Computer Engineering
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ce.et.tudelft.nl/

2013

MSc THESIS

Libswift-PPSPP Information Centric Router: SHA1 Accelerator

Gustavo E. Verdugo Sanchez

CE-MS-2013-13

Abstract

Peer to peer content streaming is the next generation content delivery method, making obsolete the client-server communication model which is not sustainable in the current Internet environment. One effort to standardize this new protocol is represented by the Libswift-PPSPP (Peer to Peer Streaming Peer Protocol) Information Centric Network (ICN). In this thesis we lay the foundation for developing a Libswift ICN router using FPGA technology. We start by describing the protocol, provide a set of requirements for the generic router and identify a subset that we will implement. Using the NetFPGA development platform, we build a hash (SHA1) computation accelerator, which is one of the fundamental building blocks for the Libswift ICN router. Our measurements show that the prototype outperforms a general purpose CPU in hash calculation. The measured speedup is of 1.54 and the area overhead is small considering the complete system. Possible optimizations are discussed and the impact of those optimizations is compared with our working implementation. We conclude with a brief description of further work needed to obtain a complete Libswift-PPSPP ICN Router.

TU Delft

Faculty of Electrical Engineering, Mathematics and Computer Science

Libswift-PPSPP Information Centric Router: SHA1 Accelerator

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Gustavo E. Verdugo Sanchez
born in Tijuana, Mexico

Computer Engineering
Department of Electrical Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

Libswift-PPSPP Information Centric Router: SHA1 Accelerator

by Gustavo E. Verdugo Sanchez

Abstract

Peer to peer content streaming is the next generation content delivery method, making obsolete the client-server communication model which is not sustainable in the current Internet environment. One effort to standardize this new protocol is represented by the Libswift-PPSPP (Peer to Peer Streaming Peer Protocol) Information Centric Network (ICN). In this thesis we lay the foundation for developing a Libswift ICN router using FPGA technology. We start by describing the protocol, provide a set of requirements for the generic router and identify a subset that we will implement. Using the NetFPGA development platform, we build a hash (SHA1) computation accelerator, which is one of the fundamental building blocks for the Libswift ICN router. Our measurements show that the prototype outperforms a general purpose CPU in hash calculation. The measured speedup is of 1.54 and the area overhead is small considering the complete system. Possible optimizations are discussed and the impact of those optimizations is compared with our working implementation. We conclude with a brief description of further work needed to obtain a complete Libswift-PPSPP ICN Router.

Laboratory : Computer Engineering
Codenumber : CE-MS-2013-13

Committee Members :

Advisor:	Koen Bertels
Advisor:	Johan Pouwelse
Member:	Said Hamdioui
Member:	Vlad-Mihai Sima
Member:	Riccardo Petrocco

"Education: the path from cocky ignorance to miserable uncertainty." – Mark Twain

Contents

List of Figures

List of Tables

Acknowledgements

Starting with the faculty, I would like to thank Vlad, Koen, Johan and Riccardo. Without your help, support, guidance, patience, enthusiasm, time and motivation, this thesis would have never been finished. Vlad, your time, patience and help made this happen. The guys at the office, thanks for your friendship, futbolito, woord van de dag, and dutch techno music. Thanks to Said as well, for agreeing to be part of my committee as well as being a good lecturer.

Important gratitude goes to my family. Without their support and seemingly never ending patience, none of my ambitions would have been achievable. Gratitude for my friends in Delft, who have been the best surrogate family, study partners, and bad influence any man could hope for. And special thanks to Delft, for slapping me in the face with reality.

Gustavo E. Verdugo Sanchez
Delft, The Netherlands
August 27, 2013

1 Introduction

1.1 Problem Description

In the last decade, Internet usage has been changing rapidly. With the rise of peer to peer applications, content distribution involves a large number of network resources on the Internet, consequently putting a strain on the current technology. Recently, as video streaming applications have become increasingly popular a further strain has been introduced into the Internet architecture with the increased demand of high quality service for video streaming experience.

The Internet's current client-server communication model of best effort service is very effective at allowing communication between two end points in the network. However, it was never designed for content distribution and is thus inefficient at this task. In fact, high definition video streaming is not possible today if the content is not located close to the end users since data has to travel through multiple networks and bottlenecks to reach its destination. The challenges in distributing content have given rise to companies and technologies whose purpose is to circumvent the limitations of the Internet's current architecture, even though these solutions can be prohibitively expensive or not fully solve the problem.

The Libswift protocol, now known as the Peer-to-Peer Streaming Peer Protocol (PPSPP), has been proposed as a new candidate to solve these limitations. This protocol combines the performance guarantees that a Content Delivery Network provides by pushing content close to the users, along with the efficiencies of P2P technology by sharing the distribution load among consumers. Furthermore, future work on Libswift pretends to rid itself of the underlying client-server communication, and focus on data oriented communication by evolving into a Information Centric Network [32].

Such a protocol poses the problem in that it has different key computational requirements than the standard client-server protocols. The distributed nature of it implies more per packet computations need to be performed in order to check and find content. With this problem in mind, Libswift will be presented in detail in Chapter 2, the core concepts of current technologies will be introduced in Section 1.2 and we will derive the thesis objectives in Section 1.3.

1.2 State of the Art

In this Section, three key technologies used in content delivery will be presented: Content Delivery Networks (CDN), Peer to Peer (P2P) and Information Centric Networking (ICN). The first two of these are widely used in the current Internet, while ICN technology is an emerging research field and has yet to be implemented in a comercial application.

1.2.1 Content Delivery Network (CDN)

A CDN is an overlay network built on top of the Internet, which attempts to solve the problems inherent in traditional client-server Internet connections, by redirecting user requests to the closest data center. A CDN offloads traffic from the original content provider by hosting the same content in its own servers close to users. A CDN has large ammount of server and bandwidth resources, which allows it to dynamically expand or decrease allocated resources to a given content provider. With the rise in popularity of content streaming, CDN providers play a critical in providing a high quality of service, since locally served content avoids costly inter-network connections and avoids the middle mile bottleneck of the Internet.

Akamai is currently the largest CDN provider in the world, as it currently delivers 15 to 20% of all web traffic [42] [27]. The Akamai CDN is built as an overlay network on top of the Internet, running as a software layer on distributed hardware across many locations. When an end user requests a service from a content provider, the Akamai mapping algorithm provides an IP address to forward the user to an Edge Server. This server delivers content close to the users by hosting part of the data offered by the content provider. If a user requests a service or data which is not located on the Edge Server, then the request is forwarded to the Origin Server and a copy is saved by the Edge Server. The Origin Server is owned by the content provider, and hosts the original content which all Edge Servers mirror.

1.2.2 Peer to Peer (P2P)

A Peer to Peer (P2P) Network is a distributed network in which all nodes participate in disseminating data by sharing their own hardware and network resources with one another. P2P is differente from the client-server communication model in which one node with a large number of resources services all other client nodes. [36].

Currently, the most popular P2P application worldwide is BitTorrent [5]. Depending on the region, BitTorrent accounts for between 40 to 70% of all Internet traffic. However, this percentage has dropped in the last years due to P2P traffic losing ground against web file hosting services and content streaming [37]. Although P2P technology is not traditionally considered a CDN, several P2P CDN hybrid networks have been deployed such as Akamai's NetSession Interface [41] and Pando Network's Content Delivery Cloud [24]. However, these CDN implementations have yet to gain the prominence of traditional CDN technology.

1.2.3 Information Centric Networking (ICN)

Information Centric Networking aims to rid the Internet of the client-server network model, and replace it with a client-data model. In the client-server model, computers are identified by IP addresses and these computers communicate on a one to one basis. If a computer requests data, it will connect to the host who possesses the data, that is, the client connects to a particular server, and requests the data. Any additional computers requesting the same data will create separate connections and generate redundant traffic.

This model introduces several inefficiencies which have only been exacerbated with the exponential rise of multimedia content distribution.

ICN attempts to replace this model with a client-data model. Instead of having clients connect to a host in a particular location to request data, a client instead will request data, and routers along the way will forward that request to the closest host with the data. The use of content caching routers, which cache content passing through in local memory, allows the content to move closer to clients as the content becomes more popular. This model thus achieves multicast natively, as routers along the way can respond with the requested data to the clients instead of forwarding the request further up the routing path to the original host [33].

1.3 Thesis Objective

In this thesis, we will begin work on what is to be the Libswift-PPSPP Information Centric Router. This component aims to be the data caching node which will form the basis of the Libswift-PPSPP Information Centric Network. Such a router must fulfill the following requirements:

1. Be able to identify Libswift traffic passing through it
2. Contain enough local memory to cache Libswift data messages
3. Be able to quickly verify the integrity of Libswift data messages
4. Be able to quickly respond to passing Libswift data request with data in router memory
5. The ICN requirements must not significantly impact the throughput of the router.

Item one requires that the router constantly monitor all traffic without delaying it. Item two requires that the router contain local data storage which must be quickly accessible through the router's hardware. Item three entails that the SHA1 Hash of each Libswift data message be calculated and verified through the use of Merkle Hash Trees. Item four requires that the router can quickly determine if the data request can be served from local memory, without significantly delaying the packet.

The design and implementation of the Libswift-PPSPP Information Centric Router is a long term project which is beyond the scope of this thesis. For such large projects, a piecemeal approach becomes necessary. A possible starting point is to begin the design of a router that can identify Libswift traffic, and verify the integrity of data messages (objectives 1 & 3). Te best scenario under which initial development can be quickly implemented, is to integrate these components with the current software version of Libswift.

However, the verification of data requires the use of Merkle Hash Trees (MHT), which are difficult structures to implement in hardware as they require a variable size hash tree to be kept in memory. The verification of data does require the calculation of the SHA1 Hash of all incoming data messages, which provides an opportunity for a simpler

design. Since this data is readily avaible as packets pass through the network interface, a SHA1 Accelerator can be implemented to calculate the SHA1 hash of this data. The SHA1 Accelerator can then provide the computed hash to the Libswift software layer to continue the process of verifying the data message with the use of the MHT.

Conveniently, if the SHA1 Accelerator will be integrated with the current software based Libswift, then it can be assumed that it will behave as an end node, and no routing functions will be needed. Removing the routing functionality, and building the SHA1 Accelerator on top of a Network Interface Card (NIC) removes unneeded complexities without impacting the design or implementation of the Accelerator for the future ICN Router.

In summary, in this thesis we will address the following objectives:

- Identify Libswift packets
- Compute the SHA1 hash of a Libswift data message in hardware
- Append the SHA1 hash to the end of the packet in the form of a hash messages
- Modify the Libswift software to work with the hardware
- Achieve higher efficiency than running only the Libswift software

1.4 Thesis Organization

In this introduction, the problem description, the state of the art, and the thesis objectives have been presented. In Chapter 2, a thorough presentation on Libswift will be given. Chapter 3 will present the experimental platform, the NetFPGA, along with a detail description on it's usage. In Chapter 4, an exploration of the design space will be studied, and a prototype design will be explained. In Chapter 5 the experimental results will be given along with a discussion avaliable opimization options. Lastly, the conclusion of this work will be given in Chapter 6.

Libswift: Peer-to-Peer Streaming Peer Protocol (PPSPP)

2

2.1 Introduction

Formally, Libswift is a content-centric multiparty transport protocol whose purpose is to distribute content among a swarm of peers [29]. It is an efficient peer-to-peer protocol which quickly distributes content such as multimedia, among many nodes in a network. To do this, Libswift indexes content into small bins, and starts distributing each bin among the nodes in the network. These nodes then start sharing the data bins among themselves. In P2P terminology, a node participating in a transfer is called a Peer, while a group of peers in a network participating in the transfer of a particular content are together called a Swarm.

For a peer to join a swarm and begin a transfer, Libswift only needs the identifying hash of the content, which is called the root hash, along with the network address of at least one peer. This is in contrast to BitTorrent which requires the joining peer to download a metadata file called a torrent file, as well as connecting to a Tracker which then directs the joining peer to other peers. The small information that Libswift needs in order to join a swarm allows it to be easily integrated into operating systems and encapsulated into URL for internet browsers, and can work transparently, without any action from the user [29].

Libswift's small footprint and transparent integration into operating systems as well as its efficient transfer capabilities, make it ideal for video on demand and live streaming applications in which it offloads traffic from the original content servers and distributes it to the participating users. This application has already been tested successfully by Wikipedia, a non-proffit website [1]. Such advantages have placed the Libswift protocol as a working group reference for a new draft Peer to Peer Streaming Peer Protocol (PPSPP) by the Internet Engineering Task Force (IEFT) [2].

As an introductory example, we refer to Figure 2.1 which shows the basic functionality of Libswift. At the start, one peer which is known as the original seeder, holds the entire data set. In case of this example, the data is divided into a set of three bins labeled 1, 2 and 3. In the top diagram the leechers, which are peers who want to download the data, connect to the swarm by means of knowing the root hash and the address of the seed. At this point, each leecher requests data from the seed and each receives a data bin as well as the location of the other peers. In the bottom diagram of Figure 2.1 the leechers have connected to each other, and have begun sharing the content that each of them has. Once the leechers have received the complete data set, they become seeders.

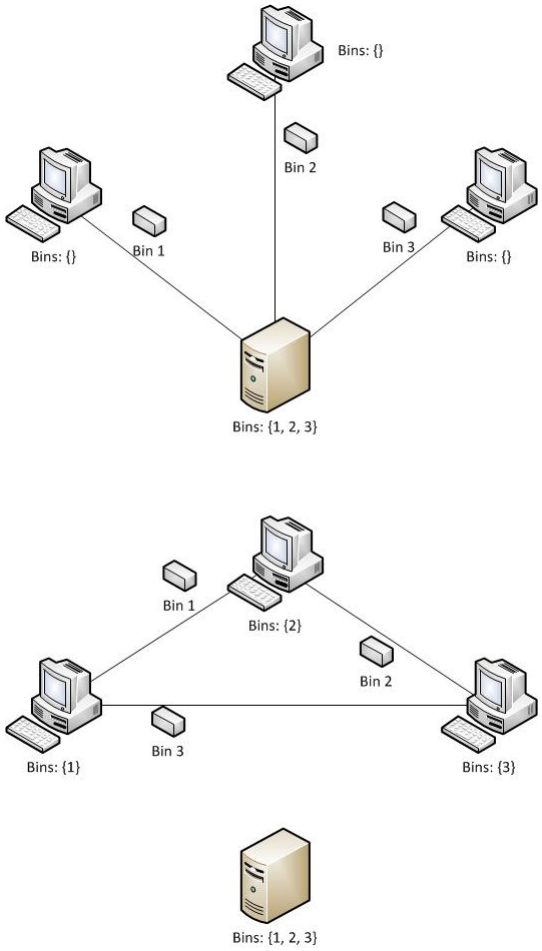

Figure 2.1: P2P Basic Functionality

2.2 Usage and Application

Libswift is designed as a download engine, and thus no user interface for end uses is provided. However, applications which behave as Libswift wrappers bring Libswift to users transparently. In the following sections, several Libswift applications are introduced.

2.2.1 Tribler

Tribler [43] [31] is an open source [28] peer-to-peer file sharing application which encompases several protocols and provides an attractive user GUI. It was originally designed to address several of BitTorrent's shortcomings. Namely it addressed the lack of content and peer search capabilities, video on demand and live streaming, and the lack of incentives for users to continue seeding after they completed a download.

The software organization of Tribler is based on three main sections. The overlay network which connects Tribler users to each other, the protocols which address the

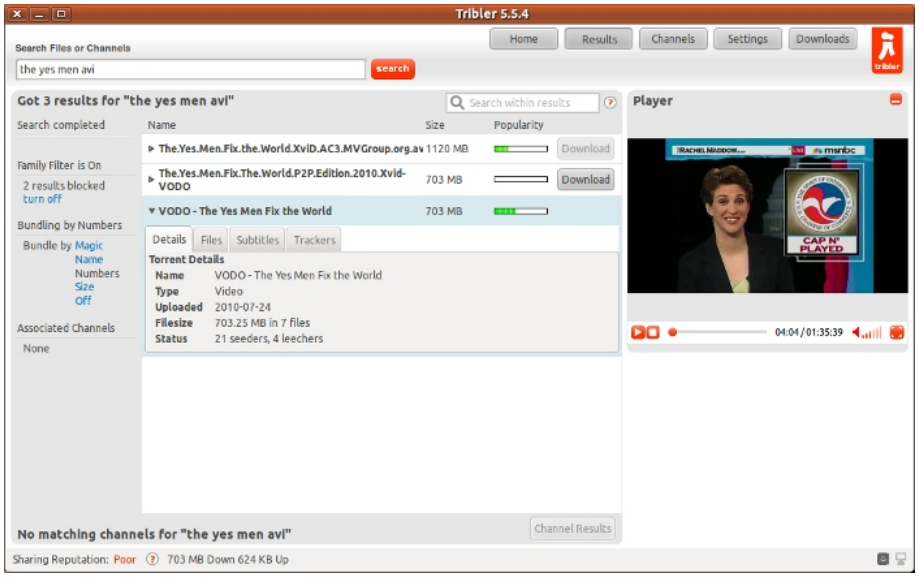

Figure 2.2: Tribler GUI

deficiencies of BitTorrent mentioned above, and most importantly for the work of this thesis, the download engine. Originally, this engine was based solely on BitTorrent which allowed users to connect to swarms to download content. As of version 5.9, Tribler also includes Libswift as its engine. This allows Tribler to join both Libswift and BitTorrent swarms, as well as converting all content received on BitTorrent swarms to Libswift [30].

It is important to understand the difference between Tribler and Libswift. Libswift, is the download engine, which apart from being involved in the actual sharing process including handlying VoD and live streaming, it regulates peer behavior inside a single swarm. Capabilities dealing with the behavior of peers across different swarms, user interaction, and the search of multimedia content lie solely in the domain of Tribler's overlays.

In the original BitTorrent protocol, torrent metadata files need to be obtained outside the protocol itself, and are often hosted on websites. Peer discovery is done by centralized trackers which introduces a single point of failure, though a Distributed Hash Table Network (DHT) extension proposal, as well as peer exchanges, avoids this issue [17]. These deficiencies are addressed in Tribler by use of the overlay network and several protocols. The use of the BuddyCast protocol connects peers with each other and thus enables peer discovery, and the MegaCache protocol which acts as a distributed database of torrent files which can then be searched by user queries.

To address Video-on-Demand (VoD) Tribler uses a modified download policy. The BitTorrent download policy chooses chunks to download based on rarest first, while Tribler instead has a balance between high priority pieces which are required for playback, and rarest pieces for the health of the swarm. Clients use Give-to-Get algorithm to pick which peers to share data with. It encourages peers to share with others who have proven to be good peers by uploading to the swarm [22]. Live streaming poses the

unique problem that no complete metadata file exists before hand since the video is generated live, thus peers have no hashes to verify data. To circumvent this, Triblers uses a combination of public/private keys to authenticate the seeder and thus trust the data received. It is very important to note that these are the modifications that Tribler has in order to enable these features in BitTorrent swarms. Since the Libswift protocol has these features implemented natively, all Tribler users participating in Libswift swarms have these capabilities enabled unlike their BitTorrent counterparts.

Lastly, BitTorrent does not have an incentive mechanism to reward peers who participate in a swarm after they have completely downloaded the content. The protocol however, does reward and punish leechers who do not upload in the domain of a single swarm using the tit for tat scheme, but possitive participation by seeders is not rewarded accross swarms. The tit for tat scheme encourages peers to send data to other peers who send data back to them. Private BitTorrent comunities based around private trackers have gotten around this limitation by tracking the upload download ratio of registered members. Tribler on the other hand, has implemented a reputation overlay named BarterCast which keeps track of how much data has been exchanged between peers. Using data recorded by BarterCast, each peer can estimate how much each peer has uploaded accross all swarms and prioritize upload to peers with a good reputation.

2.2.2 SwarmPlayer 3000

SwarmPlayer [26] is a browser extension which enables users to watch streaming video directly in a browser, while simultaneously downloading it from a swarm. As video streaming usage has soared, the bandwidth costs of maintaining such services online has risen with it. SwarmPlayer mitigates these costs by offloading traffic from the content provider to the viewers, and enabling users to share the content among themselves. This technology is already being tested in the wild with the cooperation of Wikimedia by enablying all video streams on Wikipedia for the use of SwarmPlayer [1]. Originally based around the BitTorrent engine, the latest version of SwarmPlayer, dubbed SwarmPlayer 3000 [43][25], has been updated to use the Libswift engine.

In Figure 2.3 we can observe how SwarmPlayer works. Upon encountering an HTML5 video tag, a Java script checks if the SwarmPlayer is installed in the browser. If it is not, then the video is served from the original host servers via HTTP. If it is installed, then the root hash or torrent file is requested from url2torrent.net server. With the required metadata received from the server, the respective BitTorrent or Libswift swarm engine is initialized and the peer joins the swarm by connecting to other peers. After receiving enough pieces to fill the required playback buffer, the video starts while the peer continues the download as well as uploading the content to other peers.

2.2.3 Libswift as an Information Centric Network (ICN) Protocol

The disadvantage of the leading ICN projects, such as Named Data Networking (NDN), is that they demand a complete replacement of the Internet architecture and infrastructure in one big leap. This deployment strategy requires a great deal of upfront investment, with the economic advantage arriving at some undetermined time in the future. Libswift possesses many of the features that are required for an ICN implementation of the

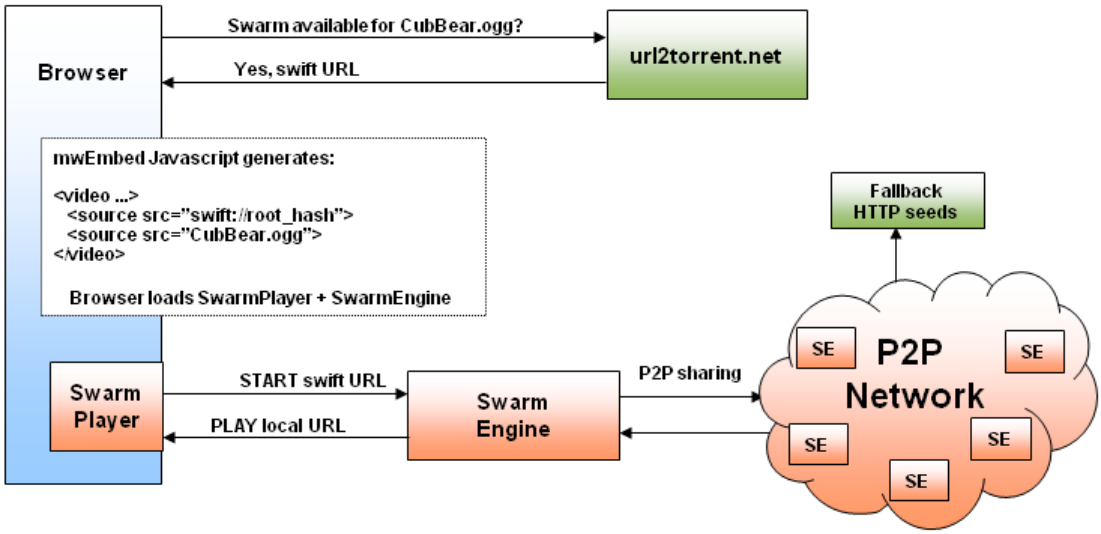

Figure 2.3: SwarmPlayer Functionality [25]

Internet. Since Libswift works over existing Internet infrastructure, the Internet can evolve step by step into an information centric architecture instead all at once [32].

