Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

David Salvisberg

An Adaptive Hardware/Software Interface for EmbedNet

Semester Thesis SA-2015-09
March 2015 to June 2015

Tutor: Dr. Markus Happe
Supervisor: Prof. Dr. Bernhard Plattner

Abstract

The static nature of the current Internet architecture increasingly proves to be a limiting factor for the always increasing diversity in use cases on end nodes. Dynamic protocol stacks aim to solve this shortcoming and hope to be a pillar of future networking architectures. EmbedNet is an adaptive end node implementation of such a dynamic protocol stack for FPGAs which utilizes their partial reconfigurability for dynamic hardware acceleration of functional blocks in the protocol stack.

EmbedNet unfortunately currently suffers from low throughput performance in its hardware/software interface. This thesis aims to improve this performance and to provide an interface that can adapt to the needs of the current application so that future work on EmbedNet may reach more promising results.

Acknowledgments

I'd like to thank my project advisor Dr. Markus Happe for his help and invaluable feedback during the course of this thesis. I would also like to thank Prof. Dr. Bernhard Plattner and the TIK Communication Systems Group for the opportunity to work on this semester thesis. It was a highly interesting project and it was a privilege to be working on it.

Contents

1 Introduction **9**

2 Background and Related Work **11**
2.1 ReconOS . . . 11
2.2 EmbedNet . . . 12

3 Methodology **13**

4 Design **15**
4.1 Hardware to Software (H2S) Interface . . . 15
4.1.1 Hardware part . . . 16
4.1.2 Software part . . . 17
4.2 Software to Hardware (S2H) Interface . . . 18
4.2.1 Software part . . . 18
4.2.2 Hardware part . . . 19

5 Evaluation **21**
5.1 Experimental Setup . . . 21
5.2 High Traffic Load . . . 22
5.3 Low Traffic Load . . . 23
5.4 Dynamic Traffic Load . . . 25
5.5 Resource Consumption on FPGA . . . 27

6 Conclusion and Future Work **29**
6.1 Conclusion . . . 29
6.2 Future Work . . . 29

A HowTo **31**
A.1 Hardware . . . 31
A.2 Software . . . 31

B Task Description **35**

C Declaration of Originality **41**

D Timetable **45**

List of Figures

1.1 Static vs Dynamic Protocol Stack . . . 9

2.1 EmbedNet FPGA design as in [2] . . . 12

3.1 Increasing throughput by buffering multiple packets . . . 13

4.1 H2S Design . . . 15
4.2 H2S Buffer Manager FSM . . . 16
4.3 S2H Design . . . 18
4.4 S2H Buffer Manager FSM . . . 19

5.1 Experimental Setup . . . 21
5.2 H2S Throughput Evaluation . . . 22
5.3 S2H Throughput Evaluation . . . 22
5.4 H2S Latency Evaluation . . . 23
5.5 S2H Latency Evaluation . . . 24
5.6 S2H Latency Evaluation . . . 24
5.7 Dynamic traffic throughput on original H2S . . . 25
5.8 Dynamic traffic throughput on new H2S . . . 25
5.9 Effect of small timeouts on dynamic traffic loads . . . 26
5.10 Effect of large timeouts on dynamic traffic loads . . . 26

Chapter 1

Introduction

The Internet as we know it today is being expanded every day by thousands of devices a sizable portion of which are mobile devices. These devices have to change their connection parameters several times a second in order to adapt to varying channel conditions. The main reason they have to do that is the ever increasing number of devices competing on the same channel for the same resources, the channel conditions change so drastically every second due to the interference of the other devices that they could not function properly if they did not adapt to them.

As the Internet of Things keeps expanding we can expect similar situations developing in all forms of communications. However, the architecture of the Internet as it stands today would make it difficult for communications to adapt to varying conditions, since the protocol stack of the Internet is static. While there are all kinds of extensions for the protocols we use today, they do not allow for the kind of on the spot multiparameter adaption we see with mobile devices talking to cell towers.

Another thing we see in wireless communications is the development and deployment of new and improved protocols on the physical layer at a much higher rate than we see it with the Internet architecture as whole which faces very little innovation, especially on the transport and network layers, the slow and painful introduction of IPv6 is one of the biggest changes we've seen on that front and the transition is still ongoing even though IPv6 was formalized well over a decade ago.

This is the main motivation behind exploring Dynamic Protocol Stacks as a clean-slate networking architecture. Dynamic Protocol Stacks would allow both for easier, on-the-fly adaptions of the stack to the conditions of the network as well as easier extendability of said stack by new functionality, such as a different method of encryption. In Figure 1.1 you can see that dynamic protocol stacks feature the same top and bottom layer i.e. the physical and MAC layer on the bottom and the application layer on the top as their static counterpart but the middle of the stack is being built dynamically with the available functional blocks. To build such a stack end nodes would talk to each other on a base protocol to negotiate an optimal protocol stack with the functional blocks available to both.

