developed static and dynamic scheduling algorithms that use the shared library approach to manage tasks accross HW and SW. A generic operating system support for handling tasks would enable researchers to develop hardware libraries that can be easily ported accross several systems.

5.2 Future Research

Current work on ASN supports only a single application on the ASN at a time mainly due to the area limitation of the FPGA. Another limitation of the ASN is that configuration of the FPGA requires the entire network to be shutdown during the reconfiguration process. Partial reconfiguration support on the network would also need appropriate protocol support for automatic reconfiguration process and application management across various elements on the network. Security in an open network is implemented by encrypting and decrypting network data. ASN processing would require special decryption engines on the switches to retrieve valid data from encrypted data packets. We hope to implement these features as part of future work. Dynamic reconfiguration on current FPGAs suffer due to the large reconfiguration time. Preloaded FPGAs with appropriate libraries could overcome this problem. As a future work, we hope to develop several hardware libraries for common data processing applications and prefetch algorithms for hiding the reconfiguration time.

Bibliography

[1] J. S. Vitter, "External memory algorithms and data structures: dealing with massive data," *ACM Comput. Surv.*, vol. 33, pp. 209–271, June 2001.

[2] L. Arge, G. S. Brodal, and L. Toma, "On external-memory mst, sssp and multi-way planar graph separation," *J. Algorithms*, vol. 53, pp. 186–206, November 2004.

[3] T. H. Cormen and E. R. Davidson, "Fg: A framework generator for hiding latency in parallel programs running on clusters," in *ISCA PDCS*, pp. 137–144, 2004.

[4] D. Ajwani, R. Dementiev, and U. Meyer, "A computational study of external-memory bfs algorithms," in *Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm*, (New York, NY, USA), pp. 601–610, ACM, 2006.

[5] http://sortbenchmark.org. http://sortbenchmark.org.

[6] G. A. Gibson and R. V. Meter, "Network attached storage architecture," *Commun. ACM*, vol. 43, no. 11, pp. 37–45, 2000.

[7] A. Acharya, M. Uysal, and J. Saltz, "Active disks: Programming model, algorithms and evaluation," in *Proceedings of International Conference on Architectural Support for programming Languages and Operating Sustems*, 1998.

[8] E. Riedel, C. Faloutsos, G. A. Gibson, and D. Nagle, "Active disks for large-scale data processing," *IEEE Computer*, vol. 34, no. 6, pp. 68–74, 2001.

[9] F.-Y. Leu, M.-C. Li, J.-C. Lin, and C.-T. Yang, "Detection workload in a dynamic grid-based intrusion detection environment," *J. Parallel Distrib. Comput.*, vol. 68, no. 4, pp. 427–442, 2008.

[10] Y. H. Cho and W. H. Mangione-Smith, "Deep network packet filter design for reconfigurable devices," *ACM Trans. Embed. Comput. Syst.*, vol. 7, no. 2, pp. 1–26, 2008.

[11] I. Sourdis and D. Pnevmatikatos, "Pre-decoded CAMs for efficient and high-speed NIDS pattern matching," in *FCCM '04: Proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, (Washington, DC, USA), pp. 258–267, IEEE Computer Society, 2004.

[12] J. Singaraju and J. A. Chandy, "FPGA based string matching for network processing applications," *Microprocess. Microsyst.*, vol. 32, no. 4, pp. 210–222, 2008.

[13] B. Parhami and D.-M. Kwai, "Data-driven control scheme for linear arrays: Application to a stable insertion sorter," *IEEE Transactions on Parallel and Distributed Systems*, vol. 10, pp. 23–28, 1999.

[14] M. Bednara, O. Beyer, J. Teich, and R. Wanka, "Tradeoff analysis and architecture design of a hybrid hardware/software sorter," in *Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures, and Processors*, ASAP '00, (Washington, DC, USA), pp. 299–, IEEE Computer Society, 2000.

[15] J. Martinez, R. Cumplido, and C. Feregrino, "An FPGA parallel sorting architecture for the Burrows Wheeler transform," in *RECONFIG '05: Proceedings of the 2005 International Conference on Reconfigurable Computing and FPGAs (ReConFig'05) on Reconfigurable Computing and FPGAs*, (Washington, DC, USA), p. 17, IEEE Computer Society, 2005.