The basic ICN requirements are fulfilled by Libswift using semi-unique hashes to identify content instead of identifying content by name and location. This allows Libswift to find content by means of the identifying hash regardless of where it is stored in the network. Furthermore, unlike BitTorrent, which requires the pre-transmission of all metadata to the joining peer, Libswift verifies data by means of a Merkel Hash Tree (MHT) [20] in which only a limited number of hashes need to be transmitted. Note that the root hash of the MHT is the hash used to identify content. This couples naming and verification of conent together. These characteristics, in theory allows data to be cached midway transmission by intermediate caching routers which would then serve data requests thus achieving the ICN objective. This system would thus push content and the inherintly attached hash necessary for verification closer to the users [12].

2.3 Protocol Overview

Following is a description of the essential components of Libswift. All information on the inner workings of Libswift is contained in the Peer-to-Peer Streaming Peer Protocol (PPSPP) draft [2] unless otherwise cited.

2.3.1 Data Organization - Binmaps

In order to transmit files of arbitrary size, data must be partitioned and organized in an efficient way. The BitTorrent protocol splits data into a particular number of chunks, and uses a Bitmap, which is basically an array of bits, to label a block of data as downloaded

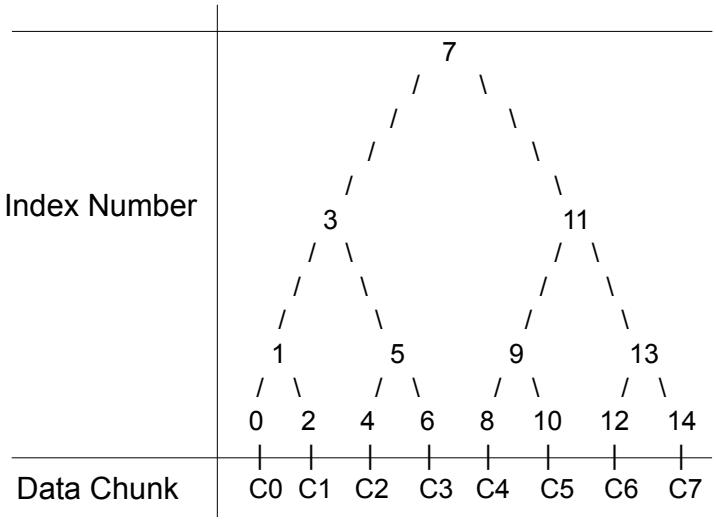

Figure 2.4: Libswift Binary Tree

or not downloaded. Peers then exchange these bitmaps to inform each other of the availability of chunks [34].

Libswift favors the use of Binmaps [13] which are a hybrid of binary trees and bitmaps, to achieve the goal of organizing data. A small binmap example is shown in Figure 2.4 in which each node, which we refer as bins, are given a decimal index number. Data is indexed into 1 KB chunks, unless otherwise specified, and are labelled C0 to Cx. These chunks are then mapped to the leaves of the binmap, so chunk C0 maps to Bin0, chunk C1 maps to Bin2, and the rest of the chunks are mapped in the same manner. If the amount of data in a file(s) is not large enough to fill a whole binmap, the rest of the binmap is assumed to be filled with zeros in order to fill the tree. The binmap is then built on top of the leaves.

Using this type of structure to organize data has important advantages. First, knowing that an inner node is built on top of other nodes, a leecher can request a continuous group of chunks of data by referring to a higher level node which encompases all of the chunks. For example, requesting data for bin 3 translates into making a request for data of bin 0, 2, 4 and 6. This simple property of binmaps contributes to reducing the communication overhead between peers. To communicate to a peer that the entire file is available, a Libswift peer has to only indicate that it has the root bin. This is in sharp contrast to BitTorrent which has to send the entire bitmap.

On a hardware perspective, the decimal index used in the tree translates neatly into a binary index. Figure 2.5 shows the translated version of the tree in Figure 2.4. Looking at the index numbers, we can note that the first zero bit from right to left indicates the level of the node. The following bits to the left of the first zero indicate the node number in the tree from left to right. For example, node 0110 indicates a level zero node, in the forth position. Node 1011 indicates the second node at level two.

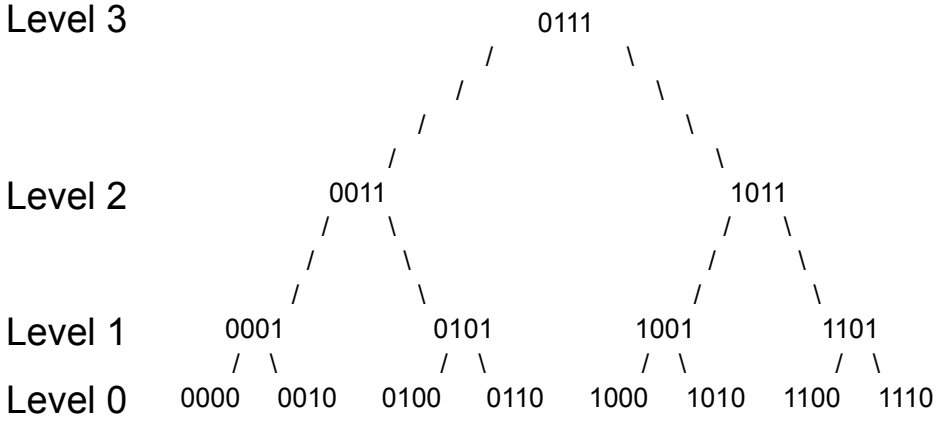

Figure 2.5: Libswift Binary Tree - Binary Indexing

2.3.2 Data Verification - Hashes

Data verification is an essential aspect of transmitting data for which Libswift provides two methods. Merkle Hash Trees (MHT) [21] are the recommended method, which is used both for downloading static content as well as live streams. Libswift can also use public keys for live streams, the details of which is not be covered in this work.

Libswift calculates the SHA-1 hash [15] of each individual chunk of data. These hashes form the leaves of the MHT which use the same indexing system as the binmaps. If no data is present to complete the base of tree, zeros are appended to the data prior to calculating the hash in order to complete the base. The inner nodes of the MHT are formed by concatenating the hashes of the children, and calculating the hash of that concatenation. This method is shown in the example of Figure 2.6 in which a small three level MHT is created. Libswift proceeds calculating hashes until the root hash of the tree is reached. This Root Hash thus becomes the identifier of the swarm and is at the core of data verification in Libswift.

In contrast to Libswift, BitTorrent stores the SHA-1 hashes of all the chunks of data in the torrent file. Large files being shared cause the size of the torrent file to become large due to the number of hashes contained. This can be mitigated by increasing the chunk size, but this causes the side effect of each peer having to wait to complete downloading large chunks before they can verify and share them. However, work on extending BitTorrent to the use of MHTs has been done in [3].

An advantage to using MHTs is that Libswift doesn't need the hashes of all the chunks in order to verify data was received correctly. Take as an example bin 8 shown in Figure 2.7. If we are to verify the data integrity for that particular bin, we will need the hash of the sibling, and the uncles in order to be able to compare the produced hash against the root hash. An uncle hash is the sibling of the parent, and recursively the sibling of that node's parent. So in our example we will need the hash for bin 10, 13, and 3. Since these hashes are stored in the MHT, the more data we verify the more complete the MHT becomes, and thus reduces the need to request hashes from other peers. For example, if the next request in our example is for bin 12, then Libswift no longer needs

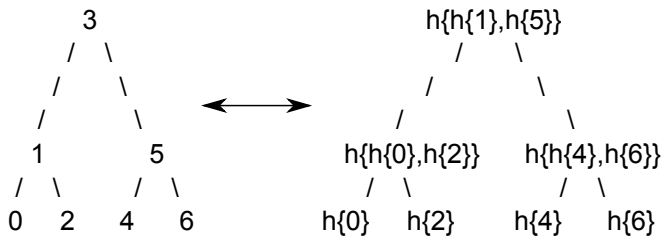

Figure 2.6: Creating a Merkle Hash Tree

to request another hash since bin 14 is empty [2].

Peak hashes are a separate concept not to be confused with uncle hashes, and play an important role in Libswift by enablying peers to calculate the file size, and allows Libswift to treat static downloads and live streaming almost equally. To define a peak hash, we must understand what filled and incomplete nodes are. A filled node is a node or leaf whose interval consists of only non-empty leaves. Leaves that are partially filled with data are also considered filled nodes. Contrary, an incomplete node is one whose interval consists of filled and or empty leaves. Thus, a peak hash is a node whose right sibling is an incomplete node. In Figure 2.7, the peak hashes are 12, 9 and 3 [2]. Note that the root node can be used to know how many peak hashes are in a tree. Take the binary representation of the index of the root node, and every 1 in the index represents a peak hash. So for our example, root bin is 7 which is binary "111" and we do in fact have 3 peak hashes.

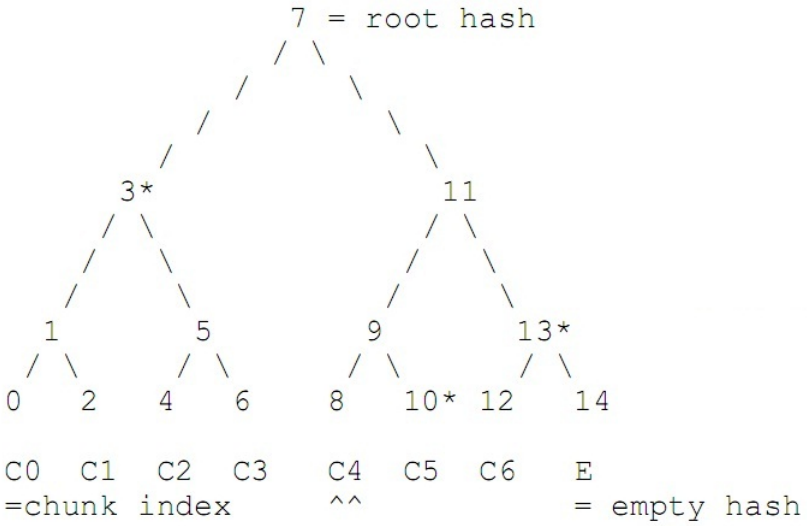

Figure 2.7: Merkle Hash Tree (MHT) [2]

To calculate the size of the file without needing to explicitly receive this information Libswift uses peak hashes. Knowing that all leaves containing data are covered by peak hashes, the remaining leaves of an MHT must be empty bins. Thus, the root hash of the tree can be checked using only the peak hashes and the assumed empty hashes. In the example of Figure 2.7, the peak hashes are bin 12, 9 and 3. Knowing that bin 14 is empty, we can use the peak hashes to calculate the concatenated hash up to the root of the tree, and compare it to the root hash we already know is correct (the swarm ID). Having verified which bin is the last one containing data, Libswift has deduced the file size of the download [2]. Thus, when a peer sends the first data message to a new peer that just joined the swarm, the sending peer must also send all the peak hashes so the new peer can calculate the download size.

The second important use is that peak hashes allow us to treat downloads and live streaming almost equally. The differences are two. First, the MHT is constantly growing, so the root hash changes as the tree expands. The second difference, is that not all hashes are transmitted and only root and peak hashes are cryptographically signed by the sender. When the MHT expands, the previous root node becomes a peak node, and the sender signs the new root node as well as the new peak hashes. This method increases the delay an application experiences in receiving data from the transport layer because the signature must be verified before data can be accepted. However, this method reduces the amount of data that must be signed by the sender [2].

2.3.3 Messaging

Libswift adheres to using Atomic Datagrams for all communication which means that every datagram sent must be independent of any other datagram. Hence a message should not be partitioned into multiple datagrams and it is for this reason that the default chunk size of Libswift is 1 KB since the Ethernet Maximum Transmission Unit (MTU) size is 1500 bytes. When transmitting data, hashes must be put into the same message as the actual data. If for some reason they do not fit, hashes must be sent before data is sent. If Libswift receives data for which it does not have the hashes, Libswift will drop the data message.

Table 2.1 contains the list of all messages available in Libswift. The message type number is also included in the table which is useful for identifying messages both in hardware as in software. In this work, we will only focus on the most essential messages for Libswift functionality. The most essential messages in the protocol follows as well as their payload descriptions.

- HANDSHAKE message initializes the channel between two peers. Shown in Table 2.2 and Table 2.3.
- DATA message contains a complete data bin. Shown in Table 2.4
- ITEGRITY messages are used to transmit hashes between peers. Shown in Table 2.5.
- REQUEST message is a request for data. Shown in Table 2.6.

Message Type	Description
0	HANDSHAKE
1	DATA
2	ACK
3	HAVE
4	INTEGRITY
5	PEX_RESv4
6	PEX_REQ
7	SIGNED_INTEGRITY
8	REQUEST
9	CANCEL
10	CHOKE
11	UNCHOKE
12	PEX_RESv6
13	PEX_REScert
14-254	Unassigned
255	Reserved

Table 2.1: Complete Message List [2]

- ACK message acknowledges the sender that data was received correctly. Shown in Table 2.7.
- HAVE message announces to other peers which bins are available. Shown in Table 2.8.
- PEX messages are used to exchange peer information.

Message type headers are one byte long. In the following figures we present brief examples of how messages are structured inside a packet. Note that all Libswift datagrams begin with the channel number which is used to identify swarms. These channels allow different swarms to use the same UDP port. Detailed view on bit ordering of packets, along with Ethernet, IP and UDP headers will be shown in Section 3.3.2 after our development platform has been introduced in Chapter 3.

To understand how messaging works, let us look at a very simple transfer between two peers as shown in figure Figure 2.8. On the left side is the message sequence, and on the right side we have a small tree that represents data consisting of only four bins. To start a connection, the leecher sends a HANDSHAKE message. The seeder then responds with a complementary HANDSHAKE along with a HAVE message. The HAVE message in this case contains the root node, which is number 3. Upon receiving this message, the leecher sends a REQUEST message with bin number 0 in it. The seeder then responds with a DATA message containing bin 0, along with the necessary uncle hashes that the leecher needs to verify the data. In this case, the uncle hashes for bin 0 are 2 and 5. After verifying the data, the leecher sends and ACK message along with a REQUEST message for bin 2. In this case, the seeder can send the DATA message without accompanying

To Unknown Chnl	Msg Type	From Chnl	Version	RootHash	Option	EOM
00000000	HANDSHAKE	00000011	v=01	si=xxxx	ca=0	end

Table 2.2: Initial Handshake Message

To Chnl	Msg Type	From Chnl	Version	Option	EOM
00000011	HANDSHAKE	00000022	v=01	xxx	end

Table 2.3: Response Handshake Message

Msg Type	Bin Range		Time Stamp	Data Payload
DATA	00000010	00000010	12345678	BA5EBA11...B16B00B5

Table 2.4: Data Bin 2 Message

Msg Type	Bin Number	Hash
INTEGRITY	00000111	1234ABCD...DCBA4321

Table 2.5: Integrity Message for Bin 7

Msg Type	Bin Number
REQUEST	00000111

Table 2.6: Request Bin 7

Msg Type	Bin Number	64-bit Timestamp
ACK	00000111	9837923

Table 2.7: Ack Bin 7

Msg Type	Bin Range	
HAVE	00000003	00000003

Table 2.8: Have Bin 3 Message

INTEGRITY messages since the leecher already has the necessary hashes. Afterwards, the seeder sends a REQUEST message for bin 5, which itself consists of bins 4 and 6. After the seeder delivers these requested bins, the leecher acknowledges receipt and the data transfer is now complete.

It is important to note that the above messaging example is relevant even when adding more peers into the swarm. In this case, every peer joins a particular swarm

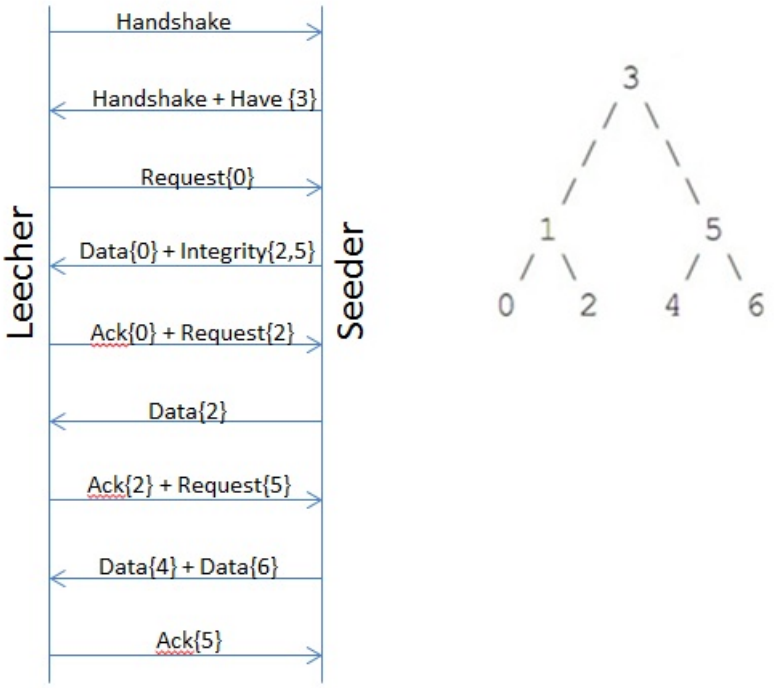

Figure 2.8: Libswift Basic Message Communication

channel, and messaging works in the same way as above except for the order of the data been transmitted. Additionally, the PEX message is used to exchange peer information with the swarm so that new connections to other peers can be established.

2.4 Performance Analysis - Comparison with other Protocols

Performance analysis on Libswift has been done in [34]. This study focuses on comparing Libswift to other protocols in specific scenarios, though it does not consider identifying which aspects of the protocol are a bottleneck. However, insightful information is provided which leads to possible applications for hardware acceleration, though this particular study did not consider these ideas. NextShareTV shows that Libswift already reaches 100% CPU usage on high download rates. Dedicated Libswift hardware could offload the CPU to achieve greater download speeds. Libswift integrated into an Android application compared power consumption compared to the YouTube application in which it showed Libswift using marginally more power than YouTube. Again, it is possible that dedicated Libswift hardware can be advantageous by lowering power consumption on this mobile device. Comparing VoD algorithms, having Libswift hardware acceleration would allow the protocol to verify data faster thus contributing bandwidth at higher rates network bandwidth is not a limiter.

Another study did offer some insight into the inner workings of Libswift. [35] finds that chunk size has a significant impact on Libswift performance. Experiments were done on 1KB, 8KB and 16KB sizes, and found that 16KB was the best performing chunk size, while 1KB fared the worst. Metrics considered were total download time, as well as total CPU time and memory usage for both seeders and leechers. All of these metrics improved using larger chunk sizes. Howerver as attractive as these benefits are, using a chunk size larger than 1KB destroys the principle of using atomic datagrams which Libswift is based on. The use of non atomic datagrams hinders the future goal of building a Libswift ICN. The reasons of why this performance impact exists is not given, but one can suspect that while the SHA-1 hashing algorithm scales with input size, using smaller chunks increases the size of the trees and the number of packets that Libswift has to process, thus greatly increasing overhead.

2.5 Profiling

In order to better understand the inner workings of an algorithm, it is necessary to have the tools to identify bottlenecks and slow performing functions within the source code. Profiling is the method to obtain such information. Within this work, we shall focus on using Gprof which is a tool wich gathers information concerning execution time within each function, as well as information on a function's ancestors and decendants.

2.5.1 Testing Platform

The simple testing platform consists of two workstations directly connected by a 1 Gbit ethernet cable. No intermediary networking components are present. The workstation specifications are as follows.

Server worksation:

- Intel(R) Core(TM)2 Duo CPU E6850 @ 3.00GHz
- 2 GB RAM
- 4 port ethernet controller: Intel Corporation 82576 Gigabit Network Connection

Client NetFPGA workstation:

- Intel(R) Core(TM)2 Duo CPU E8400 @ 3.00GHz
- 2 GB RAM
- NetFPGA with 4 dual-port 1 GBit NIC implementation.

2.5.2 Gprof

Gprof [11] is a profiling tool used to measure the running time of any called function in a program as well as to visualize a function call hierarchy. In order to achieve this, gprof inserts profiling code into the source code of the program on compile time. Specifically, it inserts this code at the points where functions are called. As expected, this introduces

some processing overhead which interferes with the timing meassurments. Interference is more severe when a function takes little time and is called a large number of times. After a program is compiled for profiling, it is executed in order for gprof to collect the necessary run time data.

Gprof measures the time a function takes to complete by samplying the value of the program counter at regular time intervals and infers the execution time by analyzing the distribution of all the samples taken during the program execution. Hence, the time reported by gprof is a statistical approximation.

Data is presented in two ways. The Flat Profile presents all the functions that were called during the execution of the program, along with the number of times they were called as well as the number of seconds they took to execute. The functions are ordered by longest execution time to lowest. The Call Graph Profile lists the functions along with the parent functions as well as the children. Execution time is indicated along with the execution time for the children. This makes it clear how much time is spent on a function by itself, and how much time is attributed to its children. Both these profiles present the data collected in textual form. In order to make the information easy to analyze, the tool Gprof2Dot [9] was used in order to make a visual representation of the Call Graph Profile shown in Figure 2.10.