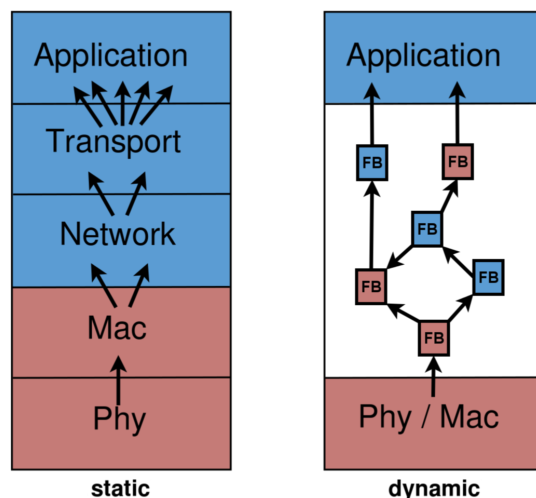


Figure 1.1: Static vs Dynamic Protocol Stack

One of the advantages of a Static Protocol Stack however is that most of the stack could be completely hardware accelerated, which is important for intermediary routers in the Internet since they will be able to handle a lot more traffic at a much lower cost while still making smart routing decisions with the information provided by higher level protocols. But hardware acceleration is also interesting for end nodes as we continue to shift our daily use of technology to mobile devices which rely on batteries. This is why the implementation explored in this thesis, namely EmbedNet, is based on FPGA technology which allows for hardware acceleration of arbitrary functional blocks which can be updated at any time by providing a new partial bitstream for the FPGA.

Unfortunately the current implementation is seeing low performance numbers which makes it difficult to determine if this could be a viable solution for the future. The design is being bottle-necked by the interface that allows for communication between the functional blocks residing in hardware and the functional blocks residing in software, this interface is necessary since the distribution of the functional blocks across software and hardware could change at any time, so the implementation has to be indifferent to whether any given functional block resides in hardware or software.

The goal of this thesis is to increase the performance of the design by improving the aforementioned interface, this should hopefully allow further research topics utilizing this design to gather more relevant data.

This report will continue with the following sections: First there will be an elaboration on the background of this work showcasing related work (Section 2), followed by a description of the methodology (Section 3). After that the design of the interface will be discussed in detail (Section 4). Next we present the results of our evaluation (Section 5) and finally a conclusion is drawn and an outlook on further research topics involving this design is given (Section 6).

Chapter 2

Background and Related Work

Dynamic Network Architectures have been a topic in research for several decades now, already in the early 1990s researchers were interested in developing dynamic network architectures to provide better reliability, scalability, security and extensibility. One of the earliest approaches was called Active Networking [1], in which users could inject custom code into the network. This approach has even seen some experiments on FPGAs. There won't be any further mention of these earlier approaches to dynamic networking in this section as most of them have not proven to be commercially successful, although they may have seen success in research. Active networking e.g. acts as the foundation for Software-defined networking [3]. More details on dynamic network architectures can be found in [2].

This chapter will instead focus on the research laying the foundation for this work, namely ReconOS which is a Linux based operating system enabling threads to dynamically reside in hardware or software on modern FPGAs and EmbedNet which is an FPGA based architecture leveraging ReconOS to implement Dynamic Protocol Stacks with the ability to have each functional block reside either in hardware or software at run time depending on the current network conditions.

2.1 ReconOS

ReconOS started out as a real time operating system based on eCos supporting both hard- and software threads i.e. allowing hardware threads to access the same operating system functions as the software threads. It has since been revised twice, first to allow for Linux and a common virtual address space shared by both hard- and software threads and then a major revision to streamline the whole design and make it more lightweight and modular. It has been covered by several academic publications extending its multithreading capabilities involving Hardware Threads. A lot of the work has been driven by innovations in FPGA technology allowing for partial reconfiguration of the hardware at run time, as such focus has shifted towards providing scheduling of hardware threads i.e. letting them go to sleep and be replaced by a different thread at run time.

Perhaps one of the most interesting of these works for software developers is *"Preemptive Hardware Multitasking in ReconOS"* [5] which allows hardware threads to be treated in the same way as software threads on the user level without restricting the scope of scheduling techniques. As such it turned ReconOS into a much more familiar environment for most software developers, allowing them to incorporate hardware accelerated threads that could be preempted and as such behave the same as software threads. Preemption is achieved by reading out partial regions of the hardware thread containing the state i.e. registers and local RAM with their current contents and storing it as part of the partial bitstream that will be used to put the thread back into hardware in the same state once it is to resumed.

Hardware threads need delegate software threads to utilize OS services but have their own interface for accessing main memory [4]. The blocks designed in this thesis are imple-

mented as such hardware threads and utilize the ReconOS library to access shared memory and to communicate with software threads.

2.2 EmbedNet

EmbedNet is an FPGA based architecture utilizing ReconOS allowing for adaptive dynamic protocol stacks, as in the functional blocks could be shuffled around between hardware and software as required to make optimal use of the given hardware resources. The architecture is covered in detail in [2].