[16] R. Marcelino, H. Neto, and J. M. P. Cardoso, "Sorting units for FPGA-based embedded systems," in *Distributed Embedded Systems: Design, Middleware and Resources* (B. Kleinjohann, L. Kleinjohann, and W. Wolf, eds.), pp. 11–22, Boston, MA: Springer, 2008.

[17] M. Estlick, M. Leeser, J. Theiler, and J. J. Szymanski, "Algorithmic transformations in the implementation of k- means clustering on reconfigurable hardware," in *Proceedings of the 2001 ACM/SIGDA ninth international symposium on Field programmable gate arrays*, FPGA '01, (New York, NY, USA), pp. 103–110, ACM, 2001.

[18] A. Goswami and R. Jin, "Fast and exact out-of-core k-means clustering," in *In ICDM*, pp. 83–90, IEEE Computer Society, 2004.

[19] A. M. Dan Pelleg, "Accelerating exact k-means algorithms with geometric reasoning," in *Proceedings of the Fifth International Conference on Knowledge Discovery in Databases* (S. Chaudhuri and D. Madigan, eds.), pp. 277–281, AAAI Press, aug 1999.

[20] T. Maruyama, "Real-time k-means clustering for color images on reconfigurable hardware," *Pattern Recognition, International Conference on*, vol. 2, pp. 816–819, 2006.

[21] A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, and M. Snir, "The NYU Ultracomputer — designing a MIMD, shared-memory parallel machine (extended abstract)," in *ISCA '82: Proceedings of the 9th annual symposium on Computer Architecture*, (Los Alamitos, CA, USA), pp. 27–42, IEEE Computer Society Press, 1982.

[22] http://www.netfpga.org. http://www.netfpga.org.

[23] J. A. C. Ajithkumar Thamarakuzhi, "Scaling the netfpga switch using aurora over sata," in *NetFPGA Workshop*, 2010.

[24] A. Thamarakuzhi and J. A. Chandy, "2-dilated flattened butterfly: A non-blocking switching network.," in *HPSR*, pp. 153–158, IEEE, 2010.

[25] A. Thamarakuzhi and J. A. Chandy, "2-dilated flattened butterfly: A non-blocking switching topology for high-radix networks," *Computer Communications*, vol. 34, no. 15, pp. 1822 – 1835, 2011.

[26] A. Thamarakuzhi and J. A. Chandy, "Adaptive load balanced routing for 2-dilated flattened butterfly switching network," *International Conference on Networking*, 2011.

[27] J. Naous, D. Erickson, G. A. Covington, G. Appenzeller, and N. McKeown, "Implementing an OpenFlow switch on the NetFPGA platform," in *Proceedings of the ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, pp. 1–9, 2008.

[28] Xilinx, Inc., "Aurora link-layer protocol."

[29] Xilinx, Inc., "LogiCORE IP Aurora 8B/10B v4.2 user guide," June 2009.

[30] J. Kim, W. J. Dally, and D. Abts, "Flattened butterfly: A cost-efficient topology for high-radix networks," in *In Proc. of the International Symposium on Computer Architecture (ISCA)*, pp. 126–137, June 2007.

[31] R. M. Karp and M. O. Rabin, "Efficient randomized pattern-matching algorithms," *IBM J. Res. Dev.*, vol. 31, no. 2, pp. 249–260, 1987.

[32] D. E. Knuth, J. James H. Morris, and V. R. Pratt, "Fast pattern matching in strings," *SIAM Journal on Computing*, vol. 6, no. 2, pp. 323–350, 1977.

[33] R. Boyer and J. Moore, "A fast string searching algorithm," *Communications ACM*, 1977.

[34] A. V. Aho and M. J. Corasick, "Efficient string matching: an aid to bibliographic search," *Commun. ACM*, vol. 18, June 1975.

[35] N. K. Govindaraju, J. Gray, R. Kumar, and D. Manocha, "Gpu terasort: High performance graphics co-processor sorting for large database management," in *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, (Chicago, IL), 2006.

[36] S. Arora, P. Raghavan, and S. Rao, "Approximation schemes for euclidean k-medians and related problems," in *In Proc. 30th Annu. ACM Sympos. Theory Comput*, pp. 106–113, 1998.

[37] P. K. Agarwal and C. M. Procopiuc, "Exact and approximation algorithms for clustering," 1997.