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 45.95      1.36     1.36  2513130     0.00     0.00  blk_SHA1_Block(blk_SHA_CTX*, unsigned int const*)
 16.72      1.86     0.50  2842030     0.00     0.00  swift::binmap_t::set(bin_t const&)
 10.81      2.18     0.32  1409596     0.00     0.00  swift::binmap_t::is_empty(bin_t const&) const
  4.39      2.31     0.13   292716     0.00     0.00  swift::binmap_t::_find_complement(bin_t const&, u
  3.38      2.41     0.10   279426     0.00     0.00  swift::binmap_t::cover(bin_t const&) const
  2.70      2.49     0.08   479581     0.00     0.00  swift::MmapHashTree::OfferHash(bin_t, swift::Sha1
  1.69      2.54     0.05   344685     0.00     0.00  swift::binmap_t::is_filled(bin_t const&) const
  1.69      2.59     0.05     9098     0.01     0.01  VodPiecePicker::Pick(swift::binmap_t&, unsigned l
  1.18      2.62     0.04  3008217     0.00     0.00  bin_t::layer() const
  1.01      2.65     0.03   977622     0.00     0.00  blk_SHA1_Update(blk_SHA_CTX*, void const*, unsign
  0.68      2.67     0.02   279365     0.00     0.00  swift::MovingAverageSpeed::GetSpeedNeutral()
  0.68      2.69     0.02   139925     0.00     0.00  swift::Channel::Reschedule()
  0.68      2.71     0.02   139715     0.00     0.00  swift::Channel::AddHave(evbuffer*)
  0.68      2.73     0.02   139630     0.00     0.00  swift::Channel::Send()
```

Figure 2.9: Gprof Flat Profile

Two instances of Libswift are executed in order to profile with Gprof. An unmodified version on the seeder, and a Libswift version compiled with the -pg compiling parameter, and default optimization using the gcc compiler. Libswift is launched from the command line, with the seeder being launched first in order to allow it to complete the initial Libswift operations such as building the MHT to obtain the root hash of the data. Afterwards, the leecher is launched to download the file and disconnect from the swarm immediately after the download is complete.

By running Gprof to profile Libswift during the leeching of a 136MB file we obtain the flat profile resuts which are shown in Figure 2.9. We can observe that the function blk_SHA1_Block(...) takes up almost 46% of the execution time. This function, along with with the other functions containing SHA1 as part of their name, form the group of functions responsible for calculating the SHA-1 hash. These functions are shown in

Figure 2.10. Note that the time meassurements in these figures vary slightly due to them having been obtained during different test runs.

From this last figure, we can observe that there are two cases when the SHA-1 hash needs to be calculated. Upon reception of a data package, the SHA-1 hash of that data which forms the leaves of the hash tree is calculated. Afterwards, the data needs to be verified against the root hash or a hash that has already been verified. For this, the nodes above the leaves of the hash tree need to be obtained by using by calculating the SHA-1 hash of the concatenation of the two children of that node. As we can see, upon reception of a Hash package which is handled by the Channel::OnHash function, the SHA-1 is also used. This is because Libswift must also verify the received hash against the root hash. Since this also requires the calculation of the SHA-1 hash of the nodes above the leaves, both this operation, and the SHA-1 hash above the leaves for a data package are handled by the MmapHashTree::OfferHash function.

Upon careful inspection of the Libswift source code, it is found that to calculate the hash of a leaf when receiving a data package, the function SHA1(...) is called by the MmapHashTree::OfferData function which is only access by the reception of data event. Having identified the SHA1(...) function as calculating the leaf hash, and observing the graph entry for that function, we can see that it takes 20.35% of the time to calculate the SHA-1 hash of a leaf. Also refering to the same graph, we know that Libswift spends almost 46% of the time calculating hashes, we can deduce that about 26% of the time is spent calculating the hash of the inner nodes. This number can also be seen by the call to Sha1Hash::Sha1Hash(...) function.

It is important to note that measurments were performed during multiple Libswift transfers as well as with file sizes varying from 130MB to 700MB. Actual graphs shown in this section are for a 135MB file transfer. In general, we can estimate that Libswift spends between 40 and 50% of the time calculating hashes. The time calculating the hashes of only the leafs varies little and stays between 18 and 22%.

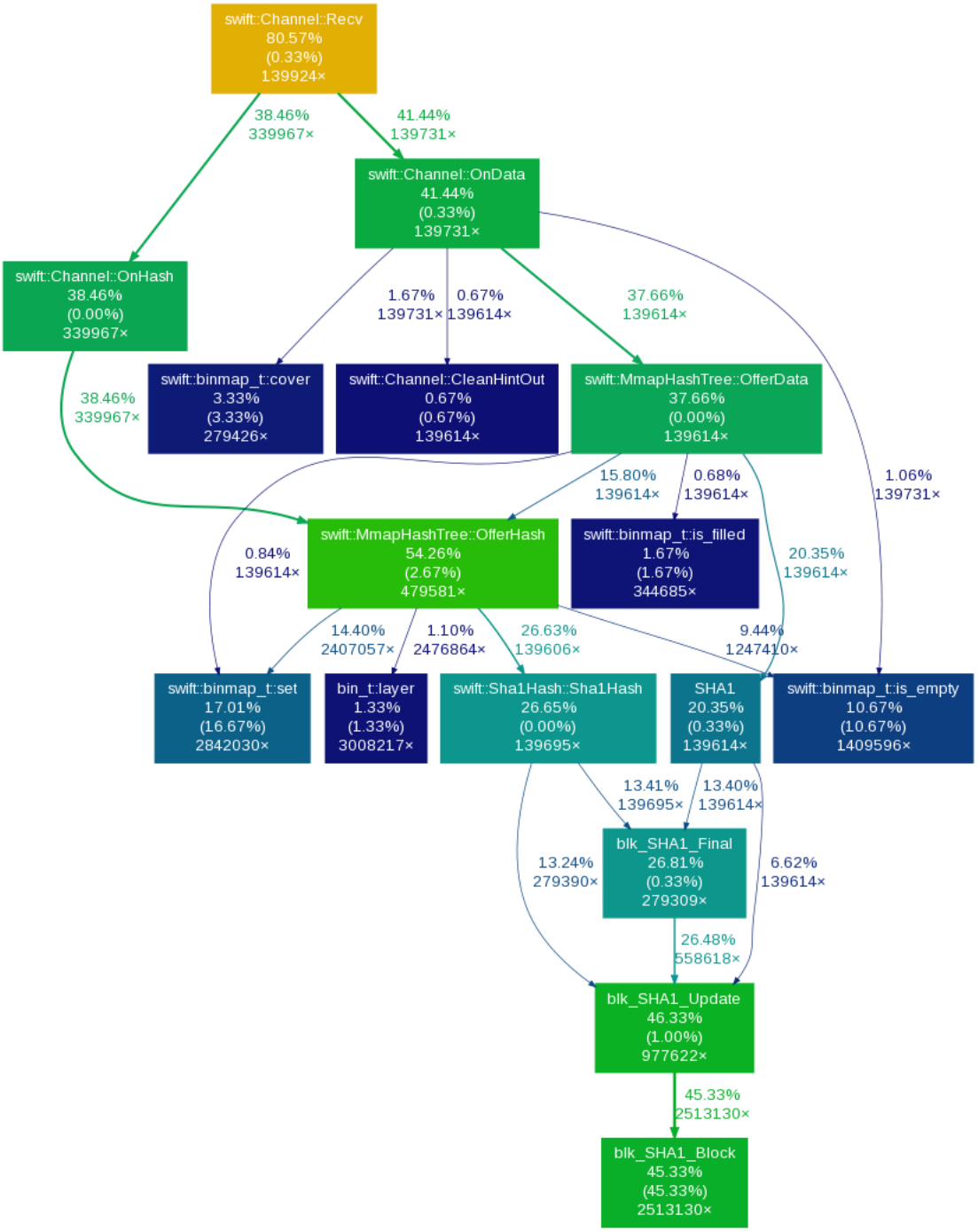

Figure 2.10: Gprof Call Graph

3 Experimental Platform

In this chapter, we introduce the NetFPGA platform which we have chosen as our experimental platform. In Section 3.1 we give a brief introduction of the platform with a short listing of its components as well as its capabilities. Section 3.2 has an overview of the NetFPGA Package (NFP), which is intended to familiarize the reader with the platform in order to understand our implementation. Section 3.3 provides detailed installation instructions, a custom module example, information on designing new tests, as well as a list of problems and errors we encountered and their respective solutions. This chapter is intended to give a new user of the platform guidance in order to improve the learning curve of using the NetFPGA.

3.1 Introduction

NetFPGA [10] is an open source hardware and software platform built for teaching, research and development of networking components. The NetFPGA is organized as an expansion card for the PCI port of a standard workstation or mount rack server. The hardware provides 4 1-Gbit ethernet ports which are connected to a Virtex II FPGA in which the User Data Path (UDP) is implemented. This allows for line rate data processing. The NetFPGA contains both SRAM and DRAM memory, although in the reference design the SRAM is occupied by the packet buffers and no hardware driver is provided to interface with the DRAM. The hardware component of the NetFPGA is responsible for the core data processing of networking applications, while control functions are implemented on the host computer. Communication with the host computer is achieved through the PCI port, which not only provides a data path for network packets, but also provides an interface for the host system to access the hardware registers in the NetFPGA.

The NetFPGA Package (NFP) includes the NetFPGA hardware, the software drivers, as well as project tools. It also comes included with several reference projects. These projects can be used to form the base of most new applications and thus have been extensively tested to ensure proper functionality. The included reference projects are as follows:

- Hardware IPv4 Reference Router provides 4-port full duplex 1Gbit interfaces
- Hardware 4-port full duplex 1Gbit NIC
- Buffer Monitoring System which adds hardware to monitor input/output buffer usage
- Software Router Kit copies the Linux routing table and ARP cache to the NetFPGA

- Software Component of NetFPGA (SCONE) provides more functionality than the Router Kit, such as a user GUI and write access to the NetFPGA registers.

The NFP also comes with several non-reference extra modules that can be implemented into a project to expand its functionality, such as Packet Delay modules and Buffer Queues implemented in DRAM memory. Further functionality is provided by the NetFPGA community in which research groups share their open source designs by following the reusable hardware methodology described in [7] which focuses on a test-driven design process. Howerver, these designs are not as thoroughly tested as the reference projects.

This development platform was chosen for this project due to various advantages. First, it is an inexpensive platform specifically designed for experimental networking applications, that enable us to build a working Libswift application. Second, the standard reference designs and user contributed designs remove the need to develop our own basic network modules. Lastly, testing tools have already been developed which allow us to simulate and verify our design under realistic scenarios. However, the NetFPGA is not without its drawbacks. A severe lack of documentation, which is based mostly on a user contributed wiki page, implies a steep learning curve to new users, greatly increasing a project's startup time for innexperienced groups. The Flex Router project [18] took note of this deficiency and tried to mold their project as an introduction to the NetFPGA.

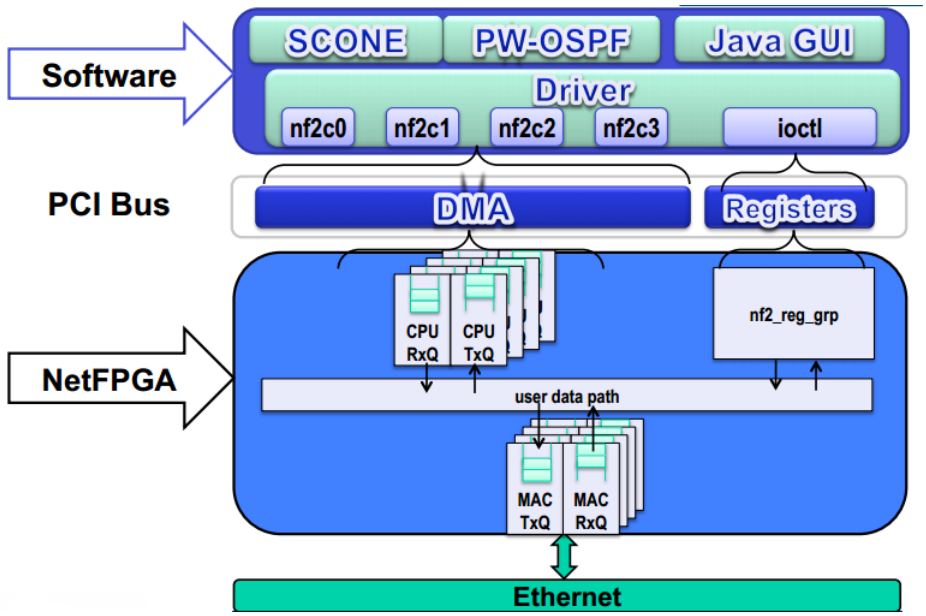

Figure 3.1: NetFPGA System Overview [14]

The NFP contains three major components [7] [6] which are the kernel module, the software package, and the reference pipeline which will be described in more detail in the following section. In Figure 3.1 we can observe the overview of the NetFPGA system in which the three subsystems can be observed. The Software layer is clearely labelled,

though the kernel module sits overlapping the software layer and PCI Bus connections. The Reference Pipeline is labelled as "NetFPGA". In it, we can observe the packets queues, the register interface, as well as the User Data Path which is where project modifications usually take place.

The Input/Output Queues serve both MAC and CPU requests. Input ports coming into the datapath are labelled as RxQ, and output ports TxQ. MAC ports are connected directly to the Ethernet, while the CPU ports are connected to the DMA PCI port. The nf2-reg-grp module is the top level module for NetFPGA register access by the CPU.

3.2 NetFPGA Package (NFP)

In this section, the NFP will be explained with special attention to the relevant components for the development of our project.

3.2.1 Development Software

The Development Software included in the NFP greatly facilitates the implementation of new designs. It includes Applications which interact with, or monitor a current design running on the NetFPGA. Includes the verification and build systems which allows for testing, building, and access to the NetFPGA components. Importantly, it includes all the source files of the reference projects and tools. In this section, these development components will be introduced.

3.2.1.1 Applications

Applications provide the top level interface with users and the control functions for the NetFPGA. Included in the NFP, SCONE provides many services such as IPv4 forwarding, ARP and ICMP messages and provides graphical user interfaces. The Java interface provides the user with real time information on NetFPGA buffer usage, registers, and network traffic as well as access to control functions.

3.2.1.2 Verification System

The Verification System consists of a set of python and perl scripts and tools which have the capacity to interact both with logic simulators, as well as with implemented hardware. This built-in verification system provides the capacity to design tests specific for networking purposes. Following the Test Driven Design approach presented in [7], providing comprehensive tests is necessary to contribute a custom project to the NetFPGA community. Do note that regression tests are referenced in many pages of the online documentation, but have been obsolete since the NFP version 3.0. This system is covered in detailed in Section 3.2.4.

3.2.1.3 Build System

The NFP Build System is responsible for linking the dependencies of a project and organizing the synthesis of a project in order to build the final bit file. The build system

includes the Register System which allocates physical memory to hardware modules and automatically creates the register definition files for supported languages. The synthesis system automatically includes modules documented in the XML file and checks the bit file for timing errors after synthesis. The synthesis system also includes all user Xilix Coregen .ngc and .xco modules located in the synth folder.

3.2.1.4 Utilities

The NFP includes various tools are used to interact with the various components of the NetFPGA. The tools can be used directly in the command line, but are most frequently used inside the Python script files. Among the most important tools are the following:

- nf_download : is used to download bitfiles to the NetFPGA. Automatic error reporting is provided to aid in debugging the NetFPGA installation.
- nf_test : used for running simulations and hardware tests.
- regread & regwrite : used for the reading and writing of registers on the NetFPGA memory.
- nf_info : provides basic current information about the NetFPGA.
- sram_dump : provices read access to the NetFPGA SRAM.

3.2.1.5 File Tree Structure of NetFPGA Package

The NFP contains all the reference projects, library modules, tools and systems that form the NetFPGA. Projects developed by users must follow a predefined structure to fit into the toolchain. Following is a description of the NFP file structure. Note that the root folder of the NFP will know be referred by its environment variable $NF_ROOT.

```
netfpga ($NF_ROOT)
├── bin :  perl and python scrips for simulations and environmental
│   setup
├── bitfiles :  synthesized bitfiles from projects
├── lib
│   ├── C : Common software tools for all designs; Includes kernel
│   │   drivers, register access tools, and testing tools
│   ├── java :  Graphical user interface written in Java
│   ├── Makefiles :  Makefiles used for synthesis and simulation
│   ├── Perl5 :  Perl scripts to manage simulations, create test data,
│   │   and interact with loaded designs on the FPGA
│   ├── python :  libraries used in regression tests
│   ├── scripts :  various tools used for development and testing
│   ├── verilog :  hardware source code for reference modules
│   └── xml :  xml schemas
└── projects :  all projects including user projects and included
    reference designs
```

Following is the project folders structure. Do note that not all folders are necessary though, at the minimum, the include and synth folder are required. From now on, the project's root folder will be known by its environment variable $NF_DESIGN_DIR. The description of the content in each folder follows:

```
projects ($NF_DESIGN_DIR)
├── doc :  documentation folder
├── include :  XML file which lists the project definition and included
│   library modules, as well as the verilog register definitions
├── lib :  register definitions for C, Perl and Python
├── src :  custom Verilog libraries for this particular project;
│   Modules in this directory overrides shared (library) modules with
│   the same name; Verilog files must not be more than one folder deep
│   in order to be detected by the build system
├── sw :  custom software for this project
├── synth :  synthesis files such as the project Makefile and Xilinx
│   core gen .xco files
└── test :  Hardware test and software simulation files
```

3.2.2 Linux Kernel Driver

The driver kernel, whose source is contained in the C files folder of the NFP tree described in Section 3.2.1.5, provides the main interface between the Software and Hardware layer of the NetFPGA system. Packets are handled using the standard Linux networking stack and are transfered to the Hardware layer through the DMA interface by the PCI bus. Registers in the NetFPGA are accessed through *ioctl* system calls which travel through the *nf2-reg-grp* module in the NetFPGA. Top level functions used to access registers are *readReg* and *writeReg*. The register system provides an alternative to transfer packets without using the DMA interface, albeit achieving slower performance.
A typical NetFPGA to Host System packet transfer through the PCI bus goes as follows:

1. Packet arrives at CPU TxQ.
2. An interrupt notifies the driver of a waiting packet.
3. DMA transfer is initialized by the driver.
4. Packet is transfered by DMA.
5. An interrupt notifies the driver transfer is complete.
6. The driver pushes the packet up the network stack.

The Software/Hardware communication which the kernel provides is not only used for packet and register access during the execution of a particular hardware application, but it also gives access for the developer to download bitfiles to the Virtex II FPGA. Only under exceptional circumstances would a project require a modification to the Kernel Driver. Our project did not modify it so we will not go into further detail.

3.2.3 Hardware Component: User Data Path (UDP)

3.2.3.1 Overview

The Hardware Component of the NFP encompases the UDP, the input/output queues, as well as other system components such as *nf_top* and *nf_core*.
Most of the custom modifications will be performed on the UDP. For the reference router design implementation as well as for the reference NIC, we can see in Figure 3.2 what components the UDP encompases. The UDP is implemented inside the Virtex II FPGA and consists of the Input Arbiter, the Output Port Lookup as well as the Output Queues. However, the MAC and CPU input/output queues are not described within the UDP. The Input Arbiter services the MAC Receive Queues as well as the CPU Receive Queues in a round robin fashion. The Output Port Lookup decides which output port the packet should be sent through, as well as performs several operations on the packet such as Time To Live (TTL) update and CRC check. The Output Port Lookup is where the Reference NIC and Reference Router differ. The NIC has a simple routing design, while the Router has to query the forwarding tables and modify the packet contents. The Output Queues module simply holds the packets until the Transfer Queues (TxQ) are ready to serve another packet.

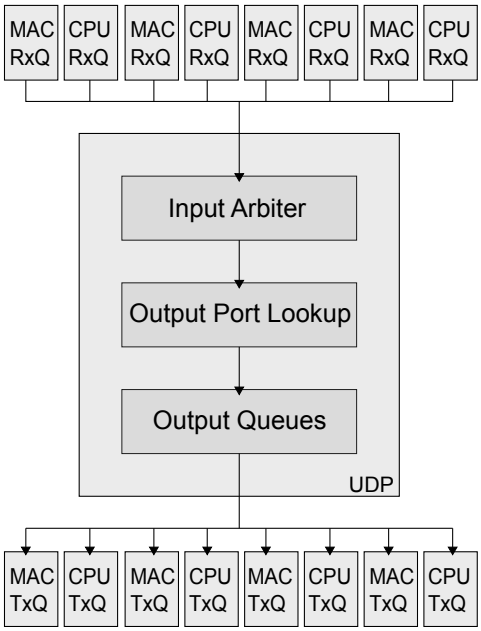

Figure 3.2: NetFPGA User Data Path

3.2.3.2 Data Pipeline

The data pipeline is 64 bits wide and runs at 125Mhz resulting in a total bandwidth of 8 Gbps. Figure 3.3 shows a block diagram of a generic module in the UDP along with its signals and buses for data communication with other modules. Do notice the name

discrepancy between the in/out name of the RDY signal and the actual direction or the signal. All “in“ signals are connected to the preceeding module while all “out“ signals are connected to the next module regardless of signal direction. Packets are transfered between modules in a first-in first-out manner on a word by word basis. Every module in the UDP, as well as the standard empty module provided as reference, contain a small FIFO which receives all the input words from the preceeding module. This FIFO is not defined within the NFP but is a standard core from Xilinx.

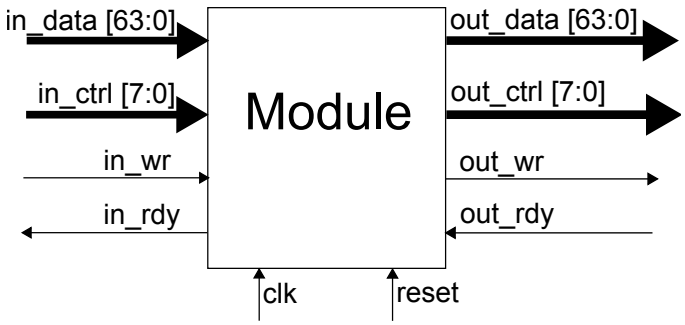

Figure 3.3: Inter-module Packet Communication

For a module to receive the next word in the data path, it must raise the RDY signal. The previous module in the pipeline reads RDY signal high, and proceeds to put the next word in the FIFO in the data bus, along with the corresponding CTRL 8-bit word, and sets the WR signal high once the data is ready in the bus. The receiving module must be ready to accept the data word one clock cycle after it has raised the RDY signal. Each clock cycle in which the RDY signal remains high signals the previous module that more data is requested. Thus if the receiving module cannot receive more data, it should lower the RDY signal one clock cycle before it cannot receive data.

BITS	63:48	47:32	31:16	15:0
FUNCTION	OutPort	TotalPkgWords	InPort	TotalPkgBytes

Table 3.1: IOQ Header

Each module is allowed to prepend a Module Header word 64bits wide to each packet. These module headers are used to pass control information between modules. In the reference designs, the only module header used is the IOQ header which is built by all the Rx queues upon packet arrival and destroyed by the Tx queues. The header which is shown in table 3.1 is structured as follows [23]. Bits 15:0 signal the length of the packet in bytes. Bits 31:16 contain the binary enconded input port. Bits 47:32 signal the packet length in terms of 64-bit words. Bits 63:48 are modified by the Output Port Lookup module and contain the one-hot encoded output port through which a packet must be sent.
The CTRL bus is used to pass control signals to the proceeding module on a word per word basis. It has mostly two functions. The first is to signal the end of a packet by

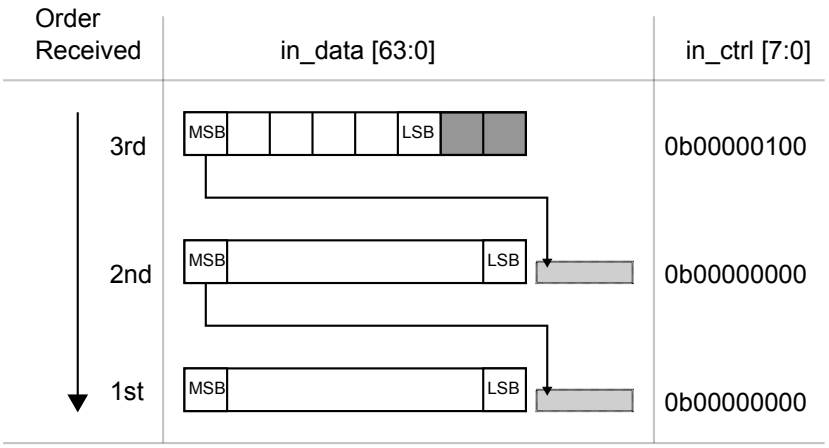

Figure 3.4: Data Order in Inter-module Communication and Last Byte Control Signal

indicating which byte of the data word is the last byte of the packet. During the transfer of non module header words, the CTRL signal is set to 8h00. This can be observed in Figure 3.4. In this figure we can also observe the byte ordering word per word. The second function of the CTRL bus is to differentiate module headers from each other. The IOQ module header is identified by the CTRL signal 8hFF which is defined in the verilog global definition IO_QUEUE_STAGE_NUM. The IOQ control code also serves as an indicator to the begining of a new packet. Any new custom module headers can be assigned any CTRL signal with the exception of 8hFF and any word with only one high bit as these are reserved for the end of the packet.

3.2.3.3 Registers

To implement host accessible registers in the UDP, the register pipeline is used. The register pipeline is 32 bits wide and runs at 125 MHz. Each module in the pipeline must have the input/output signals shown in Figure 3.5. A module must forward all signals from the inputs to the outputs unless it detects that a request is made for a register in that module. Register requests are signaled by *reg_req* high for one clock cycle, while multiple requests are done by keeping the signal high in consecutive clocks. The *req_ack* signal is used to respond to requests and must only be driven high by the responding module. *reg_rd_wr_L* signals a register request read with high, and write with low. *reg_addr* identifies the target register for wich each module is given a section of the memory address space. *reg_data* is the 32 bit data bus. *reg_src* is a two bit bus that identifies the source of the register request.
Modules are chained together in a ring where data flows one way. The advantage of a ring design over a star design is that new modules can be inserted in the ring without having to modify a central arbiter. The ring is connected in the UDP as shown in Figure 3.6. The *udp_reg_master* module interfaces the register pipeline in the UDP to the *nf2_core* module which in turn interfaces with the PCI bus in order to communicate with the host computer.

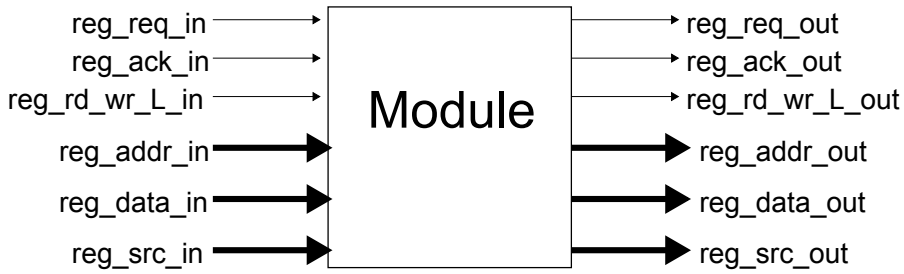

Figure 3.5: Inter-module Register Pipeline Communication

Using the Register System script *nf-register-gen.pl* which has now been integrated into the python build system, the required register files are created. These files include the Verilog register file which defines the name and width of the registers, as well as the header files in C, Perl and Python to access the registers. Each module within a project that uses registers must have its own XML file with register definitions.

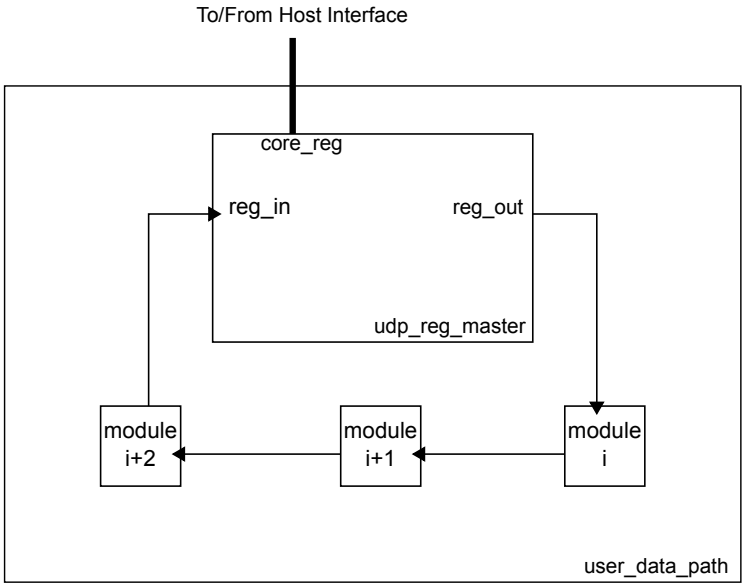

Figure 3.6: Register Pipeline

3.2.4 Verification of Custom Modules

3.2.4.1 Test

For the developer, the main interface to specific tests is the *run.py* script in $NF_DESIGN_DIR/test/"test_name"/run.py. These scripts must be placed alone, in a folder with the name interface_major_minor where interface can be sim/hw/both meaning that the test can be run as a simulation, hardware test or both. Major and minor are the

name of the test and both are required. The run.py scrip cannot be called directly, but must be accessed by the use of the *nf_test.py* tool located in $NF_ROOT/bin/nf_test.py. To run the simulations, ModelSim 6.3 is the officially supported version. In our project, we used version 6.3d. To launch a simulation, *nf_test.py* must be run with the parameters "interface –major name –minor name" and with the optional –gui parameter to launch the test waveform in ModelSim. The location of the waveform signals for the UDP inside ModelSim are located in nf_top/nf_core/user_data_path/.
Running tests in hardware requires the creation of a test that verifies the packets received with the packets expected as described in the test file. Physical connections must be made from the NetFPGA to a target client, and specific packets to send through the interfaces must be specified. Packets can be observed passing through the interfaces with Wireshark [40], though this method gives a point of view from the Operating System perspective and any modifications to the packets made inside the NetFPGA cannot be observed.

3.2.4.2 Scapy

Scapy [4] is a Python tool which is used to build and dissect, as well as to send and receive network packets. Since Scapy is used both to send and receive packets, it can compare packets and is thus used for verification purposes within the NFP tests. Scapy is Python based, thus it is seamlessly integrated in the NFP Python verification system and it is extensively called by the *run.py* scripts. Following is a brief example on how to use Scapy to build packets in order to learn how to build custom NFP tests. Do note that Scapy is not included in the NFP and must be installed individually. If using CentOS5 instead of Fedora14 which the second supported OS by the NFP, Python compatibility issues can be encountered.
Scapy uses a layering system in order to build packets. For LibLibswift, an ASCII payload will be appended to a UDP header, appended to an IP header which in turn will be appended to an Ethernet header. Scapy provides convenient function calls with default values included, which frees the developer of having to define all fields in each header. In order to append a layer, the key "/" is used. Following are basic commands to build, view and send a packet.