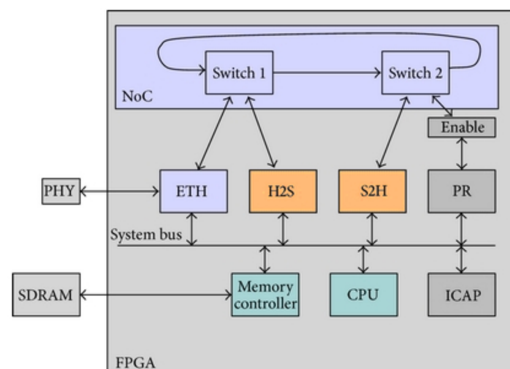


Figure 2.1: EmbedNet FPGA design as in [2]

The design of EmbedNet is depicted in Figure 2.1 and consists of a NoC[1] which in its minimal implementation uses two switches to let four hardware blocks communicate with each other, three of them are fixed, namely the Ethernet Block ETH, and the two blocks forming the interface between Hardware and Software, the Hardware to Software (H2S) and the Software to Hardware (S2H) block. The fourth block is reconfigurable and as such can be assigned to any FB[2]. The ICAP block handles the execution of the preemption of the dynamic FB, which involves reading out the current thread's state on preemption and reconfiguring the hardware region on resumption. Incoming packets will enter through the physical interface into the ETH block which replaces the Ethernet header with the NoC header to send the packet to the first FB in the chain via the Noc, this FB will in turn pass it on to the next FB. In each step the packet might leave the hardware through the H2S interface or enter it again through the S2H interface. Outgoing packets will enter the ETH block through their corresponding FB chain through the NoC eventually and be outfitted with the appropriate Ethernet header.

All of this showcases a minimal implementation of the architecture featuring one of each required block. The implementation could be scaled up to more FBs in hardware by increasing the number of switches in the NoC.

Keller et al. [2] concluded that their approach to networking can improve the performance of communications with regards to several aspects, such as the required number of packets, packet loss and CPU usage.

There have been a few theses by fellow students implementing parts of EmbedNet, the most relevant of which is the thesis that developed the NoC and the H2S/S2H blocks[6] since our thesis aims to improve upon this original H2S/S2H design. Beyond that there have been theses that implemented FBs like a Huffman Compression FB and a CRR reliability FB [7], an AES encryption/decryption block [8], an Intrusion Prevention block [9] and finally a generic FB used to evaluate EmbedNet[10].

[1]Network on Chip
[2]functional block

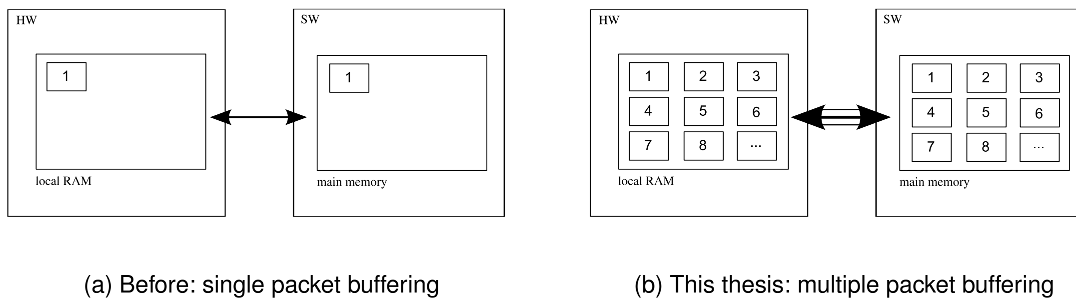
Chapter 3

Methodology

Since this work's focus is to improve upon an existing implementation we need a frame of reference to judge the new solution versus the old one. This section will elaborate on the design and performance parameters chosen to give that frame of reference.

The current implementation's Hardware/Software interface is bottle-necked by its low throughput caused by invoking the overhead of moving packets between a hardware core and the main memory and hardware/software synchronization for every single packet. As such buffering multiple packets and moving them all at once is a good way to distribute that overhead and increase throughput as a whole at a latency penalty for packets entering the buffer first.

Figure 3.1 visualizes this concept of making better use of the memory bandwidth, to the left you see the old approach which moves one packet at a time and to the right the new approach which moves many packets.



(a) Before: single packet buffering

(b) This thesis: multiple packet buffering

Figure 3.1: Increasing throughput by buffering multiple packets

Because of that throughput and latency are considered as good performance measures for this work. Beyond that there are a few parameters one could think of that would result in differing performance results. Immediately obvious is the size of the packets, which is an inherent variable in Ethernet protocols and can vary from 64 Bytes to 1500 Bytes in frame sizes and should significantly impact the throughput in the old implementation, but even in a buffered approach we can expect some variance due to fragmentation i.e. some packet sizes might get closer to the hard limit boundary in memory where the hardware has to flush the buffer.

Another variable that comes to mind, since we aim to buffer multiple packets, is the size of that buffer. We can expect to get big performance gains at first but with diminishing returns as we keep increasing the buffer size. Since increasing the buffer will increase the maximum latency as well as the occupied hardware area we want to find a good trade off here.

To combat starvation due to semi-full buffers while the network sees low traffic we also want to introduce a timeout into the design. Finding a good value for this timeout is important, so we will have to observe the latency behavior of the design at high traffic loads as well as low and dynamic traffic loads.

To summarize: we want to measure latency and throughput figures for...

- multiple packet sizes
- multiple buffer sizes
- multiple timeout durations

at high, low and dynamic traffic loads and draw conclusions based on these figures to find a good set of design parameters that improve the throughput significantly while invoking minimal penalties in latency.

The hardware/software interface should also be adaptive so that these parameters can be tweaked during synthesis or even at run time depending on the needs of the application. Low latency applications might prefer to never buffer multiple packets or only few to keep the latency figures as low as possible, whereas other applications might only care about raw throughput.

Chapter 4

Design

The design of the hardware/software interface for EmbedNet is split into two logical units based on the direction of the communication. There is a unit handling communications from hardware FBs[1] to software FBs called H2S and a unit handling the communications in the opposite direction called S2H.