[38] X. Wang and M. Leeser, "K-means clustering for multispectral images using floating-point divide," in *Proceedings of the 15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, (Washington, DC, USA), pp. 151–162, IEEE Computer Society, 2007.

[39] A. G. da S. Filho, A. C. Frery, C. C. de Araújo, H. Alice, J. Cerqueira, J. A. Loureiro, M. E. de Lima, M. das Graças S. Oliveira, and M. M. Horta, "Hyperspectral images clustering on reconfigurable hardware using the k-means algorithm," in *Proceedings of the 16th symposium on Integrated circuits and systems design*, SBCCI '03, (Washington, DC, USA), pp. 99–, IEEE Computer Society, 2003.

[40] S. W. Son, S. Lang, P. Carns, R. Ross, R. Thakur, B. Ozisikyilmaz, P. Kumar, W. K. Liao, and A. Choudhary, "Enabling active storage on parallel I/O software stacks," in *Proceedings of 26th IEEE Conference on Mass Storage Systems and Technologies (MSST 2010)*, 2010.

[41] A. Thamarakuzhi and J. A. Chandy, "2-dilated flattened butterfly: A nonblocking switching network," in *11th International Conference on High Performance Switching and Routing (HPSR 2010)*, (Texas, USA), 2010.

[42] S. K. Jim Held, Jerry Bautista, "From a few cores to many: A tera-scale computing research overview," *Citeseer*, p. 12, 2006.

[43] M. Gschwind, H. P. Hofstee, B. Flachs, M. Hopkins, Y. Watanabe, and T. Yamazaki, "Synergistic processing in cell's multicore architecture," *IEEE Micro*, vol. 26, pp. 10–24, March 2006.

[44] J. Williams, C. Massie, A. D. George, J. Richardson, K. Gosrani, and H. Lam, "Characterization of fixed and reconfigurable multi-core devices for application acceleration," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 3, pp. 19:1–19:29, November 2010.

[45] R. G. Williams, George and Suresh, "Computational density of fixed and reconfigurable multi-core devices for application acceleration," in *Proceedings of Reconfigurable Systems Summer Institute*, 2008.

[46] Z. K. Baker and V. K. Prasanna, "Automatic synthesis of efficient intrusion detection systems on fpgas," *IEEE Trans. Dependable Secur. Comput.*, vol. 3, pp. 289–300, October 2006.

[47] J. Singaraju, L. Bu, and J. A. Chandy, "A signature match processor architecture for network intrusion detection," in *Proceedings of the 13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, (Washington, DC, USA), pp. 235–242, IEEE Computer Society, 2005.

[48] S. C. Goldstein, "Piperench: A coprocessor for streaming multimedia acceleration," 1999.

[49] K. Denolf, A. Chirila-Rus, R. D. Turney, P. R. Schumacher, and K. A. Vissers, "Memory efficient design of an mpeg-4 video encoder for fpgas," in *FPL*, pp. 391–396, 2005.

[50] C. He, W. Zhao, and M. Lu, "Time domain numerical simulation for transient waves on reconfigurable coprocessor platform," *Field-Programmable Custom Computing Machines, Annual IEEE Symposium on*, vol. 0, pp. 127–136, 2005.

[51] H. J. Kim and W. H. Mangione-Smith, "Factoring large numbers with programmable hardware," in *Proceedings of the 2000 ACM/SIGDA eighth international symposium on Field programmable gate arrays*, FPGA '00, (New York, NY, USA), pp. 41–48, ACM, 2000.

[52] J. R. Hauser and J. Wawrzynek, "Garp: a mips processor with a reconfigurable coprocessor," in *Proceedings of the 5th IEEE Symposium on FPGA-Based Custom Computing Machines*, pp. 12–, IEEE Computer Society, 1997.

[53] S. Hauck, T. W. Fry, M. M. Hosler, and J. P. Kao, "The chimaera reconfigurable functional unit," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 12, pp. 206–217, February 2004.

[54] R. Razdan and M. D. Smith, "A high-performance microarchitecture with hardware-programmable functional units," in *Proceedings of the 27th annual*

international symposium on Microarchitecture, MICRO 27, (New York, NY, USA), pp. 172–180, ACM, 1994.