```
>>> x = Ether(dst="90:E2:BA:1F:A9:60")/IP(dst="192.168.11.21")
/ICMP()/"HelloWorld"
>>> x.pdfdump("/some/folder/output.pdf", layer_shift=1")
>>> x.show()
#Console output with x contents.
>>> sendp(x)
.
Sent 1 packets.
>>> "ba53ba11".decode("hex")
'\xbaS\xba\x11'
>>>
```

In the above example, a packet "x" was built with an Ethernet header, specifying the destination interface. We appended an IP header with the destination IP. The default

ICMP header is an echo-request message. We also appended a packet payload with the ASCII payload “HelloWorld“. This built packet can be printed in console with the command x.show() or printed to PDF with the command x.pdfdump for which a similar output is shown in Figure 3.7. The function sendp() was used to send the packet from layer 2 since the MAC addresses had already been specified. Without building the Ether() header, using the function send() is necessary in order to enable routing table lookup. If one desires to manually insert a hexadecimal or binary value into the payload of the packet, the payload must be converted to ASCII with the command ”text“.decode(”hex“). Sending packets through the wire can then be observed using Wireshark and appear in the hex format as shown in Figure 3.7.

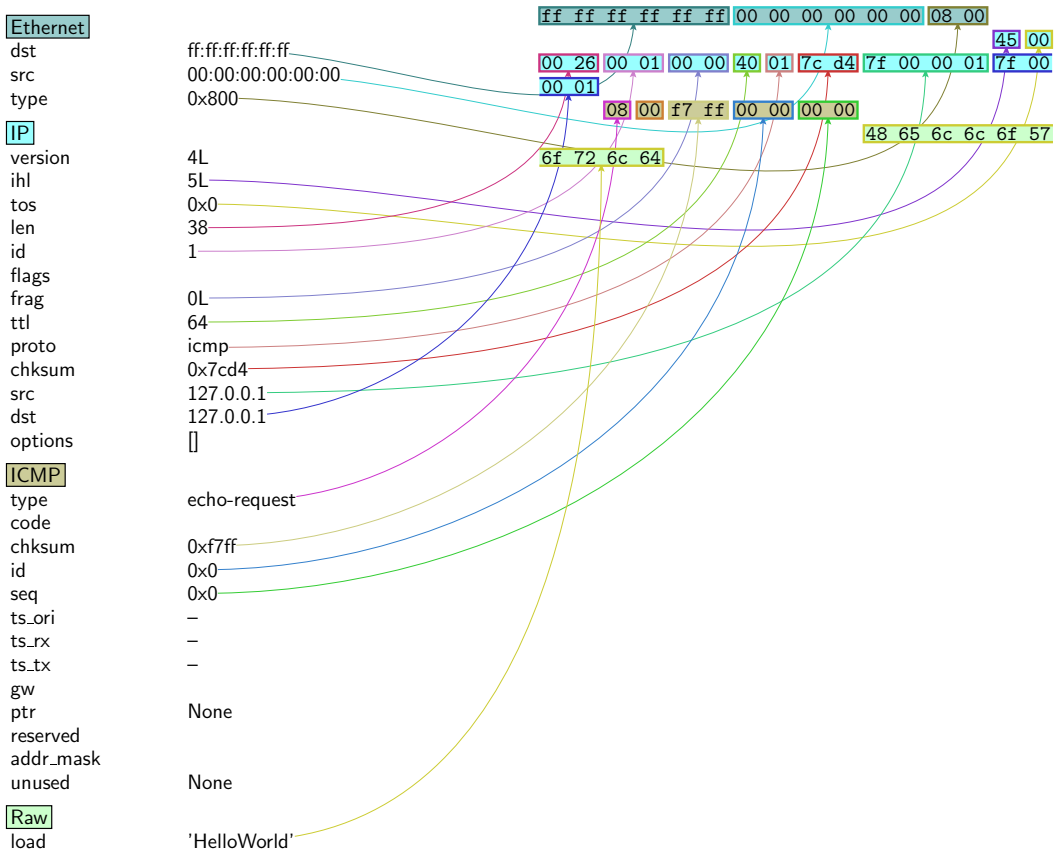

Figure 3.7: Scapy Packet

3.3 NFP Installation and Usage

3.3.1 Installation

Installation of the NFP can be a daunting task if errors are encountered for which no documentation exists. From our experience installing the NFP on CentOS5, we would

recommend that future users use Fedora14 instead as it the only supported OS in future versions of the NFP. For an explanation and solution to the errors we encountered, refer to Section 3.3.4.

1. Install NFP by either using *yum* or by manually downloading and unpacking a tar file from the NetFPGA wiki page and follow the driver installation instructions.

2. The on board CPCI system needs to be reprogrammed every time the NetFPGA host is restarted. The CPCI is the DMA interface between the Host PC and the NetFPGA and is hosted on the two on-board Spartan3 FPGAs. To reprogram the CPCI automatically on reboot add the following line to the file in /etc/rc.local "/usr/local/netfpga/lib/scripts/cpci_reprogram/cpci_reprogram.pl –all".

3. Download a bit file and test. The NFP comes included with bit files for the reference projects. Download the reference_nic.bit located in $NF_ROOT/bitfiles/ using the *nf_download* tool located in $NF_ROOT/bin/. Assign an IP address to an interface, connect the corresponding Ethernet port to another computer and send a ping. Only 1Gbit links are supported by the NetFPGA and a cross over cable is not necessary.

4. Install Xilinx ISE 10.1 and ModelSim 6.3

5. Install Scapy

6. Program the envirnoment variables NF_ROOT and NF_DESIGN_DIR in the *bashrc* file.

7. Attempt to Synthesize the reference_nic project by setting the NF_DESIGN_DIR variable to point to this project, and running make in the $NF_DESIGN_DIR/synth folder.

8. Attempt to simulate the reference_nic project using the nf_test tool introduced in Section 3.2.4.1.

3.3.2 Modifying the UDP

New projects in the NetFPGA can take as a basis one of the reference projects which provides a throroughly tested base from which to begin. This not only makes it easier to debug a new design, but it allows the developer to focus on the project goals instead of on building an entire network system from scratch. The following steps give a brief overview of the process needed to implement a new module in the NetFPGA. In our proof of concept project, we designed and implemented a module that changes the "source" field in an IP header. The module was placed in between the Outport Port Lookup and the Output Queues module in the UDP. Since our design was a simple proof of concept, support for ARP messages was not implemented. Furthermore, the register pipeline was not connected to our module as the use of registers was not required.
The necessary steps to create a new module are as follows:

1. Copy the reference NIC project in $NF_ROOT/project/reference_nic into a new project folder in $NF_ROOT/project/. Rename the folder to name the new project and update the environment variable NF_DESIGN_DIR in the *bashrc* file to point to the new project.

2. If NFP library modules are needed, update lib_modules.txt in the $NF_DESIGN_DIR/include directory to indicate which and how many modules are needed. Furthermore, indicate where in the NetFPGA the module should be placed by updating the project.xml file.

3. If registers are to be used, the module xml file must be updated to allocate memory space for them.

4. To make a new module, it is recommended to use the module template $NF_ROOT/lib/verilog/core/module_template/src/module_template.v as a base which already contains the FIFO interface, and place in the $NF_DESIGN_DIR/src directory. Note that the build system will automatically detect verilog source files for custom modules in the $NF_DESIGN_DIR/src directory up to two folders deep.

5. Copy the reference user data path $NF_ROOT/ lib/ verilog/ core/ user_data_path/ reference_user_data_path/ src/ user_data_path.v and place in $NF_DESIGN_DIR/src/udp/ folder. This file will be your project's new UDP since the build system prioritizes a file with the same name in a project folder over a library module.

6. Modify the project's UDP by instantiating the newly copied custom module which is currently empty, and connecting it to the data path. If using registers, these must be connected as well.

7. Implement the new custom module to include the desired functionality. Our proof of concept consisted of a state machine which activates at the data path control signal 8h'FF and counts up to the 5th word to find and modify the IPsrc field in the IP header of the packet.

8. Run a simulation from the reference projectes or run a new personalized test in order to ensure correctness. Information on how to design a new test can be found in Section 3.2.4.1 and Section 3.3.3.

9. Run make in synth folder which calls the build system. Note that only Xilinx ISE 10.1 is supported and working correctly.

10. Download the resulting bitfile to the NetFPGA by using the *nf_download*.

11. Hardware tests can be performed at this point using the Verification system if the test was specifically designed for this purpose. Alternatively, packets can be sent by submitting the NetFPGA to a real-usage scenario or by sending custom packets built with Scapy. Incoming packets can be seen using Wireshark.

3.3.3 Building a New Test for LibLibswift

Becoming familiar with *run.py* is necessary in order to build a new test. *run.py* uses many custom functions which are defined in various files inside the NFP. The function *make_IP_pkt()* is defined in $NF_ROOT/lib/python/NFTest/PacketLib.py and it is here were Scapy is used to build packets. If we are to custom build packets by defining specific LibLibswift packet functions, it is here where modifications should take place.
For the purposes of this thesis, building custom LibLibswift packets by hand would be useful for creating comprehensive automatic tests. However, it is much simpler to build a new test which can read a Packet Capture (PCAP) file and send those packets through the desired interface. To do this, first a LibLibswift file transfer is captured in a PCAP file using Wireshark from the perspective of the NetFPGA as a leecher. As a basis for our test, we take any *run.py* file and modify it to use Scapy directly in this file to read the PCAP file which should be named and placed as "$NF_DESIGN_DIR/test/pcap/current_pcap_test.pcap". The simulation that was built was specifically made for observing packet waveforms in ModelSim. No automatic packet correctness checks were implemented in the test.
However, there are limitations to using this simplified approach. First, we are limited in which packets we can test by the packets we can force LibLibswift to produce. Building LibLibswift packets by hand allows us to be very specific on the payload content. Second, in the *run.py* script, the function *nftest_send_dma* is used to send all the packets present in the PCAP file. This function places a packet in a specified CPURxQ as if it was originating from the CPU. This does not change the content of the packet (Ether, IP, UDP, Payload), but it does have an impact on the "InPort" field of the IOQ header of the packet which is used to easily identify the source of the packet. The function *nftest_send_phy* can be used to put packets in the MACRxQ, so simple modifications have to be made to the test scrip in order to accomodate this. For the purpose of this thesis, we are only focused on modifying incoming packets. A simplified test that reads a PCAP with selected packets and send them through the MACRxQ port, with a few simple hand made packets sent through the CPURxQ to verify them passing unmodified, is all that is required.

3.3.4 Errors

During the first execution of Synthesis and Simulation, the following errors where encountered. A brief description and the solutions are listed.

1. CPCI version is incompatible error: Reprograming the CPCI must be done every time the NetFPGA host PC is restarted. If error persists, download a new NFP package and reinstall the NetFPGA drivers from scratch.

2. Scapy incompatible with CentOS5 Python version. Updating to a new Python will cause erratic behavior with the test scripts. Make sure a "Python" call in terminal calls a Python compatible with Scapy. Modify the header of each *run.py* test to specifically call python2.7. All other files in the system should remain with their original headers. The error is due to CentOS5 no longer being supported by NFP

3.0 and above. If Python claims that the Scapy package is not installed, use the *pip* tool in Python to search and install the Scapy package for Python.

3. TeamCity Python library not found during simulations. In our particular system, setting the following environment variable fixed the problem: PYTHONPATH=$NF_ROOT/lib/python.

4. Xilinx COREgen problem. Upon the first simulation launch, the Build System will run in order to call Xilinx Coregen to generate the necessary cores. In our first test, Xilinx encountered a license error with the Tri-Mode Ethernet Media Access Controller (TEMAC) core. The required license is only needed for the first build of the core, and only for simulation purposes and not for synthesis. Downloading the evaluation license for all packages which contain the core and update the LM_LICENSE_FILE environment variable to append the path to the location of the license file fixes the problem.

Exploration and Design - Proof of Principle

4

In this chapter, the proof of principle of the Libswift SHA-1 Accelerator will be presented. Problem exploration, design solution, and implementation will be discussed. Section 4.1 establishes a standard Libswift development branch for hardware support. In Section 4.2 Hardware System Design, the exploration process will be explained in which the system requirements where established, and a design solution presented. Section 4.3 explains the verification system which was required to be implemented before work on the Libswift acceleration could begin. Sections 4.4, 4.5 and 4.6 thoroughly explain the individual components of the Libswift SHA1 acceleration. Section 4.6 explains in detail the SHA-1 algorithm, and discusses problems that were encountered and the solution that was implemented.

4.1 Libswift Build for NetFPGA Support

Due to ongoing development in the Libswift protocol, it is of paramount importance that development of hardware Libswift applications be done for a specific version of the protocol. Libswift development can result in discrepancies in Libswift messages which can render hardware implementations inoperable.
Hardware development originally started on Libswift build 32414 with the option ENABLE_IETF_PPSP_VERSION in the file *swift.h* enabled. This option forces Libswift to behave according to the PPSPP draft version 06 [2]. Leaving this option disabled results in the usage of a different bin indexing system which is not recognized by our hardware. Build version 32414 was used on our profiling analysis in Section 2.4.
Later in development, a new Libswift branch was created based on build version 32414, in order to make the necessary modifications for SHA-1 hardware acceleration. The hardware running on the NetFPGA will append a new hash message with header number 14, followed by the 160-bit SHA-1 hash result to the end of every packet containing a Libswift data message. The new Libswift branch has been modified to look for this extra message at the end of a packet, at the moment it needs to calculate the hash of a newly received data message. As a fail safe, if no hash message is found, the software will proceed to calculate the hash as normal.
There are two important reasons for which the Libswift hardware appends the new hash message at the end of a packet. A design aspect of Libswift is that a data message must be the last message present in a packet. Inserting a new hash message in the middle of a packet involves partitioning the packet and then reasemblying it with new content in the middle. Given that our hardware needs to access the data message content at the end of the packet, extra memory would be needed to store the packet while this content of the end of the packet is accessed. Thus, appending the hash message at the end of the packet reduces the need for control logic and extra memory. Further, by having the

hash message appended after a data message, a minimally modified Libswift protocol can run on top of the NetFPGA without using hardware acceleration.

4.2 Hardware System Design

To accelerate the SHA-1 calculation in Libswift using the NetFPGA, we identify the following requirements:

- Identify a Libswift data message as a packet moves through the data path.
- Provide memory to hold every packet while searching for a Libswift data message.
- If a Libswift data message is present, modify the IOQ packet header with updated packet size information. If no data is present, allow the packet to fall through.
- Calculate the SHA-1 hash from the data message payload.
- Append the resulting hash using the new Libswift message type.

The location of the SHA-1 hardware acceleration with respect to the User Data Path (UDP) can be observed in Figure 4.1. This location was selected for several reasons. First, it is inside the unified path through which all packets, regardless of IO port must pass though. This allows for one module to monitor all traffic going through the NetFPGA. Second, placing our module after the Output Port Lookup allows us to bypass Ethernet and IP checksums which are performed in this module. Modifying a packet before the Output Port Lookup without recalculating the checksums will result in the packet being dropped. Third, placing it after the Output Queues module is inpractical since this module connects directly to all Output Buffers which would cause us to either need very large multiplexed inputs, or to have multiple SHA-1 hardware accelerators. It is important to mention that the later choice can increase performance of our acceleration if multiple Libswift streams are used going to different NetFPGA output buffers, with the expense of a larger FPGA area used.
The block diagram of the final design which meets the stated requirementes is shown in Figure 4.2 and consists or five main components. The Message Parser module scans all incoming packets in search of a Libswift data message. It checks the packet headers and parses through the Libsiwft payload until a data message header is found. The Packet Modification System (PMS) updates the packet headers, activates the SHA Modules, and appends the SHA-1 hash result message to the end of the packet. This module also has memory to hold an entire packet while the Message Parser makes a final decision. The memory is shown as a separate module named Packet Buffer. The SHA modules consit of a group of three modules whose purpose is to take the payload of the Libswift data message and calculate the SHA-1 hash. The SHA Interface module connects to the NFP datapath and aligns the data to be fed into the SHA-1 algorithm. The SHA Wrapper module implements control and data delivery for the SHA Core. The SHA Core calculates the SHA-1 hash of a single 512-bit block of data and depends on the SHA Wrapper for all control functions. All components will be explained in detail in the rest of this chapter.

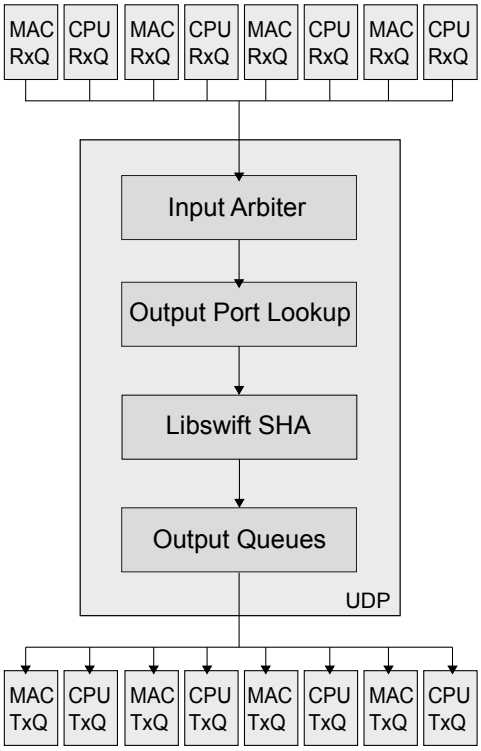

Figure 4.1: Modified NFP Datapath for SHA-1 Acceleration

4.3 Testbench

Thorough testing of any implementation is crucial when building a system of this complexity. Testing can be done before synthesis in the form of simulations, or after synthesis in the form of real hardware tests. However, both forms of testing are required for proper design verification. For pre-synthesis testing, a carefully planned testbench has to be designed which must target the complete system or a subset of it.
For our project, we can group all modules in two categories. Those modules which are directly connected to the UDP datapath, and those that are not. The modules which are not connected to the datapath have specialized ports and functions which require individual unit testbenches. The other modules are connected to the UDP datapath and are designed to handle data in the form of packets. The NFP has a built in testbench which can test the UDP using ModelSim, with packets generated through Python simulation scrips. These scripts can be modified in order to hand build Libswift messages using Scapy. However, unless significant effort to automate this process is invested, hand building packets becomes a very time consuming process. A much more time efficient method is to modify these simulation scrips to read packet PCAP files and insert that data into the simulator.

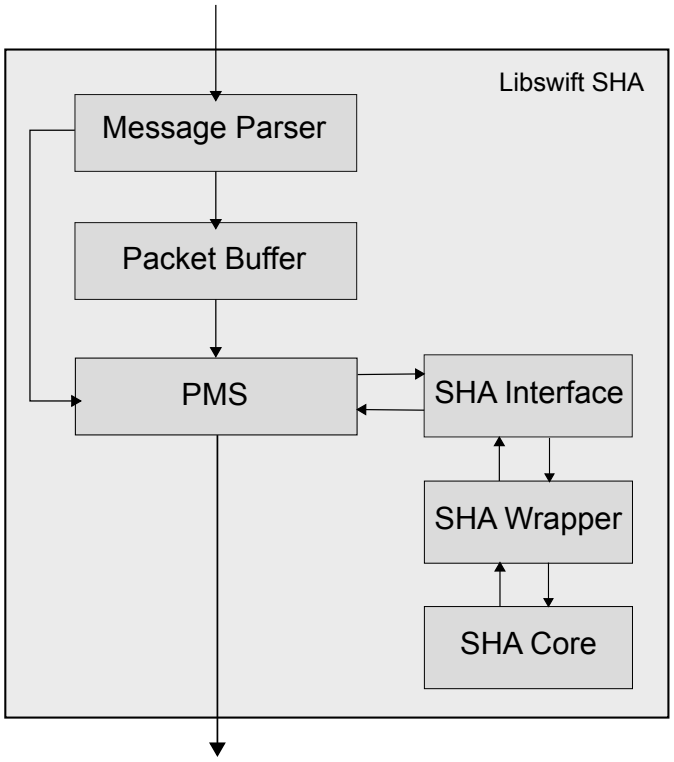

Figure 4.2: Diagram of the SHA-1 Acceleration

4.3.1 PCAP Testbench

Using the PCAP Testbench allows for easy capturing of packets and using them as an input to the testbench. By using PCAP files, we can use data that is used in the actual running of Libswift transfers. However, the ease in which we can capture packets for the testbench, also means that control is lost over the fine tuning those packets. If very specific packets are needed for testing, then creating packets by hand using Scapy is an option.
It is difficult to get packets of various sizes and types in order to make a completely thorough testbench in which all possibilities are tested. However, a reasonably complete testbench can be created and used effectively, if the limitations of the test are known when implementing hardware. Special care must be taken when writing Verilog code which is known not to be tested in the current testbench. Note must be taken of these sections of code in order to facilitate system debug after implementation.
In order to have a testbench with an accurate IOQ header, our original testbench from Section 3.3.3 had to be modified to send packets both from the DMA interce as well as the MAC interface. This requirement is necessary because the port through which the packet enters the NetFPGA determines the values of the IOQ header. Our expanded testbench checks the Ethernet destination address of every packet. If a match is made with the local MAC address of the NetFPGA, the packet is classified as an incoming

packet and is pushed into the MAC interface. Otherwise, it is classified as an outgoing packet, and it is placed in the DMA interface to be sent out though the physical ethernet ports. A necessary check was introduced for incoming broadcast packets since they would all be classified as outgoing. This modification ensures that the proper IOQ header is appended on the packet upon being received either by a MAC or CPU input queue.

4.3.2 Testbench Packets

Given that our objective is to calculate the SHA-1 hash of a data message, we need to include in the testbench a packet containing a Libswift data message. Incoming Libswift packets vary with size depending on which Libswift messages preceed the data message. However, the number of packets must be kept to a minimum in order to make the testbench manageable when observing it through the ModelSim waveforms. Thus many PCAP testbenches must be created in order to thoroughly test the complete system.
Knowing that the NetFPGA must be able to handle a variety of Libswift packets, as well as packets from other protocols, a selection of packets was made to test our design. The testbench shown in the following list, simulates a variety of protocols and allows us to verify that non Libswift packets are not obstructed by our implementation. Other testbenches, such as to test a Libswift transfer, were also created though their details will not be listed here.