Each unit consists of two parts, one of which is implemented in hardware and the other in software, since it forms an interface between hardware and software. In the following sections we will look at the design of each unit and both their hardware and software parts.

4.1 Hardware to Software (H2S) Interface

Figure 4.1 shows the design of the H2S interface as a whole, the thin arrows between hardware and software signify synchronization signal issued through the ReconOS mbox[2] library. The big arrow signifies the packet path.

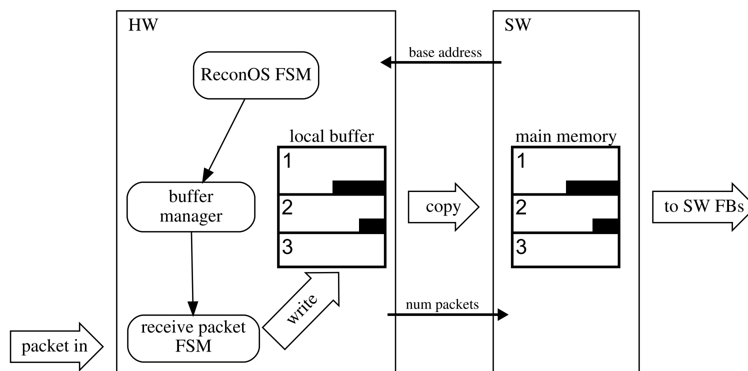


Figure 4.1: H2S Design

The hardware part contains three FSMs[3] and a local RAM buffer:

- `ReconOS FSM` is responsible for synchronization (arrows between HW and SW) with the software part of the unit and initiating the copy of the local buffer to a selected region in main memory.
- `buffer manager` manages the local buffer to ensure the unit will stop receiving packets and flush the buffer to main memory as soon as the timeout is reached or the buffer is too full to fit another packet of maximum size.
- `receive packet FSM` is responsible for receiving the packets from the NoC.

[1] functional block
[2] message box to pass information between hardware and software thread
[3] finite state machine

Furthermore we have chosen to store all our packets word aligned[4] to simplify reading from and writing to these buffers. Although this means we won't make optimal use of the space, we lose at most 5% of the space to fragmentation with small packets and much less than that in the average case.

The software part is simple by comparison. As an initial setup it allocates a buffer in main memory large enough to fit the the hardware buffer and sets up the hardware thread. After this initial setup, for each iteration it will pass the base address of the buffer in main memory to the hardware and then read out the buffer after receiving the number of packets that were copied from the hardware buffer.

4.1.1 Hardware part

Figure 4.2 displays the `buffer manager` which is the central unit of the hardware design managing most of the new functionality added over the old one.

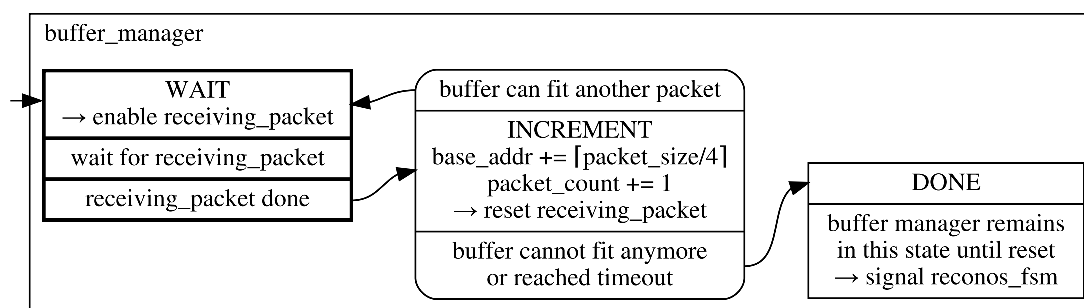


Figure 4.2: H2S Buffer Manager FSM

The `buffer manager` keeps track of how many packets there currently are in the buffer and how much space they are taking up, so it can decide whether it can store another packet. All of this happens in just two states, first in `WAIT` it will enable the receive packet FSM and wait until it is done receiving a packet, after that in `INCREMENT` it will increase the base address by the word size of the packet written last and increase the packet count and reset the `receive packet FSM`. If at this point the buffer cannot store another packet of maximum size or the timeout for this iteration has been reached the FSM will enter its `DONE` state in which it will remain until the `ReconOS FSM` initiates the next iteration.

To add the timeout functionality a timer has been added that gets started as soon as the interface is ready to accept the first packet of the current iteration and will stop at the timeout given in cycle counts and starts emitting a timeout signal until it gets reset. The timeout signal gets checked by the `receive packet FSM` during its idle time as it is waiting for the next packet on the interface as well as on each iteration of the `buffer manager`.

In order for the interface to be adaptive at run time the `ReconOS FSM` will first receive two initializing signals through the delegate thread to configure buffer sizes smaller than the synthesized one and a timeout given in milliseconds, after that it will enter its main loop of receiving a base address in main memory, intializing the `buffer manager`, waiting for it to conclude, copying the local buffer to main memory using the local base address from the `buffer manager` as an upper memory boundary, and sending out the number of packets copied until the thread gets terminated.

The original design consisted of only a `ReconOS FSM` and a `receive packet FSM`. Since the `ReconOS FSM` didn't really care how many packets it was transferring or in fact that it was a packet at all, since it is just copying a memory segment, it seemed apparent to insert the `buffer manager` as a third state machine between the two. This way a good portion of the original design could be reused with small modifications.