[55] C. Ralph, R. D. Wittig, R. D. Wittig, and R. D. Wittig, "Onechip: An fpga processor with reconfigurable logic," in *In IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 126–135, 1995.

[56] J. A. Jacob and P. Chow, "Memory interfacing and instruction specification for reconfigurable processors," in *Proceedings of the 1999 ACM/SIGDA seventh international symposium on Field programmable gate arrays*, FPGA '99, pp. 145–154, ACM, 1999.

[57] T. J. Callahan, J. R. Hauser, and J. Wawrzynek, "The garp architecture and c compiler," *Computer*, vol. 33, pp. 62–69, April 2000.

[58] M. J. Wirthlin, "A dynamic instruction set computer," in *Proceedings of the IEEE Symposium on FPGA's for Custom Computing Machines*, (Washington, DC, USA), pp. 99–, IEEE Computer Society, 1995.

[59] P. M. Athanas and H. F. Silverman, "Processor reconfiguration through instruction-set metamorphosis," *Computer*, vol. 26, March 1993.

[60] M. Wirthlin, D. Poznanovic, P. Sundararajan, A. Coppola, D. Pellerin, W. Najjar, R. Bruce, M. Babst, O. Pritchard, P. Palazzari, and G. Kuzmanov, "Openfpga corelib core library interoperability effort," *Parallel Comput.*, vol. 34, May 2008.

[61] M. Wirthlin, D. Poznanovic, P. Sundararajan, A. Coppola, D. Pellerin, W. Najjar, R. Bruce, M. Babst, O. Pritchard, P. Palazzari, and G. Kuzmanov, "Openfpga general api specification 0.4 (draft for comment),," July 2008.

[62] M. Huang, I. Gonzalez, S. López-Buedo, and T. A. El-Ghazawi, "A framework to improve ip portability on reconfigurable computers," in *ERSA*, pp. 191–197, 2008.

[63] P. Saha, E. El-Araby, M. Huang, M. Taher, S. Lopez-Buedo, T. El-Ghazawi, C. Shu, K. Gaj, A. Michalski, and D. Buell, "Portable library development for reconfigurable computing systems: A case study," *Parallel Comput.*, vol. 34, pp. 245–260, May 2008.

[64] D. G. Feitelson, L. Rudolph, U. Schwiegelshohn, K. C. Sevcik, and P. Wong, "Theory and practice in parallel job scheduling," in *Proceedings of the Job Scheduling Strategies for Parallel Processing*, IPPS '97, (London, UK), pp. 1–34, 1997.

[65] I. D. Baev, W. M. Meleis, and A. E. Eichenberger, "Algorithms for total weighted completion time scheduling," in *Proceedings of the tenth annual ACM-SIAM symposium on Discrete algorithms*, SODA '99, (Philadelphia, PA, USA), pp. 852–853, Society for Industrial and Applied Mathematics, 1999.

[66] A. Burns, D. Prasad, A. Bondavalli, F. Di Giandomenico, K. Ramamritham, J. Stankovic, and L. Strigini, "The meaning and role of value in scheduling flexible real-time systems," *J. Syst. Archit.*, vol. 46, pp. 305–325, February 2000.

[67] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood, "Multifacet's general execution-driven multiprocessor simulator (gems) toolset," *SIGARCH Comput. Archit. News*, vol. 33, pp. 92–99, November 2005.

[68] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner, "Simics: A full system simulation platform," *Computer*, vol. 35, pp. 50–58, February 2002.

[69] W. Blanding, P. Willett, and Y. Bar-Shalom, "Ml-pda: Advances and a new multitarget approach," *EURASIP J. Adv. Signal Process*, vol. 2008, January 2008.

[70] R. Pottathuparambil and R. Sass, "A parallel/vectorized double-precision exponential core to accelerate computational science applications," in *Proceeding of the ACM/SIGDA international symposium on Field programmable gate arrays*, FPGA '09, (New York, NY, USA), pp. 285–285, ACM, 2009.

[71] J. Zhou, Y. Dou, Y. Lei, J. Xu, and Y. Dong, "Double precision hybrid-mode floating-point fpga cordic co-processor," in *Proceedings of the 2008*

10th IEEE International Conference on High Performance Computing and Communications, HPCC '08, (Washington, DC, USA), pp. 182–189, IEEE Computer Society, 2008.