1. ARP request broadcasted from seed.
2. ARP reply from leech (NetFPGA).
3. Ping request, IPv4, ICMP, from seed.
4. Ping reply, IPv4, ICMP, from leech.
5. Libswift (IP/UDP) handshake from leech to seed.
6. Libswift handshake reply from seed to leech.
7. Libswift data request from leech to seed.
8. Libswift data message reply from seed to leech including integrity messages.

4.4 Message Parser Module

The Message Parser module determines if a Libswift data message is present in an incoming packet. Its operation is divided in two parts. First, the Message Parser must check the packet headers as explained in Section 4.4.1. Second, it must parse through the messages in the packet payload in search for a data message header.
The top level interface of the Message Parser module can be seen in Figure 4.3. Since it is connected directly into the UDP, it uses the standard NetFPGA module port system introduced in Figure 3.3. The Message Parser has a few additional ports used to communicate with the PMS. The output port *out_msg_parser_result[1:0]* signals the result of the packet analysis. The result is maintained until the PMS confirms receipt of the

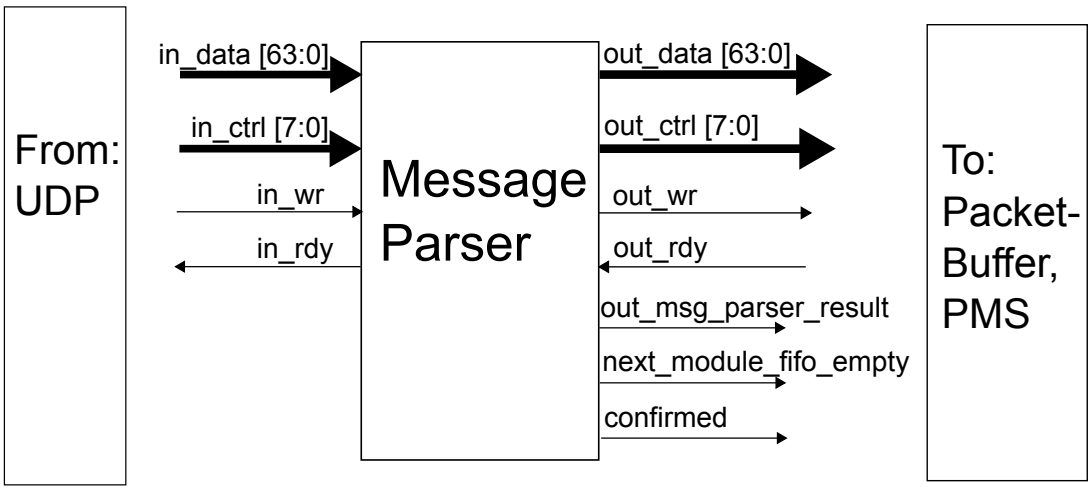

Figure 4.3: Block Diagram of Message Parser

result by the input port *confirmed*. Input port *next_module_fifo_empty* signals the Message Parser that the previous packet has already cleared the FIFO in the PMS. In the current implementation of the Message Parser, stalling the datapath is not supported. Having the proceeding FIFO cleared and ready to receive an entire packet is therefore necessary in order to prevent erroneous behavior.

4.4.1 Packet Headers

The following list contains information regarding the packet headers, fields to check, and the NFP datapath word in which these are found. The structure of a packet can be seen in Table 4.1.

1. Word 0 - IOQ - Packet port destination and packet size
2. Word 1 - Ethr - Nothing to Check
3. Word 2 - Ethr - Bits [31:16] with value 0x0800 indicating an IPv4 header
4. Word 3 - IP - Bits [7:0] with value 0x11 indicates a UDP header
5. Word 4 - IP - Nothing to Check
6. Word 5 - UDP - Bits [31:16] - Source Port with value 0d7788 (0x1E6C)
7. Word 6 - UDP - Nothing to Check.

From the IOQ header, we are interested in the packet destination port and packet size. Destination port (bytes 1 and 2) is one hot enconded output and it is necessary to check this field in order to let outgoing Libswift packets fall through. The output ports destined to the CPU are enconded as 0x2, 0x8, 0x20 and 0x80. See Table 4.2 for Input/Output port encoding. The size of the packet payload must be at least 1024 bytes in order to

word#	in_ctrl	in_data							
	7:0	63:56	55:48	47:40	39:32	31:24	23:16	15:8	7:0
0	0xFF	ioq_port_dst		ioq_word_length		ioq_port_src		ioq_byte_length	
1	0x00	mac_dst						mac_src	
2	0x00	mac_src				protocol		ip_ver	ip_tos
3	0x00	ip_length		ip_id		ip_flags		ip_ttl	protocol
4	0x00	ip_checksum		ip_src				ip_dst	
5	0x00	ip_dst		udp_src		udp_dst		udp_length	
6	0x00	udp_checksum		Libswift Payload					
7	0x00	Libswift Payload							
8	0x00	Libswift Payload							
...	0x00	...							
n-1	0x00	Libswift Payload							
n	0x08	Libswift Payload					xxxxxx		

Table 4.1: Structure of a Libswift Packet Through the NetFPGA [44]

contain a Libswift data message. Checking the size of the packet is not required, but allows for fast processing of packets too small to contain a data message. The IOQ header has two fields which contain the packet size, one in terms of 64-bit words, and another field with the total number of bytes in the packet. The word length field is ideal for us to check, since the packet must contain at least 128 64-bit words and can be checked with a simple OR gate on the most significant bits of that field. If the number in this field is lower than 128, then no data message is present.

Input Encoding	hex 0	hex 1	hex 2	hex 3	hex 4	hex 5	hex 6	hex 7
Port	MAC1	CPU1	MAC2	CPU2	MAC3	CPU3	MAC4	CPU4
Output Encoding	hex 1	hex 2	hex 4	hex 8	hex 10	hex 20	hex 40	hex 80

Table 4.2: IOQ Header Port Hexadecimal Encoding

After the IOQ header is checked, the Message Parser must check if there is an IP header, followed by a UDP header. Inside the UDP header, the Source port must be 7788 to indicate that the packet originated from Libswift. It is important to note that the port 7788 must be specified when lauching Libswift in the Seeding computer. If at any point during the verification of these headers, the packet does not match, the packet is allowed to fall through. If a Libswift packet is confirmed, the Message Parser proceeds to parse the message headers in the packet payload.

4.4.2 Parsing the Libswift Payload

The payload of a Libswift Packet contains one or more messages. Each message starts with a header which indicates the type of message it is. These messages are of varying

length, and a Libswift packet can contain almost any combination of them. It therefore becomes necessary to read the headers of all the messages in the payload in order to find the data message header.
The Message Parser uses an index register to be able to jump from one message header to the next. Upon encountering a header, the module will increment the index by the length of the current message which can consist of many datapath words. The datapath will continue, and the index register will be checked. If the next header is not in the current datapath word, the index will be decremented. If a header is present, it will be read and the index will be updated with the location value for the next header.
The Message Parser searches through all the message headers for the data header number decimal '1'. If it is found, then the packet is confirmed as containing a data message by sending the confirmed message to the PMS. The Message Parser will also write the index of the first valid byte of the data message payload to the CTRL. In other words, this index skips the header and bin number of the data message, to point directly at the content. This index facilitates finding the data which is to be fed to the SHA modules for processing. During the time the Message Parser is searching for the data message header, it also constantly monitors the CTRL bus for the End Of Packet signal. If it is encountered before a Libswift data header is reached, the packet is classified as having no data message.

4.4.3 Limitations

In the current implementation, the Message Parser has several limitations. The first is that it cannot handle a stalled datapath. If a module further down stalls the datapath, the Message Parser will handle the stalled data word as a new word which will lead to undesired behavior. The second limitation stems from the previous one in which it cannot handle multiple Libswift message headers in one word. Headers such as PEX requests contain no payload, and thus can lead to this situation. To aleviate this limitation, the Message Parser should be expanded to be able to stall the datapath when it detects that the next message header is in the current data word.

4.5 Packet Modification System (PMS)

The Packet Modification System (PMS) is the module responsible for performing all modifications to the incoming Libswift packet. It holds all incoming packets in a large FIFO, while it waits for a decision from the Message Parser concerning the Libswift Data content of the packet. If a data message is not found, the PMS will let the packet fall through. If a data message is present, the PMS updates the IOQ packet header, feeds the SHA modules with the data message payload, and appends the resulting SHA1 hash at the end of the packet.
The PMS contains the standard NFP port structure, with a few additions for communicating with the Message Parser. No actual ports for interfacing with the SHA modules are provided since they are arranged as subcomponents of the PMS and use internal signals. However, Figure 4.4 shows the port structure and internal connections to the SHA modules. In the following subsections, the individual functions of the PMS will be

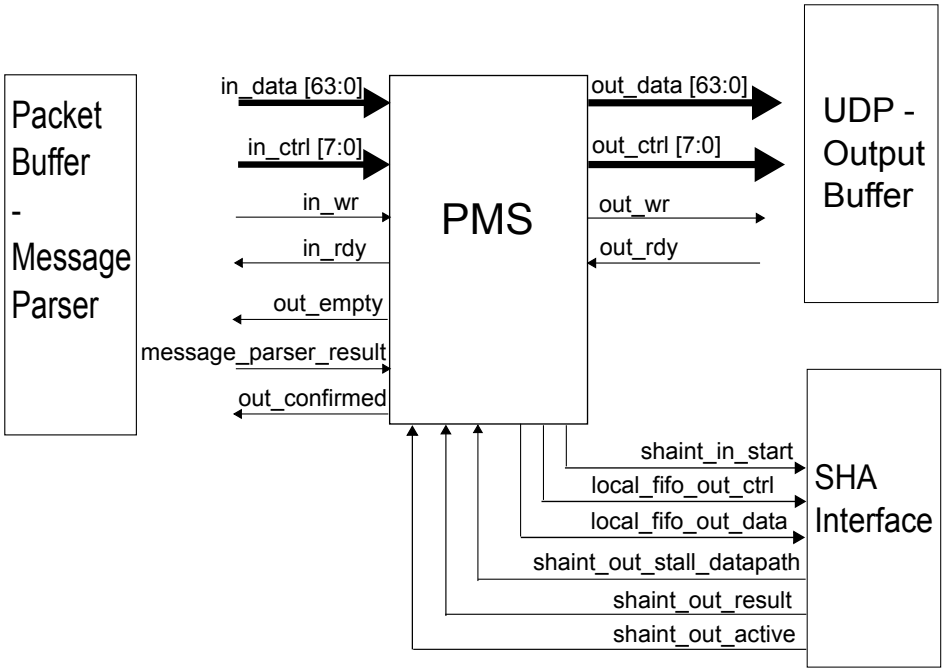

Figure 4.4: Block Diagram of PMS

described.

4.5.1 FIFO - Packet Buffer

The Packet Buffer is a FIFO large enough to hold an entire packet which is placed at the input of the PMS. Holding the packet in memory while its contents are analyzed is necessary since the IOQ packet header must be updated if we add the hash message by modifying the size field. The PMS will hold a packet until it receives a response from the Message Parser on the *message_parser_result* input port.
Upon reception of the response from the Message Parser, the PMS will issue a one clock response through the output port *out_confirmed* which allows the Message Parser to reset its registers and prepare for processing a new packet. At this point, the PMS proceeds to process the packet emptying the Packet Buffer. The Message Parser will not begin processing a new packet until the Packet Buffer has been cleared. The PMS will signal the Message Parser on the status of the buffer by the output port *out_empty*. This behavior of waiting for the buffer to clear was done to facilitate prototype development, and the cost is added latency to the packet flow.
The original design of the prototype intended to use the standard input FIFO used on every module in the NFP. The verilog interface provided in the input FIFO instantiation allows the specification of the FIFO depth. However, we realized that changing the parameters of the instantiation results in simulation and implementation errors. Therefore a new FIFO had to be generated using Xilinx CORE Generator with a fixed 2KB size specified inside CORE Generator for:

- Family: Virtex 2P
- Device: xc2vp50
- Package: 1152
- Speed: -7

4.5.2 IOQ Header Update

Updating the byte length field is done simply adding decimal 21 to the IOQ header field (signal *local_fifo_out_data[15:0]*). Updating the word length field is more complex since the number of words to add depends on which index the original packet ends. However, analysis revealed that the 3 least significant bits of the original IOQ packet byte length field determines the byte index in which the packet ends. A value of 1, 2 or 3 in these bits signifies that our hardware hash message will fit in only two extra words. Any other value corresponds to three extra words. The value computed this way is then added to the IOQ word packet length field.

4.5.3 Integrating SHA Module

Once the IOQ header has been updated, the packet is put in fall throuh state until the special control message in the control path *in_ctrl* is found. This message was inserted into the control path by the Message Parser to indicate the byte index in which the data message payload begins. The message is in the hexadecimal form 8'hFX in which X is the binary encoded index. This control message must be cleared before it leaves the module as not doing so will cause errors further down the NFP datapath.
The SHA Interface module is the top module in the set of 3 SHA modules. It is directly connected to the NFP control and datapath which facilitates the delivery of data. The PMS only has to activate the module by raising *shaint_in_start* control signal, and give control of stalling the datapath to the SHA Interface module. The PMS must then wait for the *shaint_out_active* control signal to be raised to take back control of the datapath and begin appending the result which is available in the *shaint_out_result* register from the SHA Interface.

4.5.4 Appending Result

The end of packet is signaled in the datapath by a message in the control path. This message is a one hot encoded index of the last valid byte in the datapath. When the *shaint_out_active* control signal is set high, the PMS takes control over the datapath and saves the last valid byte index. The control signal is then cleared as a new end of packet control message will be added after the hardware hash has been appended.
The PMS then proceeds to add the Libswift Hardware Hash message header (14), before begining to append the hash result which is held in a dedicated buffer coming from the SHA Interface. The structure of the Libswift Hardware message is shown in Table 4.3. Once all the contents have been appended, the new end of packet control message is added, and the packet is allowed to leave the PMS.

Byte Size	1	20
Content	Header: 14	SHA1 Hash Payload

Table 4.3: Libswift Hardware Hash Message

4.6 Secure Hash Algorithm (SHA-1)

The Secure Hash Algorithm (SHA-1) is an iterative algorithm, which processes an input message and produces as an output a 160-bit message digest which we refer to as a Hash. The algorithm is one-way which implies that the original input cannot be recovered from the hash output. Any variation in the input message is almost certainly guaranteed to produce a different hash. The algorithm is defined in FIPS 180 [15].
As explained in Section 2.3.2, the SHA-1 algorithm is used by Libswift to verify the integrity of all incoming data packets. Implementing it inside the NetFPGA and processing data messages as they come is meant to offload this work from the CPU running Libswift. In this section, a brief description of the SHA-1 algorithm will be given, followed by a discussion on the three SHA modules: SHA Core, SHA Wrapper, and the SHA Interface.

4.6.1 SHA-1 Standard

The SHA1 algorithm consists of two main stages. The first stage consists of preparing data for processing by padding the data with a special purpose message in order for the input message size to be a multiple of 512 bits. For the SHA1 algorithm, the data message is divided into blocks of 512-bit length. The second stage consists of processing the message blocks in order to generate the hash.
In our implementation, the Libswift data message is split into 16 512-bit data blocks. A 17th block is included which includes the preprocessing information which is required by the SHA1 standard. In the case of our implementation, it is of fixed size and is hard coded into the SHA Wrapper Module. The 17th block consists of the following:

1. Bit value '1'
2. A variable number of zeros in order to fill the 512-bit block. In our implementation 447 zeros are used.
3. A 64-bit binary encoded word indicating the total size of the data message to process in terms of bits. In our implemenentation, the size is 8291 bits.

The processing of one block is shown in Figure 4.5. The algorithm consists of 80 rounds, each processing data with one of three logic functions which depend on the round number. In Figure 4.5, each function block processes the data for 20 rounds. *Hb-1* is the result of the SHA1 calculation of the previous block. In the case we are processing the first block, this value is set to a default value specified by the SHA-1 standard. The input *data_input* is the current block data being processed. Variables A, B, C, D, E are the 5 32-bit components of the resulting hash. The variables Kt and Wt are also determined

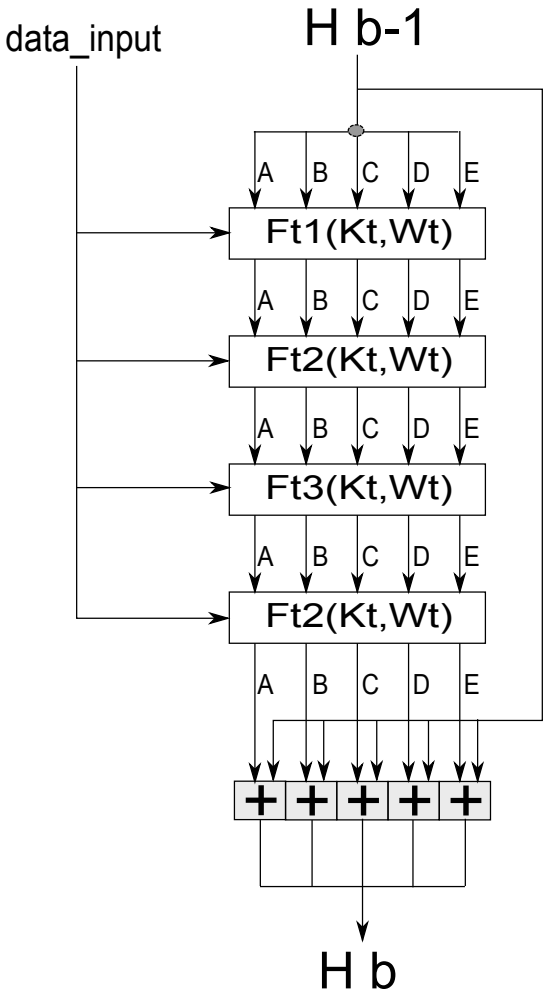

Figure 4.5: SHA-1 Processing of One Block [39]

by the current round. At the end of all rounds, the resulting message digest is added to the previous block hash result to produce the hash result. Once the 16 blocks of Libswift data, plus the extra block have been processed, the SHA-1 hash of the data message has been produced.

4.6.2 SHA Interface Module - sha_interface.v

The main function of the SHA Interface Module is to connect the UDP to the SHA modules. Even though the UDP is 64-bit wide, a libswift data message is not guaranteed to be aligned so that a 64-bit word can be split into two 32-bit words to be fed into the SHA Wrapper. Instead, a data message can start in any of the 8 bytes of a word in a datapath and cause data to be misaligned. The SHA Interface Module implements control and memory structures to guarantee correct and timely data delivery as requested by the SHA Wrapper. Given that data cannot be fed to the SHA Wrapper at the same

throughput as the UDP, the SHA Interface Module is handed control over the Datapath by the PMS through the use of the *stall_datapath* signal.

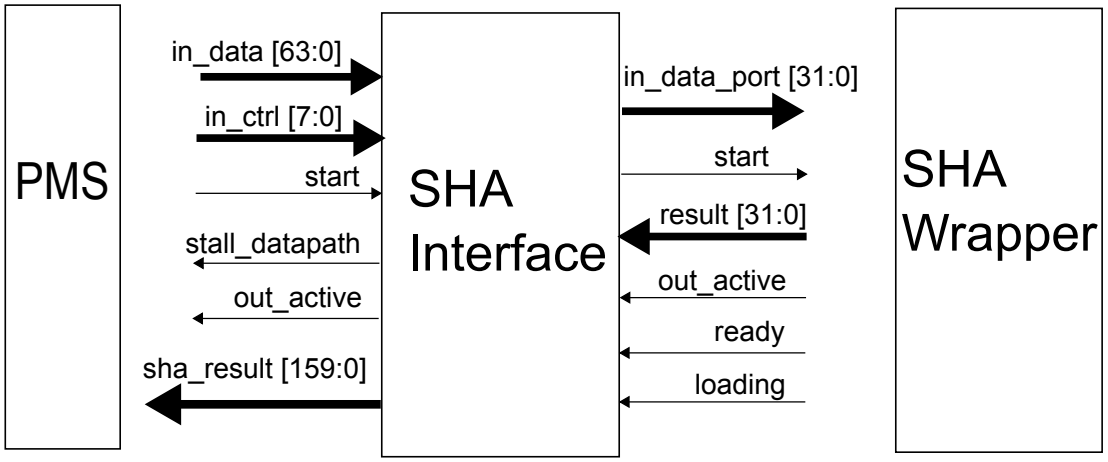

Figure 4.6: Block Diagram of SHA Interface Module

The SHA Interface block diagram is shown in Figure 4.6 in which IO ports can be seen. This module lies in between the PMS and the SHA Wrapper Module. To start the module operation, the PMS must set the input port *start* high only when the hex word 8'hFX is in the control bus of the UDP. This control word signals the start of the Libswift data message, along with the binary encoded index of the first valid data message byte in the four least significant bits of the control code. At this point, the PMS will have given control over the NFP datapath to the SHA Interface module.
The SHA Interface will save each valid word of the Libswift data message into a buffer as shown in Figure 4.7. The four most significant bytes of this buffer are fed into the data input of the SHA Wrapper. Eash time the SHA Wrapper requests new data, the buffer is shifted to the left (MSB) and new data is inserted into the buffer in order for it to not fall empty and cause undesired behavior in the SHA Wrapper.
Upon starting, the SHA Interface pushes the valid data of the first Libswift data message into the buffer. Depending on number of bytes stored in the buffer, the module will decide to start consuming data immediately, or continue loading the buffer. Three states are used for controlling the buffer and the decision process is shown in Figure 4.8. The "Start Index" value refers to the index in the control path which points to the first data word. The value of this index indicates how much valid data will be inserted into the buffer. This is shown in the central matrix to the right of the Start Index values. The central matrix is divided, grouping data by 4 bytes which can be fed into the SHA Wrapper. The states are shown to the right of the central matrix. Only Append indicates that no data is ready, and the buffer must be filled with more data. This state can only be active at the start of operations. Append and Consume indicates that data can be fed into the SHA Wrapper, and data must be inserted into the buffer to avoid buffer underflow. Only Consume is self explanatory, and enough data is in the buffer to feed the SHA Wrapper for two clock cycles. Buffer Valid Index is the internal index of the buffer.

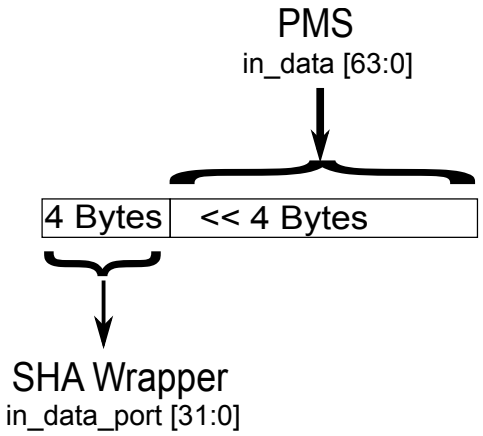

Figure 4.7: Shifting Buffer of SHA Interface Module

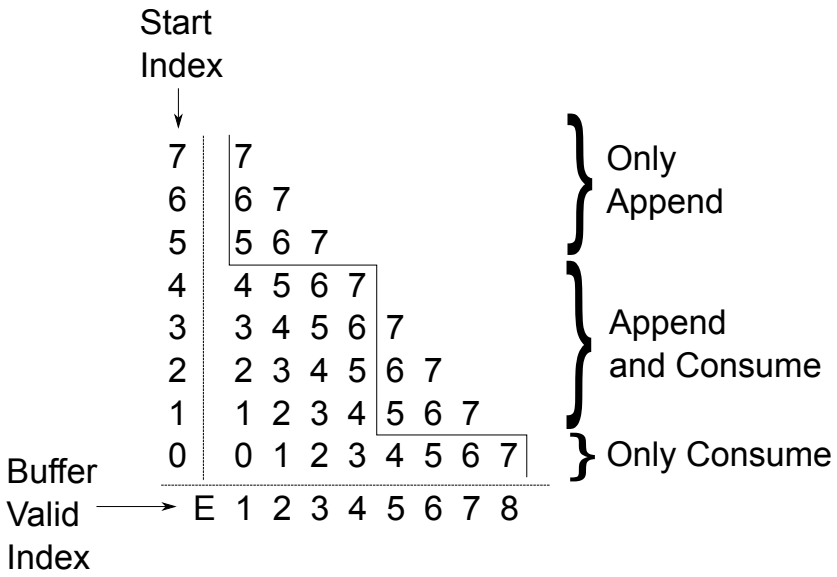

Figure 4.8: Buffer Decision Diagram of SHA Interface Module

When encountering the end of packet control signal, the module inserts the last data word into the buffer, stalls the datapath, and then goes into a group of End Of Packet control states. Once the resulting hash is ready for capture, the SHA Interface module loads the 160-bit hash result and pushes it to the *out_sha_result* output port and sets *out_active* high. It then proceeds to clear all working registers, and returns control of the UDP Datapath to the PMS.

4.6.3 SHA Wrapper Module - sha1_wrapper.v

The function of the SHA Wrapper is that of control, data input, and data padding for the SHA Core. It's main purpose is the control of the SHA Core in which it prepares the data in 512-bit blocks and instructs the SHA Core on how to process them by setting the appropriate *cmd_i* input values.
The SHA wrapper was designed to fit the specific needs of handling a Libswift data packet. The control structure takes a 1024 byte Libswift data message and feeds it to the SHA core. The SHA Wrapper will also append the correct padding as required by the SHA-1 protocol. In total, each Libswift data message consists of 16 512-bit blocks plus one 512-bit block for the padding block.

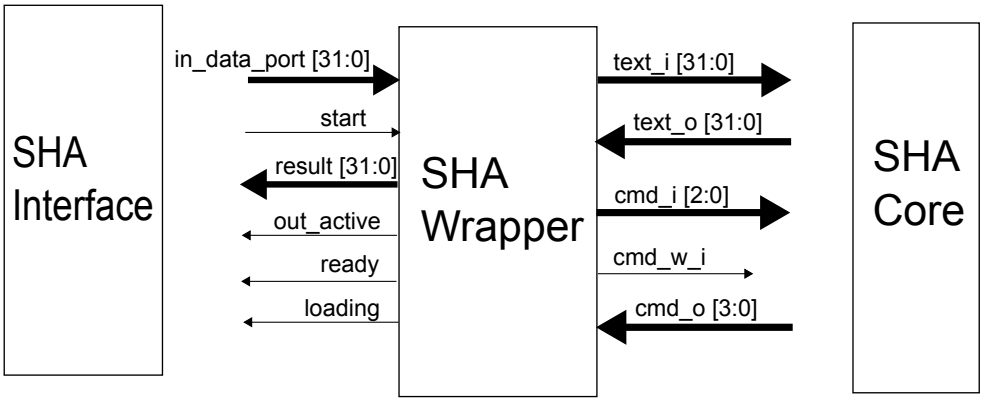

Figure 4.9: Block Diagram of SHA Wrapper Module

The SHA Wrapper block diagram is shown in Figure 4.9 in which IO ports can be seen. This module lies in between the SHA Core and the SHA Interface module. The input port *start* is set high when the SHA Interface wants to start processing a new Libswift datapacket. The SHA Core module must no longer be busy in order for *start* to be valid. On the following second clock cycle after *start* is valid, SHA Wrapper will begin taking data from the 32-bit input *in_data_port*. At this point, the *loading* output port will go up, and will remain up for the next 16 cycles corresponding to 16 32-bit words of data. This port is used as a "request new data" signal for the SHA Interface module. Once the whole data block has been received, output port *loading* will be set low. New data must be ready at the *in_data_port* for further capture.