[4]a word is 32 bits on the microblaze architecture [11]

The changes to the `ReconOS FSM` have been covered in the paragraphs above, as for the `receive packet FSM`: In its original design it would always write the packet to the start of the local buffer, this was changed to work off the local base address provided by the buffer manager.

4.1.2 Software part

After setting up the delegate thread for the H2S hardware block and configuring the buffer size and timeout duration to be used using ReconOS synchronization messages the software will allocate a buffer and pass its base address to the hardware block using the same method after which it will in turn wait on the message containing the amounts of packets written from the hardware.

Packets get read out of the buffer much like you may deserialize complex, variable size data structures. The first packet, and as such also its header is placed at the start of the buffer, the header contains the size of the payload, allowing the software to read out the rest of the packet and then advance to the next word boundary for the next header until the packet count received from the hardware has been reached, at which point the software will signal the hardware that the buffer is ready for the next wave of packets. This process could be optimized yet by preparing a second buffer in main memory in advance to receiving the first one, so the hardware could start receiving packets again immediately into the second buffer, while the software processes the packets in the first one.

From the original software only minor portions could be reused like the general setup of the delegate thread for the most part the software was written from scratch.

4.2 Software to Hardware (S2H) Interface

In Figure 4.3 we see the software/hardware design of the S2H interface. Since the packet path starts on the software side we will present the software part before moving on to the hardware part.

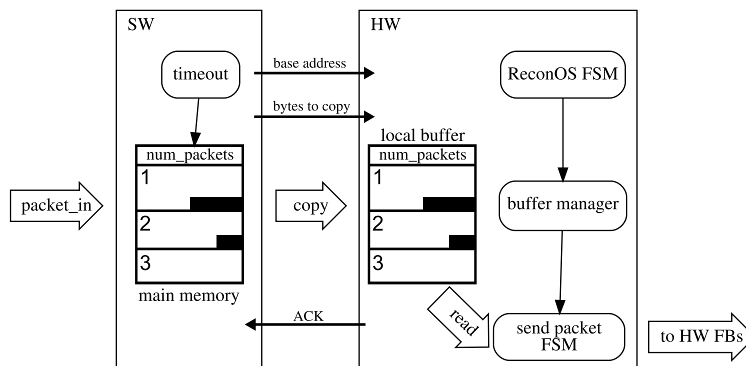


Figure 4.3: S2H Design

The software part is more involved this time, but still simple by comparison. The packets are once again stored word aligned, however the first word of the buffer is reserved for the number of packets stored in the buffer. This way we avoid introducing an additional expensive synchronization signal to pass that information from software to hardware.

Once again the software part will take care of all the initial setup work. In each iteration it will then go on to write packets as they come in starting after the reserved word at the head of the buffer until there is no more space in the buffer or the timeout for this iteration has been reached. At which point it will write the number of packets written into the buffer and send it off to the hardware by telling it how many bytes to copy starting at a given base address.

As with the H2S block, the hardware part once again consists of three FSMs and a local RAM buffer:

- `ReconOS FSM` synchronizes with the software and copies the buffer from main memory.
- `buffer manager` keeps track of which packet has to be send out next and how many packets are left.
- `send packet` is responsible for sending the packets out to the NoC.

4.2.1 Software part

The software part had to be changed significantly since it now has to manage a buffer instead of just dropping a packet in at the head of it and initiating the transfer. Additionally the software has to now respect a timeout to make sure it doesn't starve out the following nodes in the network.

Since frequent context switching turned out to introduce a significant performance hit on the software part on MicroBlaze we decided that each software FB should manage its own buffer so that context switches don't get forced with every single packet, which might only have taken a few tens of cycles to get generated, but rather with each buffer being forwarded.

Tests on an Intel i7 processor suggest however that frequent context switching may be viable on fast processors, since they will be able to generate packets a lot faster than they can be sent out, so the wait on the context switch would only result in waiting time you would have had to wait in the first place. This decision should definitely be revisited when the platform changes.

Performance tests have also only been performed with one packet generator and a single buffer as part of this thesis so it remains to be seen how much of a win this decision actually brings in more complex setups.

4.2.2 Hardware part

As displayed in Figure 4.4 the `buffer manager` works much in the same way as in the H2S block, but since it has to read out the number of packets from the local RAM it has to spend a few states on doing just that due to the two cycle delay of a read.

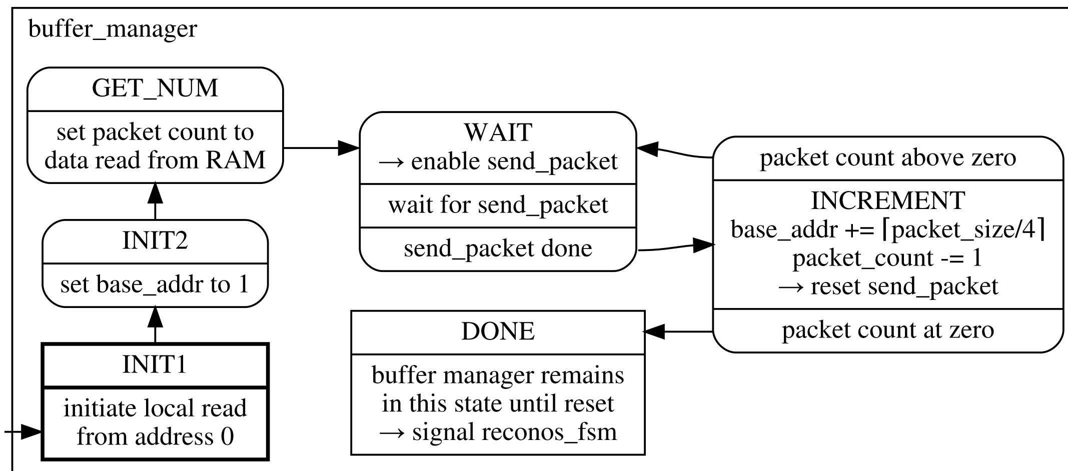


Figure 4.4: S2H Buffer Manager FSM

Apart from that after it has initialized itself with the packet count it performs the same two state loop of `WAIT` and `INCREMENT` as in the H2S version, except the exit condition is decreasing the packet count down to zero this time instead of checking for overflowing the buffer with another packet.

The original design once again already had a `ReconOS FSM` and a `send packet FSM`. The solution that was applied to H2S works here as well since send packet FSM will already inspect the packet header to determine the amount of bytes to be sent out. As such the only thing that needed to be modified after inserting the `buffer manager` between the two was the base address the `send packet FSM` was working off.

Chapter 5

Evaluation

Before presenting our results in the following subsections we will first discuss the experimental setup used to collect the data for our evaluation of the design. Section 5.2 will focus on raw throughput measurements for high traffic loads whereas Section 5.3 will focus on latency measurements to determine a suitable timeout value that promises to make good use of the buffer in most cases while still preventing exorbitant latencies with tiny bit rates. In Section 5.4 we will showcase how our design copes with dynamic traffic loads. Finally Section 5.5 will cover the resource usage on the FPGA of the new hardware designs over the old ones.

5.1 Experimental Setup

Figure 5.1 showcases the experimental setup. The FPGA used in this thesis is a Virtex-6 ML605 board. The board was programmed using the Xilinx ISE toolchain in its 14.7 version. Terminal output from the FPGA was gathered over the JTAG[1] connection cable using minicom on a 64 bit Linux machine. A dedicated network card rated at 1 Gbit/s was used in this machine to form the Ethernet connection with the FPGA board.

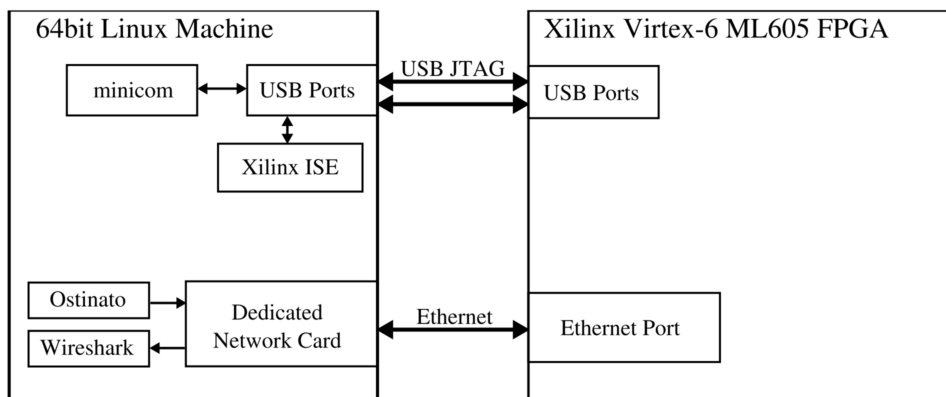


Figure 5.1: Experimental Setup

For evaluating the H2S block packets needed to be generated at a set bit rate by the machine connected via Ethernet to the FPGA, for these purposes a tool called Ostinato [12] was employed. To test whether the S2H block was sending packets out properly Wireshark [13] was used.

All time measurements have been performed by software executed on the FPGA using the system clock of 100 MHz. These time measurements have been used to calculate both throughput and latency figures directly on the board itself.

[1] Industry standard for testing and debugging integrated circuits

5.2 High Traffic Load

For evaluating the throughput of the H2S block we generated packets using Ostinato at 1 Gbit/s so we could measure the achieved throughput on the FPGA. We took measurements for the old design as well as power of two buffer sizes ranging from 2KB to 64KB, the timeout was disabled by setting it to a very large value.

In Figure 5.2 you see the results of these measurements for each of the packet sizes listed in the top left corner of the graph.