SHA Wrapper expects to receive 16 512-bit blocks of data, and will only present the result of the SHA calculation after all these blocks, plus the padding block have been processed by the SHA Core. Since each block takes 80 clock cycles to process, and the Core must process 17 blocks, we can achieve a theoretical minimum time of 1360 clock cycles per Libswift data message. Once all calculations are done and the result is ready for output, the output port *out_active* will be set high one clock cycle before the resulting hash is presented on the *result* output port as a set of five 32-bit words.
Since the SHA Wrapper expects to receive a fixed input size of 1024 bytes, it is not currently able to handle data messages of varying length. Given that the last bin of a

Libswift data transfer can be of a smaller size, care must be taken to avoid a testing scenario in which a data message of innapropriate size is fed into the SHA Wrapper.
For verification purposes, a simple testbench was created with the file name *sha1_wrapper_testbench.v*. This testbench simply controls the input ports and pushes in constant data into the SHA Wrapper. The resulting output is then verified manually.

4.6.4 SHA Core Module - sha.v

In order to implement the core of the SHA algorithm, the verilog implementation by [19] was used from Open Cores. This module implements the structure seen in Figure 4.5 which pushes one block of data through the algorithm. It does not have any control structures for multiple blocks, nor for padding the data as required by the SHA-1 algorithm. Such control functionality is handled by the SHA Wrapper module described in Section 4.6.3.

4.6.4.1 Functionality

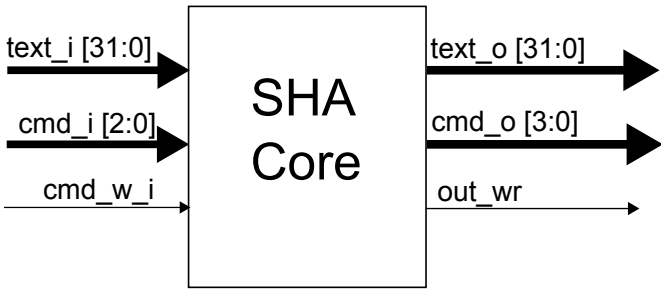

Figure 4.10: Block Diagram of SHA Core Module

The top level block diagram of the module is shown in Figure 4.10 which shows the input/output ports of the module. The SHA Core is controlled by the *cmd_i[2:0]* control port and the *cmd_w_i* port which signals the Core that a control input is valid. The *cmd_o[3:0]* output port signals the SHA Wrapper of the status of the Core. Both *cmd_o[3:0]* and *cmd_o[3:0]* have the same bit meaning except for the most significant bit. Their value is explained in Table 4.4. The *cmd_w_i* input should go up for one clock cycle only when *cmd_i* is valid.

	bit 3	bit 2	bit 1	bit 0
0	Core Ready	First Block	Write Data False	Read Data False
1	Core Busy	Internal Block	Write Data True	Read Data True

Table 4.4: SHA Core CMD Ports

To write a block into the SHA Core, *cmd_o[3]* must be low, and *cmd_i* input port must be set at "010" and *cmd_w_i* set high for on clock cycle. Data will be captured at the

clock cycle after *cmd_w_i* was set high and the SHA Core will capture 32-bit words of data on every clock until 16 words have been captured. Following, the SHA Core will proceed to compute the SHA-1 hash of this block. If the block to be compute is not the first block, then *cmd_o[3]* must be set to "110" instead of "010" which will signal the SHA Core to use the internal hash already computed from a previous block.
To read a resulting hash from the SHA Core, *cmd_o[3]* must be low, and *cmd_i* input port must be set at "001" and *cmd_w_i* set high for on clock cycle. Data will be ready at *result [31:0]* on the second cycle following this pulse. Every clock cycle a new 32-bit data word will be on the output for a total of 5 words (160 bits).

4.6.4.2 SHA1 Core Acceleration

Although the documentation of the SHA Core states that the module can run at 250 MHz, after implementing on the NetFPGA it was noted that the module could only run at 87.780MHz and did not meet timing requirements set at 125MHz by the UDP. Consulting the timing report after implementation, it was noted that the datapath from *round* to the register *A* was not meeting timing requirements. The register *round* simply determines in which of the 80 rounds the algorithm is currently calculating. The critical path can be seen in Figure 4.11 as the *Ft* block depends on *round* and the next clock cycle value of A cannot be calculated on time. This figure also contains an extra adder after the value of A which is used only in the final round of operations. It was thus included as part of the critical path.
Several solutions where studied which would accelerate the SHA Core to the 125MHz required frequency. Efforts were hindered by the limitation of modifying an existing design, and the challenge presented by the strict Verilog coding style of the implementation. Further, the single step diagram shown in Figure 4.11 can lead to an oversimplified solution.
After exploring many solutions, the resulting modifications are shown in Figure 4.12. The addition of the register E and variable Kt was done one clock cycle in advance. Special control structures had to be implemented to compute in advance the future value of E and Kt as the existing code did not provide access to this data.
The decision of which function Ft to use was unrolled. All function values are precalculated and added to the predetermined values of A. A final decision of which value of A to use is done at the end. To further accelerate the core, the final addition required in the final round was moved to the following clock cycle. These modifications allowed the SHA Core to run at 135Mhz. The performance and cost difference as indicated by the synthesis reports are shown in Table 4.5 which includes the cost of the SHA Wrapper and SHA Interface.

4.6.4.3 Verification - test_sha1_gradual.v

Although the Open Cores package came with a testbench for the Core, it was not very extensive. Another problem was that because of a lack of documentation, the function of each port was not clear. An expanded testbench was designed which gives flexibility in how many blocks to process, and also works as a blueprint for the behavior that the SHA Wrapper module is expected to have.

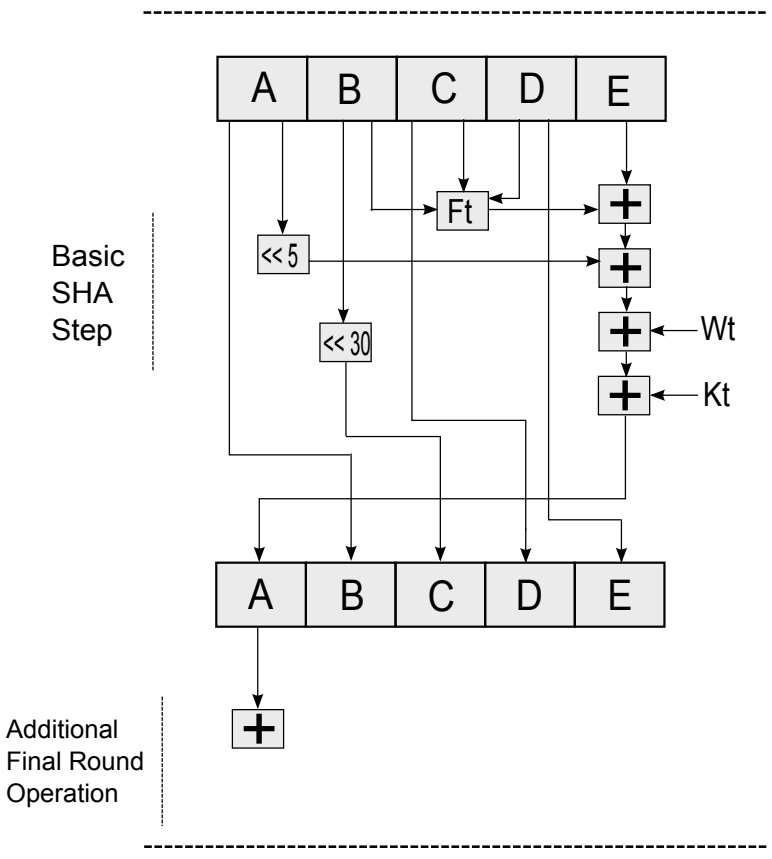

Figure 4.11: Final Step of the SHA-1 Algorithm [39]

	Original		Accelerated	
Frequency	87.780MHz		135.686MHz	
Slices	1,945	8%	1,927	8%
Flip Flops	1,188	2%	1,231	2%
LUTs	3,305	6%	3,214	6%

Table 4.5: SHA1 Group Modules - Cost Comparison for Virtex II

With the default test parameters, the Libswift Data message from our PCAP testbench is inserted into the SHA Core along with the necessary padding block. The Core was verified as working by comparing the output against the SHA-1 Hash calculated using several online SHA-1 calculators [8] [38].

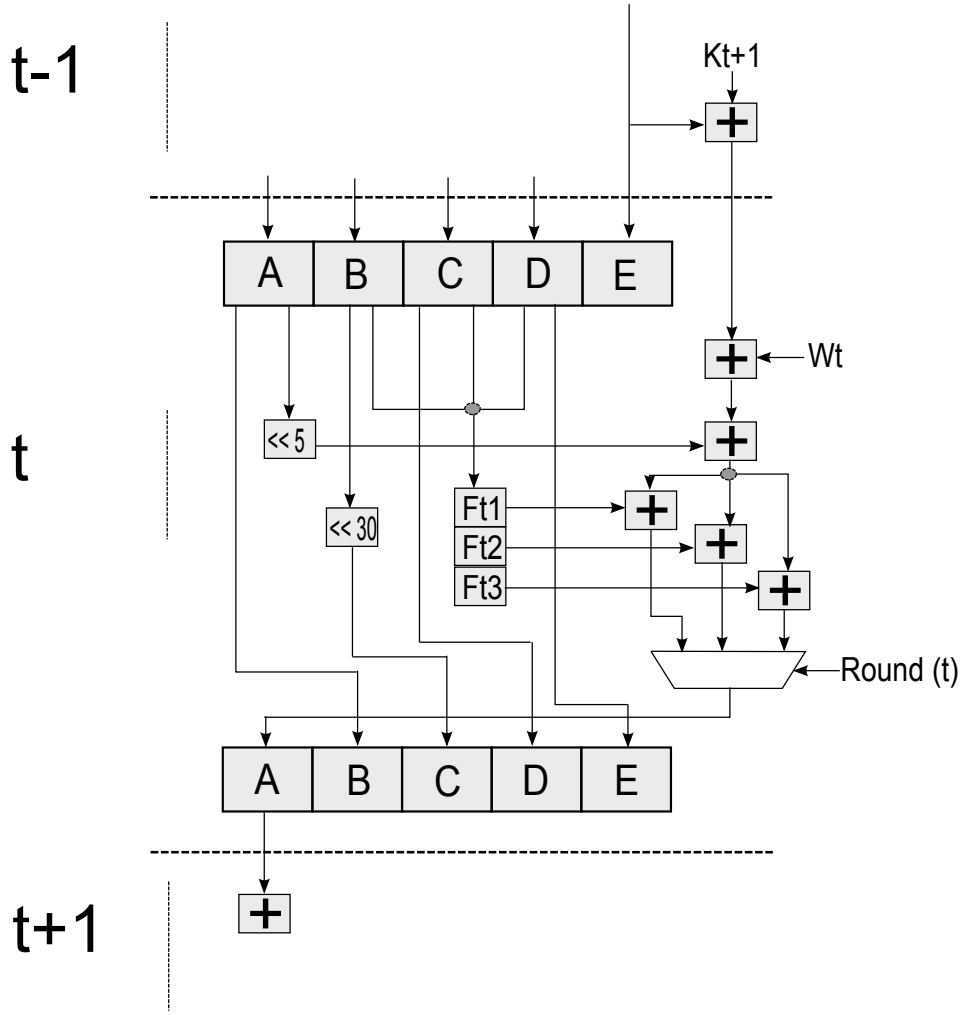

Figure 4.12: Acceleration of the SHA-1 Algorithm

5 Experimental Results

This Chapter deals with the result, their analysis, and optimizations for the SHA1 Accelerator. Section 5.1 talks about the results obtained both from simulation as well as from implementation. A discussion on the findings and their significance is also discussed. Section 5.2 explores possible improvement paths which can be taken with respect to the SHA1 Accelerator.

5.1 Performance Results

In this Section, the performance of the SHA1 Accelerator of a Libswift Data packet will be meassured and analyzed. The measurements are based both on simulation and real world implementation measurements. All the components were tested and are working, but due to lack of time, some integration issues remained. These issues, explained later in this document, prevented us from collecting the results from a complete Libswift transfer. Regardless, we will give arguments that will show that these integration issues do not affect the results presented in a significant way.

5.1.1 Execution Time

We define Execution Time as the moment the first word of a packet enters the UDP, until the last word of the packet exists the UDP. The execution time is a key metric in analyzing the performance of our implemented SHA1 Accelerator. By comparing the execution time of our implementation, to the execution time of the same packet traversing through the unmodified Reference NIC, we can obtain the overhead cost of calculating the SHA1 Hash in hardware. The result of this calculation, can be compared to the measured time it takes for a CPU to calculate the SHA1 Hash in order to get insight into the performance of the SHA1 accelerator.
Per packet execution time is difficult to measure inside an FPGA due to having no visibility on the inner workings of the hardware. Trying to measure execution time as the time between an external server sending a packet to the NetFPGA client computer, and the client Operating System receiving the packet, introduces non-deterministic events which makes measurements for a single packet unrealiable. These non-deterministic events can be caused by CPU Scheduling variations, DMA bus traffic, to unnexpected interrupts.
A way around this problem is using simulation results in ModelSim to measure the number of clock cycles a single packet spends inside the UDP, since this operation is deterministic. Knowing the clock frequency of the NetFPGA is set at 125MHz allows us to calculate the execution time of a single Libswift data packet.
The results of these measurements are shown in Table 5.1. *In Time* indicates the time

	Reference NIC with CPU	SHA-1 Accelerator NIC
In Time	47068ns	47324ns
Out Time	49828ns	60924ns
Total Time	2760ns	13600ns
Total Clk Cycles	345	1700
SHA1 Clk Cycles	-	1355
SHA1 Calc. Time	16720ns	10840ns
Speedup	1.0	1.54

Table 5.1: Performance Results of a Libswift Data Packet - Average of 10K Samples for CPU Time

when the first word of the packet enters the UDP. *Out Time* indicates the time when the last word of the packet leaves the UDP. *Total Time* is the time taken by a packet to traverse the UDP. *Total Clk Cycles* is the total amount of clock cycles taken by a packet to pass through the UDP. *SHA1 Clk Cycles* is the extra time it takes for a packet to traverse the UDP of our SHA1 Accelerated NIC with respect to the unmodified Reference NIC. This metric is the overhead cost in order to calculate the SHA1 hash and append it onto the packet. This metric is expanded to the *SHA1 Calc. Time* which was calculated by multiplying the overhead *SHA1 Clk Cycles* by the clock period of 8ns corresponding to the frequency of 125MHz. The *SHA1 Calc. Time* is also shown for the CPU calculating the SHA1.
This final metric was obtained outside of simulation since it is the CPU generating the hash of incoming data packets. The specifications of the computer used are the same as in Section 2.4. A transfer of a large file was done through Libswift between two computers directly connected by a 1Gbit ethernet cable. All data would pass from the seed computer, to the leecher on the NetFPGA using the Reference NIC. Timers where inserted into the Libswift code in the area which calculates the SHA1 hash in order to measure the execution time. A total of 10 thousand samples were taken, and the average time calculated which was presented in Table 5.1.

5.1.1.1 Interpreting Results

The most important metric presented in Table 5.1, is that of the comparison between the values of *SHA1 Calc. Time*. The SHA1 Calculation Time for the SHA1 Accelerator at 10840ns corresponds to a speedup of 1.54 with regards to the Reference NIC and CPU. We define Speedup following its use in Parallel Computing as the execution time of the original implementation, divided by the execution time of the modified implementation. It is essential to make clear that this speedup does not represent a speedup in the total time taken by Libswift to complete an entire transfer. It only represents the speedup of calculating the SHA1 hash of a single Libswift data message. The Libswift protocol running on the CPU will still need to calculate the hashes of the inner nodes of the Merkle Hash Tree.
A cost benefit analysis of this data will be presented in Section 5.1.2. Further analysis

steming from the knowledge of the SHA Core being the the most time consuming module in the critical path will be presented. In particular, proposals for future improvements will be explored in Section 5.2.

5.1.1.2 Comparisson with Expected Results

An unnexpected result from Table 5.1 is that the *SHA1 Clk Cycles* is lower than our expected minimum cycle requirement. Considering that our SHA Core is implemented in a sequential manner, the processing of each block takes 80 clock cycles. Considering we have 17 blocks to process, and ignoring any control overhead cycles, the minimum execution time for a Libswift data message would be 1360 clock cycles. Indeed, measuring the total execution time in ModelSim for the SHA Core to process one data packet we see that it takes 1445 clock cycles which is close to our estimation. However, the measured *SHA1 Clk Cycles* at 1355, which is the extra time it takes to process a package in comparisson to the unmodified Reference NIC, is smaller than our predicted minimum execution time of 1360 cycles.
To understand how this happens, lets look at the *Total Clk Cycles* for the Reference NIC. This value consists of the time it takes for a packet to not only be processed, but also the time it takes the packet to traverse, word by word, through all the modules and buffers. Considering that a module such as the Output Port Lookup takes only 2 cycles to reach a decision and output the first word that appeared on its input, we can conclude that most time in the NIC is spent in simply traversing the datapath and not being stalled by control logic. The total time it takes for 1024 bytes of Libswift data to traverse a one level buffer in the UDP is 128 clock cycles. During the time the packet takes to traverse the PMS module, the SHA Core is concurrently taking the data and processing it as the packet is moving, albeit at half the normal speed. This parallelism allows our SHA1 Accelerator to reach speeds which are better than our predicted minimum operation time.

5.1.2 Area Cost

The area cost is defined as the number of resources used in the FPGA for any given implementation. In Table 5.2 we compare the area cost between the Reference NIC, and our SHA1 Accelerator labelled as "libswift_sha1". The area cost is an essential metric in the cost benefit analysis in any given implementation strategy.

	reference_nic		libswift_sha1	
Slices	12,603	53%	14,944	63%
Flip Flops	12303	26%	14074	29%
LUTs	17,581	37%	23,100	48%
BRAM	89	38%	91	39%

Table 5.2: Area Cost Comparison of Implementation

In Table 5.2, we can observe that our SHA1 Accelerator uses about 10% more FPGA Slices and LUTs, though only 3% more flip flops than the standard Reference NIC. These

resources account for all control and arithmetic logic, as well as all memories elements in the form of flip flops. Concerning our implemented modules, these resources account for the Message Parser, the PMS, and all three of the SHA modules with the only exception of the large packet buffer.
Further analysis into the distribution of resources can be done when we see the area cost of the 3 SHA1 Modules separately (SHA Interface, Wrapper, and Core). These results are shown again in Table 5.3. From the 10% increase in slices used by the SHA1 Accelerator, the three SHA Modules account for 8% of that increase. The Message Parser and PMS thus account for only 2% of the Virtex II slices. Similar results are seen when comparing the number of Flip Flops and LUTs with the Message Parser and PMS utilizing 1% more Flip Flops and 5% more LUTs. It becomes clear from this comparisson, that the SHA1 Modules have the highest area cost.

	Original		Accelerated	
Frequency	87.780MHz		135.686MHz	
Slices	1,945	8%	1,927	8%
Flip Flops	1,188	2%	1,231	2%
LUTs	3,305	6%	3,214	6%

Table 5.3: SHA1 Group Modules (Interface, Wrapper, and Core) - Cost Comparison

The packet buffer is stored in a dedicated FIFO which is implemented in the BRAM. As can be seen in Table 5.2, our packet buffer takes less than 1% of BRAM resources accounting for only 2 out of a total available 232 BRAMs. The amount of memory available in the Virtex II FPGA makes it possible to implement the Merkle Hash Trees of files of medium length, or to consider expanding the peformance of the SHA1 Accelerator by unstalling the datapath to allow regular ethernet traffic to pass through while Libswift Data packets are stored in dedicated buffers. These ideas are explored in Section 5.2.

5.1.3 Latency - Overhead Packet Delay

To test the latency introced by the the SHA1 Accelerator modules in the UDP, a non-Libswift file transfer was conducted and the arrival times of the packets were measured. Small packet delays are expected to be introduced by the FIFO interfaces at the Message Parser and PMS. However, due to the design decision taken during building of the prototype, which focused also in obtaining a working design in a limited amount of time, some delay is introduced. This delay is due to the fact that the Packet Buffer Module needs to flush before the Message Parser begins processing the next packet. This is not a fundamental limitation, but it simplified the initial development.
A transfer of a file of about 1MB in size was made using the Secure Copy application using SSH. The NetFPGA computer sends out a transfer request a time zero, and the server responds by transfering the file. The arrival time of the first 1000 packets was recorded and is shown in Figure 5.1. This procedure was done with two implementations in the NetFPGA. The NetFPGA running an unmodified reference NIC is shown in red, while the NetFPGA running the SHA1 Accelerator is shown in green.

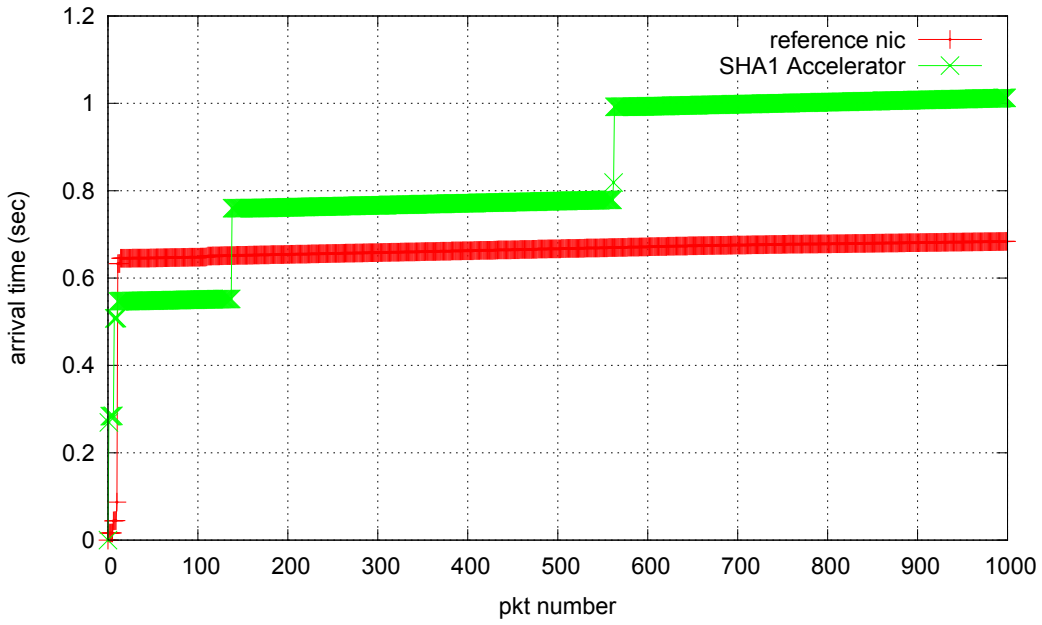

Figure 5.1: Packet Arrival Time of Secure Copy Transfer

Looking at Figure 5.1, we can see that the SHA1 Accelerator arrival times are in the shape of a step function. The reason for this shape is that packets are being dropped. Indeed, analysis of the contents of the packets around the step time, reveals that packets are being retransmitted which implies a loss of packets. Looking back at the design of the SHA1 Accelerator in Figure 4.2, we remember that the Packet Buffer must flush before the Message Parser begins processing a new packet. This operating behavior was done in order to facilitate the design of the prototype. However, this decision does cause delay in the packet arrival time. As packets arrive one after the other, the delay gets accumulated and eventually the input buffers overflow, causing packets to be dropped.
In order to compare the packet arrival without the packets being dropped, we have zoomed in to the arrival time between packet 200 and 400, and have set the start time to zero. This graph, shown in Figure 5.2, allows us to see how the arrival delay accumulates between successive arriving packets and gives an insight into the time dealy that happens inside the NetFPGA. Starting at time zero, the SHA1 Accelerator shown in the green graph starts diverging from the reference NIC shown in red.
We define the inter packet delay to be the difference between the arrival time of one packet with respect to the arrival time of the previous packet. Obtaining the inter packet delay of all packets in Figure 5.2, we can calculate the average inter packet delay. The average for the Reference NIC is 41.566us, and 47.172us for the SHA1 Accelerator, a difference of 5.606us. The inpact of this delay difference can be seen in the diverging of the graphs in Figure 5.2.

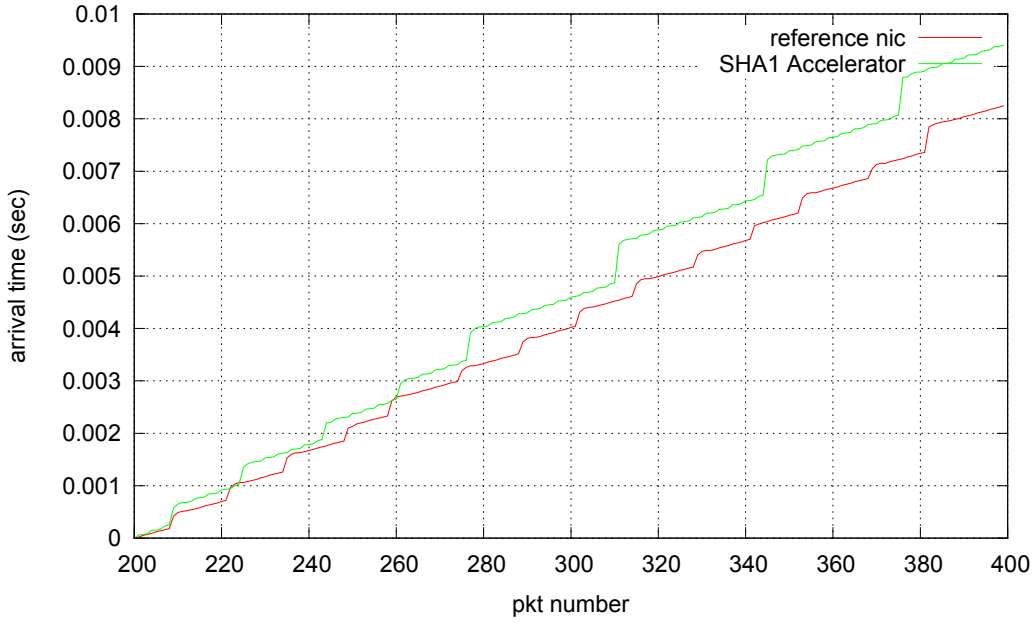

Figure 5.2: Packet Arrival Time Between Packet 200 and 400

5.1.4 Application Level Improvements

When discussing future improvements, it is necessary to understand what the maximum theoretical performance increase of Libswift that could be obtained with acceleration. Refering to the Libswift Profiling Section 2.5, it was understood that Libswift spends about 20% of its execution time calculating the SHA1 hash of the leaf nodes of the MHT. If our future accelerator could eleminate this delay entirely, the workload of an excecuting Libswift would be reduced by 20%. How this workload reduction and speed increase in calculating the SHA1 of leaf nodes would translate to the speed of file transfers would depend heavily on the network and thus an estimate is difficult to make. However, if assume that a lower workload is directly translatable to speed, then the speedup achieved by reducing the SHA1 hash leaf calculations to zero would be 1.25.

5.2 Estimated Results for Optimizations

In this Section, several optimization for the base design of our SHA1 Accelerator are presented, and their estimated results are discussed. Optimizations follow two separate strategies: modifying the critical path of the SHA Core to increase the clock frequency, and parallelizing the SHA1 Accelerator to achieve higher throughput.

5.2.1 Critical Path Modification

The critical path of our SHA1 Accelerator lies in the SHA Core. This is the most time consuming module by several orders of magnitude. By using a different system clock for