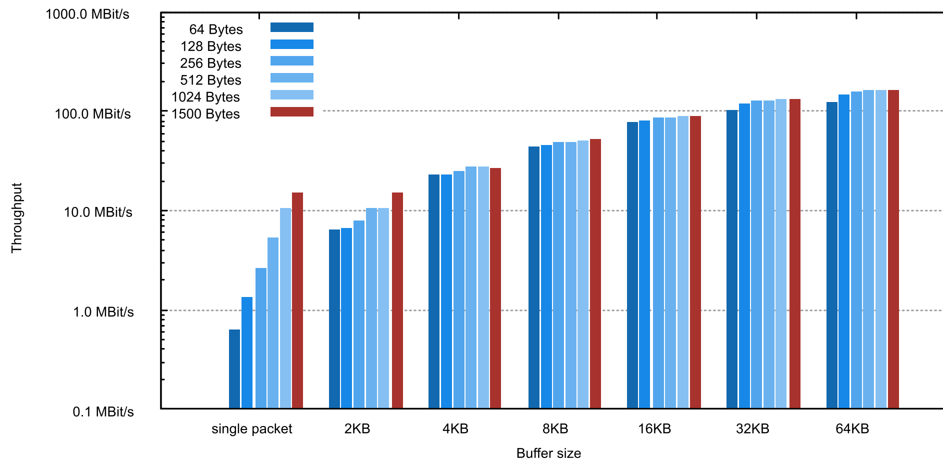


Figure 5.2: H2S Throughput Evaluation

These results are especially encouraging for small packets, as we can achieve a 10x speed up already at small buffer sizes and can go up to more than 100x with the largest buffer size that was tested. Even with large packets we can achieve 10x speed ups with a large enough buffer. Small packets seem to result in lower performance across the board which indicates that there is a part of the overhead per packet that cannot be reduced by buffering multiple of them, like e.g. initiating the transfer from the NoC for each of them or the inter-packet-delays introduced by the Ethernet device.

For the S2H block both generation of the packets and measurement of the throughput were performed by custom software on the FPGA itself. Wireshark was used solely to verify that the packets arrived. The same set of measurements have been performed for S2H for Figure 5.3.

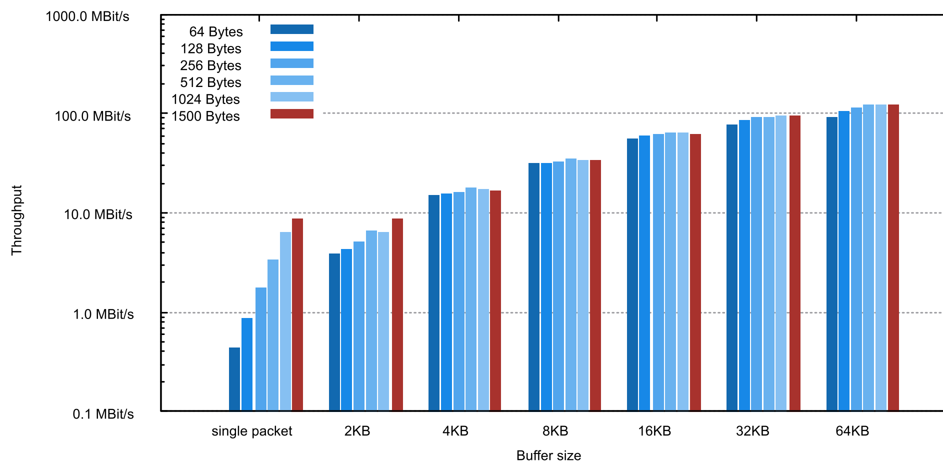


Figure 5.3: S2H Throughput Evaluation

S2H generally seems to achieve lower throughput than H2S but also started at an already lower base performance in the old design, so the relative improvements are similar for both

blocks across the old design and the different buffer sizes. E.g. from 15 MBit/s to 150 MBit/s for packets of 1500 Bytes in H2S versus from 9 MBit/s to 120 MBit/s in S2H both of which are improvements of one order of magnitude.

5.3 Low Traffic Load

The following measurements have once again been performed with the timeout disabled by setting it to a large value, since we want to evaluate just how much latency we're introducing for the first packet that enters the buffer.

For H2S the packets were once again generated by Ostinato at the desired transmission rate and packet size. Latency has been measured on the board as the time between receiving buffers which corresponds to the maximum amount of time a packet will spend sitting in the buffer. We took these measurements for the biggest buffer size we tested in Section 5.2 as well as the old design. In Figure 5.4 you see the results of these measurements for H2S. The orange line corresponds to the absolute minimum latency achievable at full throughput and single packet buffering.

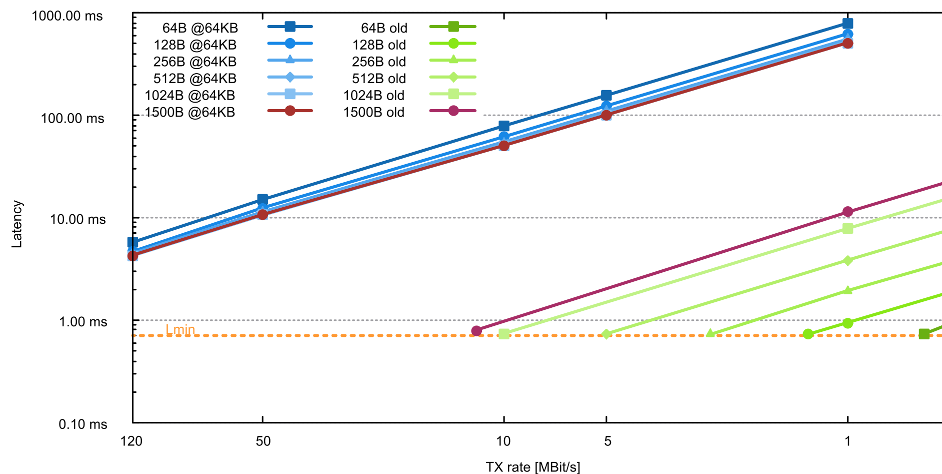


Figure 5.4: H2S Latency Evaluation

As expected with a buffer as large as 64KB you introduce a lot of latency to packets entering the buffer earliest at lower transmission rates, this reinforces the need of a timeout and the fact that this gain in transmission rate does not come for free.