the SHA Core, the critical path can be shortenned and use of a faster clock provides a performance improvement. In contrast, the SHA Core can be unrolled and if necessary, a slower clock can be used resulting in higher throughput. Unfortunately, the Virtex II on board the current NetFPGA 1G does not have any spare system clocks. Therefore, these ideas can only be implemented in the newer NetFPGA 10G which has an onboard Virtex 5 FPGA.

5.2.1.1 Shorter Critical Path

Using a faster clock to drive the SHA Core involves shortening the critical path of the SHA Core in order to drive a faster clock. The problem posed by this approach is essentially the same as when the original SHA Core did not meet our timing requirements. Further improvements can be done in estimating in advance the value of registers and precomputing sums. Although this approach can offer some immediate performance advantages, it is not a strategy that offers large improvement because it is limited by the critical path which at a certain point will not be able to be reduced further.
Taking into consideration the scheduling improvements already done in Section 4.6.4.2, further improvements can be made. Taking Figure 4.12 as reference, we can observe that several operations can be rescheduled. The function Ft1, Ft2, and Ft3 depend only on registers B, C, and D whose values are simple shift operation from previous known values. Further, the decision on which function Ft to chose can also be done ahead of time. Knowing that the value of Wt can be easily precomputed a clock cycle before it is needed, Wt can be added with the value of Ft. This method will reduce the chain of operations to get the value of A to just one logical shift and two additions.

5.2.1.2 Loop Unrolling

Unrolling the loop for the SHA1 algorithm has some similarities with the proposal in Section 5.2.1.1 in which necessary calculations are precalculated as soon as the required data is available. However, by unrolling the loop, redundant operations are eliminated, and each round operation after the first are simplified.
Consider the critical path of the SHA1 round which can be expressed as

$$A_{t+1} = A_t <<< 5 + f_t(B_t, C_t, D_t) + E_t + K_t + W_t \tag{5.1}$$

The values of B, C, D, and E only require elementary shifting operations to calculate in which each of these depends on the previous variable. In other words, B depends on A, C depends on B, D depends on C. The critical path falls on variable A as it depends on all variables. However, this means that the shifting nature of the algorithm can be exploited to do most calculations in parallel.
This idea was explored in [16] in which they unrolled Equation 5.1 by a factor of five. The resulting operation is shown as

$$A_{t+5} = A_{t+4} <<< 5+[f_{t+4}(A_{t+3}, A_{t+2} <<< 30, A_{t+1} <<< 30)[A_t <<< 30+\Sigma K_{t+4} W_{t+4}] \tag{5.2}$$

As can be seen, At+5 depends on arithmetic operations from values which are known for more than one round, and only a rotation operation from the previous value of At+4 which is in the critical path.

This method naturally extends the critical path for the SHA1 algorithm which will force the SHA Core to use a slower clock. However, even with a slower clock, the throughput can be increased by shortening the number of rounds to calculate. The result of the prototype in [16] show on a Virtex FPGA, speedup of almost 2.0 can be achieved compared to a sequencial implementation. However, area cost can be almost three times as many slices as the sequencial version.
The complexity of Loop Unrolling compared to shortening the critical path as in Section 5.2.1.1 is much higher. Strong understanding on parallel algorithms, scheduling and resource allocation is necessary. The implementation complexity in Verilog would also be much greater. However, verification efforts would remain simple as any small mistake would propagate to the output and cause a significantly different hash result due to the nature of the SHA1 algorithm.

5.2.1.3 Comparisson

Table 5.4 shows the comparisson between the currently avaiable SHA Cores, and the proposed cores. *Core Frequency* indicates the clock speed driving the SHA Core. *clk cycles @ 125MHz* is an estimate of how many NetFPGA system clock cycles it will take the Core to compute the SHA1 Hash. The *Total Time* is the last metric converted to nano seconds. Lastly, *Speedup* lists the expected speedup of all cores compared to the measured time it took the CPU using the reference NIC to compute the SHA1 hash.

	reference nic	original open source	sha1 accelerator	estimated rescheduled	estimated unrolled
core frequency (MHz)	-	87	125	160	95
clk cycles @ 125Mhz	-	1947	1355	1058	973
total time (ns)	16720	15575	10840	8469	7787
speedup	1	1.073	1.54	1.97	2.14

Table 5.4: Comparisson of Proposed SHA Core Estimated Performance

Figure 5.3 shows the bar graph on the expected Speedup of these proposals when compared to the base speed of using the reference NIC with the CPU. The original open source design, even though running at a low frequency of 87MHz, can still outperform the CPU. However, time overhead costs when integrated into the SHA1 Accelerator, makes this design an unnatractive option. The *sha1_accelerator* is the optimized core that was implemented with the SHA1 Accelerator in this thesis. When looking at the area cost in Figure 5.4, we can see that this design does not consume more area than the original open source design. *estimated_rescheduled* is the proposal for shortening the critical path explained in Section 5.2.1.1. By continuing the operation rescheduling method as used in the *sha1_accelerator*, we can achieve a greater speedup of 1.97, while still using the same area. Estimates for *estimated_unrolled* were based on work presented in [16]. We can observe that this design can achieve the highest speedup of 2.14. However, the area cost, at 24% of the Virtex II Pro, is too high when compared to the performance and area cost of the *estimated_rescheduled* core.

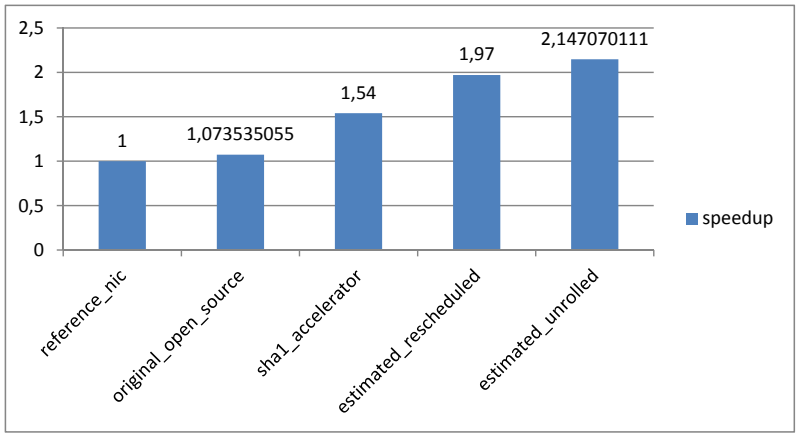

Figure 5.3: Proposed SHA Core - Speedup Comparisson

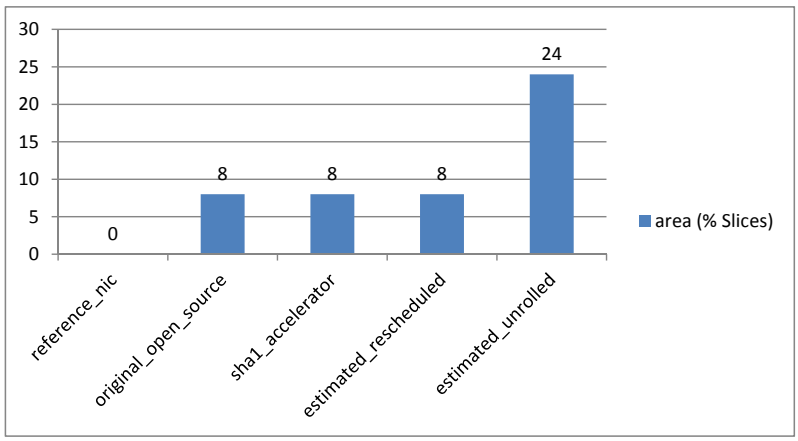

Figure 5.4: Proposed SHA Core - Speedup Comparisson

5.2.2 Parallel Processing

Parallel Processing involves instantiating multiple SHA1 Accelerators and distribute the workload of multiple Libswift Data packets between them. These proposals can increase

the throughput of the SHA1 Accelerator only if multiple Libswift data packets are received. The changes to our current SHA1 Accelerator lie in the system level as multiple modules must be modified or moved from their current position. The ideas explored in this section are complementary to the ideas in modifying the critical in Section 5.2.1.

5.2.2.1 Multiple Streams Accelerator - Parallel Processing

The Multiple Streams Accelerator is the easiest modification to implement with the current working version of the SHA1 Accelerator. This modification consists of instantiating a SHA1 Accelerator just before each of the four CPU output queues as shown in Figure 5.5. This provides a dedicated SHA1 Accelerator to each Libswift stream provided that each uses a different port.

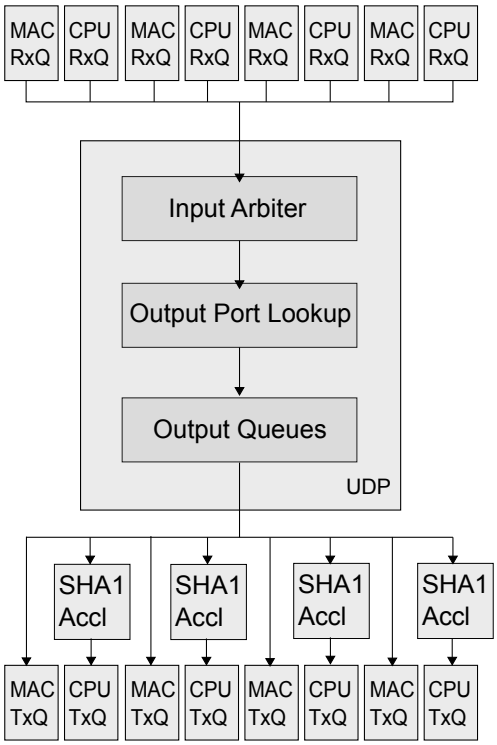

Figure 5.5: Multiple SHA1 Accelerators for Parallel Streams

This implementation eases the dissadvantage of a blocked datapath in our current accelerator, by only blocking specific output ports, granted that no spill over congestion occurs. However, this design offers no advantage for multiple streams using the same port. The ease in which this design can be implemented can provide a prototype in which the performance of multiple streams can be assessed.
The complexity of implementing this design is trivial since the implementation of the current SHA1 Accelerator is left intact. The instantiation of four SHA1 Accelerators and the proper connections have to be made in the UDP. Given that our current version of the SHA1 Accelerator uses about 10% of FPGA slices, instantiating four accelerators

would bring total Slice usage up to the uncomfortable 93% total usage.

5.2.2.2 Unblocking Datapath - Packet Buffer Queues

Packet Buffer Queues, shown in Figure 5.6, are a group of Libswift data queues which holds a Libswift data packet until a SHA1 Accelerator is free to process it. Packet Buffer Queues offer many advantages. First, it unblocks the datapath to regular traffic as a Libswift data packet will be put aside in a dedicated queue, and other packets will be allowed to pass through the datapath. Secondly, the Packet Buffer Queues system allows for a variable number of SHA1 Accelerators to be instantiated depending on performance and area requirements. These SHA1 Accelerators are assigned a packet from the Packet Buffer Queues on a first come first serve basis. Once a packet is processed, the hash is appended to the end of the packet and it is inserted back into the datapath. IF the Packet Buffer Queues are full, the Libswift Data packet can be allowed to fall through the datapath to be processed by the CPU.

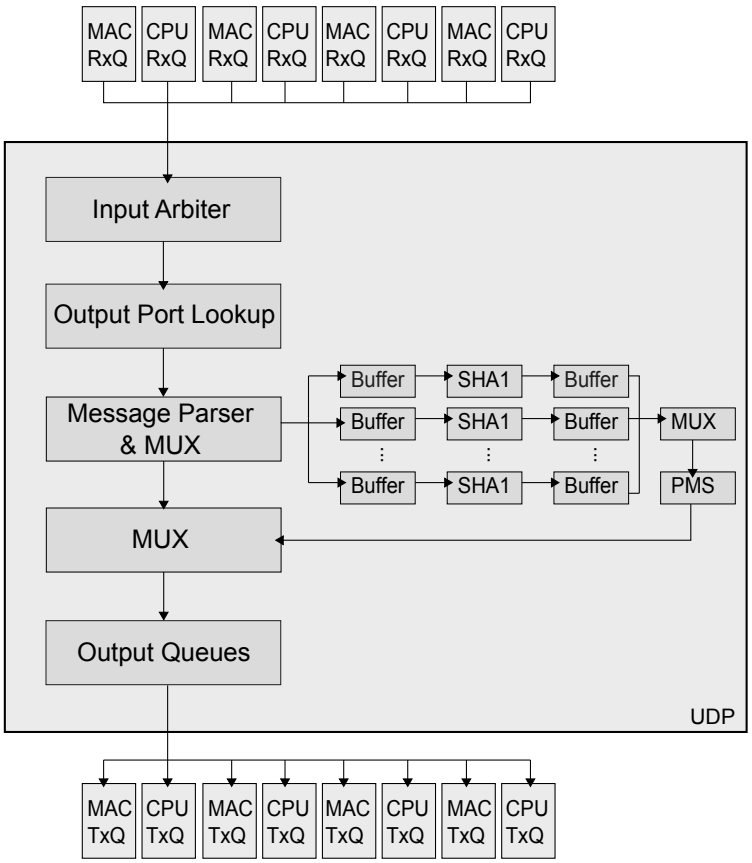

Figure 5.6: Packet Buffer Queues

This approach offers parallelization for all Libswift streams passing through the network interface, without the risk of blocking the datapath. Further, if only one Libswift stream

is passing through, all SHA1 Accelerators will be assigned to this stream. Considering that each packet buffer only accounts for 2 BRAMs, a large ammount of these can be instantiated to make a relatively large queue. However, FPGA Slice requirements are a concern. As with the Multiple Streams Accelerator in Section 5.2.2.1, the cost of each SHA1 Accelerator remains relatively the same. A theoretical maximum of four accelerators can thus be implemented.

5.3 Remarks

In this chapter, the results and potential for future work was explored. The approach used showed that even using outdated technology, we can deliver higher performance for calculating the SHA1 hash compared to the CPU. Area analysis for the SHA1 Accelerator showed that enough area is available for further improvements, however, overly ambitions plans will quickly be constrained on the Virtex II.
For future development of the SHA1 Accelerator, a combination of increased throughput of the SHA Core, along with parallelization as shown in the Packet Buffer Queues is the best option. This solution would increase throughput considerably, at an area cost that can fit in the Virtex II. However, any future developments for the Libswift ICN should focus on having the smallest SHA1 Accelerator possible to allow for more functionality to be implemented. Further, to ease the area constraint and lack of spare system clocks, we recommend using the newer generation NetFPGA 10G which has an onboard Virtex 5 FPGA.

6 Conclusions

This thesis presented the work on the initial development of the Libswift-PPSPP Information Centric Router. This chapter will review our achieved objectives, summarize our contributions, and present suggestions for future work.

6.1 Achieved Objectives

We achieved the objectives specified in Section 1.3, by designing and implementing the SHA1 Accelerator. More specifically, we achieved the following:

- Identified Libswift packets
- Computed the SHA1 hash of a Libswift data message in hardware
- Appended the SHA1 hash to the end of the packet in the form of a hash messages
- Achieved higher per-packet efficiency against the CPU, with a speedup of 1.54

From a design and engineering viewpoint, we demostrated the success of implementing a SHA1 Accelerator within the NetFPGA. Performance results indicate that the NetFPGA outperforms a general purpose processor in hash calculations. Concerning future work on the Libswift-PPSPP Information Centric Router, area occupation analysis suggest that the design and implementation of a router is possible with newer FPGA technology. Advances in documentation, testing tools and experience will allow for faster development of future work.

6.2 Other Contributions

The detailed information in Chapter 2 gave insight into the usage and protocol specifications of Libswift. The detailed performance analysis section in that chapter, identifies the SHA1 Hash computation as the main computational bottleneck. Although initial development in the NetFPGA was hindered by lack of documentation, the content in Chapter 3 lessens the startup time of any future project by providing detailed information and instructions on the usage of the platform which we consider an important achievement of this project.
Chapter 4 gives documentation on the implementation details of the SHA1 Accelerator prototype. We give the system requirements, outline limitations for the prototype, and discuss design decisions. The PCAP testbench developed in order to speed up the verification process is presented. Work on shortening the critical path of the SHA Core was also performed in order to meet timing requirements.

In Chapter 5 we have shown that per packet execution time using the SHA1 Accelerator achieved a speedup of 1.54 with respect to the software execution of Libswift. Latency introduced to regular traffic currently has a measurable impact, and was expected for the first prototype. However, with the proposed system improvements, throughput will not be affected. The area cost for the SHA1 Accelerator is moderate. Given more time for implementation, optimization of the SHA Core are suggested. Section 5.2 presents various optimization strategies that can be followed, with their expected performance advantages and area cost.

6.3 Future Work Towards a Complete ICN Router

Long term development should focus on completing the functionality required for a Libswift-PPSPP ICN Router. Two main development goals must be realized:

- A first step would be to complete the verification of Libswift data capabilities on the NetFPGA by integrating Merkle Hash Trees. This would completely eliminate the need to compute any SHA1 hashes in software.
- Further work can proceed to have data storage implemented in the software layer, with a lookup table of which data bins are stored locally. This step would allow the ICN Router to begin caching data passing through it, and also allow it to serve data requests which are encountered.

When the two stated objectives had been completed, the working prototype can be considered to be a working ICN router. However, in order to take advantage of the technological advancement, both in hardware and in development tools, it is our recommendation that future work be performed on the new NetFPGA 10G. This version would provide ample area to work with, as well as spare clock domains and avaiable DSP resources for optimizations.

Bibliography

[1] A. Bakker, R. Petrocco, M. Dale, J. Gerber, V. Grishchenko, D. Rabaioli, and J. Pouwelse, *Online video using bittorrent and html5 applied to wikipedia*, Peer-to-Peer Computing (P2P), 2010 IEEE Tenth International Conference on, aug. 2010, pp. 1 –2.

[2] A. Bakker, R. Petrocco, and V. Grishchenko, *Peer-to-peer streaming peer protocol (ppspp) draft-ietf-ppsp-peer-protocol-06*, (2013).

[3] Arno Bakker, *Merkle hash torrent extension - bep30*, 2009, Online; accessed 19/3/2013.

[4] Philippe Biondi, *Scapy*, 2013, Online; accessed 17/4/2013.

[5] BitTorrent, *Bittorrent*, 2012, Online; accessed 5/12/2012.

[6] G. Adam Covington, Glen Gibb, John W. Lockwood, and Nick Mckeown, *A packet generator on the netfpga platform*, Proceedings of the 2009 17th IEEE Symposium on Field Programmable Custom Computing Machines (Washington, DC, USA), FCCM '09, IEEE Computer Society, 2009, pp. 235–238.

[7] G.A. Covington, G. Gibb, J. Naous, J.W. Lockwood, and N. McKeown, *Encouraging reusable network hardware design*, Microelectronic Systems Education, 2009. MSE '09. IEEE International Conference on, July, pp. 29–32.

[8] David Crowe, *Sha1 calculator*, 2013, Online; accessed 13/8/2013.

[9] Jose Fonseca, *Gprof2dot - jrfonseca - convert profiling output to a dot graph.*, 2013, Online; accessed 19/2/2013.

[10] G. Gibb, J.W. Lockwood, J. Naous, P. Hartke, and N. McKeown, *Netfpga: An open platform for teaching how to build gigabit-rate network switches and routers*, Education, IEEE Transactions on **51** (Aug.), no. 3, 364–369.

[11] Susan L. Graham, Peter B. Kessler, and Marshall K. Mckusick, *Gprof: A call graph execution profiler*, SIGPLAN Not. **17** (1982), no. 6, 120–126.

[12] Victor Grishchenko, Flutra Osmani, Raul Jimenez, Johan Pouwelse, and Henk Sips, *On the design of a practical information-centric transport*, Tech. report, PDS Technical Report PDS-2011-006, 2011.

[13] Victor Grishchenko and Johan Pouwelse, *Binmaps: hybridizing bitmaps and binary trees*, 2009.

[14] James Hongyi Zeng, *Netfpga tutorial day 1*, Lecture at Tsinghua University, November 2010.

[15] Information Technology Laboratory, *Federal information processing standards: Secure hash standard (shs)*, National Institute of Standards and Technology (March 2012), no. 180-4.

[16] Roar Lien, Tim Grembowski, and Kris Gaj, *A 1 gbit/s partially unrolled architecture of hash functions sha-1 and sha-512*, Topics in Cryptology–CT-RSA 2004, Springer, 2004, pp. 324–338.

[17] Andrew Loewenstern, *Dht protocol*, 2008, Online; accessed 21/3/2013.

[18] Alfio Lombardo, Carla Panarello, Diego Reforgiato, Enrico Santagati, and Giovanni Schembra, *A module for packet hijacking in netfpga platform*, Digital System Design (DSD), 2011 14th Euromicro Conference on, IEEE, 2011, pp. 283–286.

[19] marsgod, *Secure hash algorithm (sha-160)*, 2004, Online; accessed 13/8/2013.

[20] Ralph C. Merkle, *A digital signature based on a conventional encryption function*, A Conference on the Theory and Applications of Cryptographic Techniques on Advances in Cryptology (London, UK, UK), CRYPTO '87, Springer-Verlag, 1988, pp. 369–378.

[21] Ralph Charles Merkle, *Secrecy, authentication, and public key systems.*, Ph.D. thesis, Stanford, CA, USA, 1979, AAI8001972.

[22] Jacob Jan-David Mol, Johan A Pouwelse, Michel Meulpolder, Dick HJ Epema, and Henk J Sips, *Give-to-get: free-riding resilient video-on-demand in p2p systems*, Electronic Imaging 2008, International Society for Optics and Photonics, 2008, pp. 681804–681804.

[23] NetFPGA, *Wiki - netfpga*, 2013, Online; accessed 13/8/2013.

[24] Pando Networks, *Pando content delivery cloud*, 2012, Online; accessed 5/12/2012.

[25] P2P Next, *Swarmplayer 3000*, 2012, Online; accessed 14/2/2013.

[26] ———, *Swarmplayer v2.0*, 2012, Online; accessed 14/2/2013.

[27] E. Nygren, R.K. Sitaraman, and J. Sun, *The akamai network: A platform for high-performance internet applications*, ACM SIGOPS Operating Systems Review **44** (2010), no. 3, 2–19.

[28] Delft University of Technology, *Tribler svn*, 2009, Online; accessed 13/2/2013.

[29] ———, *Swift: The multiparty transport protocol*, 2012, Online; accessed 5/12/2012.

[30] ———, *Tribler 5.9 release notes*, 2012, Online; accessed 13/2/2013.

[31] ———, *Tribler*, 2013, Online; accessed 13/2/2013.

[32] Flutra Osmani, Victor Grishchenko, Raul Jimenez, and Björn Knutsson, *Swift: the missing link between peer-to-peer and information-centric networks*, Proceedings of the First Workshop on P2P and Dependability (New York, NY, USA), P2P-Dep '12, ACM, 2012, pp. 4:1–4:6.

[33] Diego Perino and Matteo Varvello, *A reality check for content centric networking*, Proceedings of the ACM SIGCOMM workshop on Information-centric networking, ACM, 2011, pp. 44–49.

[34] R. Petrocco, J. Pouwelse, and D.H.J. Epema, *Performance analysis of the libswift p2p streaming protocol*, Peer-to-Peer Computing (P2P), 2012 IEEE 12th International Conference on, sept. 2012, pp. 103 –114.

[35] Thomas Schaap, *Performance assessment of libswift*, Master thesis, Delft University of Technology, 2012.

[36] R. Schollmeier, *A definition of peer-to-peer networking for the classification of peer-to-peer architectures and applications*, Peer-to-Peer Computing, 2001. Proceedings. First International Conference on, 2001, pp. 101–102.

[37] H. Schulze and K. Mochalski, *Internet study 2008/2009*, IPOQUE Report (2009).

[38] Waraxe IT Security, *Free online sha-1 hash calculator*, 2013, Online; accessed 13/8/2013.

[39] William Stallings, *Secure hash algorithm*, Cryptography and Network Security - Principles and Practices, Prentice Hall, 2003, pp. 357–364.

[40] The Wireshark Team, *Wireshark - go deep*, 2013, Online; accessed 17/4/2013.

[41] Akamai Technologies, *Akamai netsession interface*, 2012, Online; accessed 5/12/2012.

[42] ———, *Akamai: The leader in web application acceleration and performance management, streaming media services and content delivery*, 2012, Online; accessed 5/12/2012.

[43] Niels Zeilemaker, Mihai Capotă, Arno Bakker, and Johan Pouwelse, *Tribler: P2p media search and sharing*, Proceedings of the 19th ACM international conference on Multimedia (New York, NY, USA), MM '11, ACM, 2011, pp. 739–742.

[44] Hongyi Zeng, John W Lockwood, G Adam Covington, and Alexander Tudor, *Airfpga: A software defined radio platform based on netfpga*, NetFPGA Developers Workshop, 2009.