You will notice that there's barely any spread in latencies in the new design across the different packet sizes whereas the old design has a clear spread between them, this is due to the fact that the buffer size is constant and as such the amount of bytes sent in one iteration is roughly constant as well in the new design, whereas in the old design the amount of bytes transmitted will vary per iteration.

It is also worth mentioning that packets entering the buffer last will possess the same kind of latencies that they do in the old design, so the measurements from the old design can also be seen as the corresponding L_{min} to the L_{max} that has been measured here. As such they showcase the entire spread of latencies packets in the buffer will experience at this transmission rate and buffer size. So for example packets of 1024 Bytes at 10 MBit/s will see latencies between 0.7 and 490 milliseconds if you don't introduce a timeout.

Figure 5.5 shows the same set of measurements for S2H, the latencies have been measured on the board as the time between being able to send out buffers.

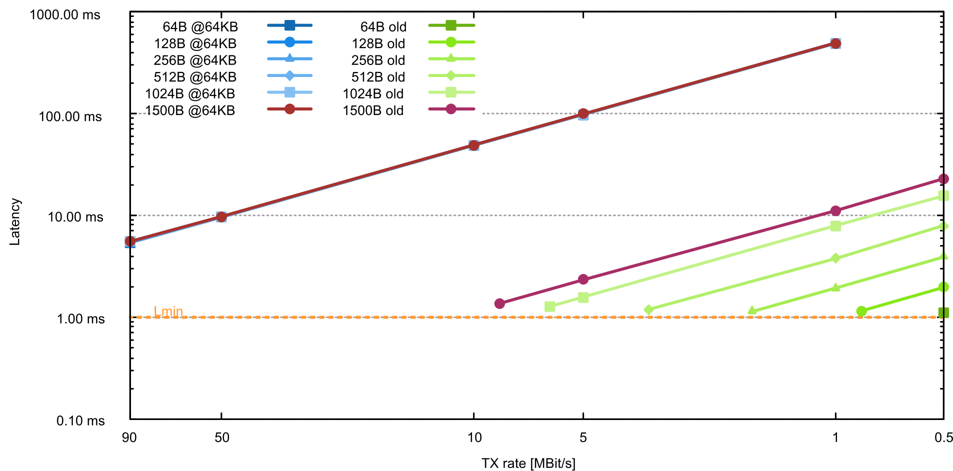


Figure 5.5: S2H Latency Evaluation

Once again we see very similar results with both interfaces, although the absolute minimum latency is larger for the S2H block.

Both the graphs in this section are very useful to determine the kind of timeout values you might want to use, since the the timeout will flatten the curve starting at that value until it hits the corresponding curve from the old design. Figure 5.6 displays this behavior for an example timeout of 10 milliseconds and a packet size of 1500 Bytes.

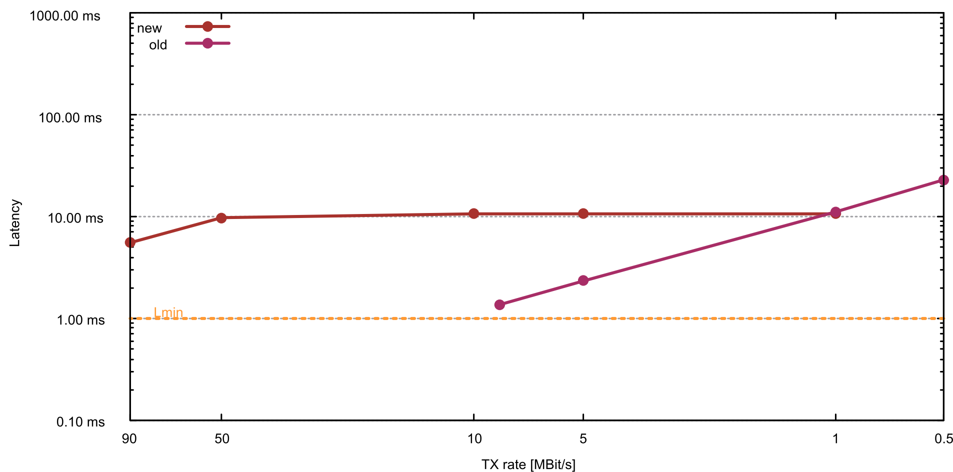


Figure 5.6: S2H Latency Evaluation

As soon as the two latency curves meet the old and new design will behave roughly identical, since the timeout will occur before a second packet can be received. One might also expect that there's more variation in latencies as you approach this boundary, since two packets will take almost twice as long to be received at transmission rates this low and an ongoing transmission will never be interrupted. The variation is however only this big if the packets themselves are being transmitted this slowly which is generally not the case. Usually the low transmission rates are achieved by having a bigger delay between the packets, while the packets themselves are still being transmitted at the maximum rate possible.

5.4 Dynamic Traffic Load

In this section we take a look at how our design copes with more dynamic traffic loads with a predetermined set of design parameters. For the traffic load we have chosen to superimpose two triangle waves with two different periods, phase and amplitude are the same for both. A packet size of 1024 bytes was chosen, since the old design can still reach reasonable transmission rates with it. The traffic was generated using a custom C program.

In Figure 5.7 you can see how the original design copes with this load.

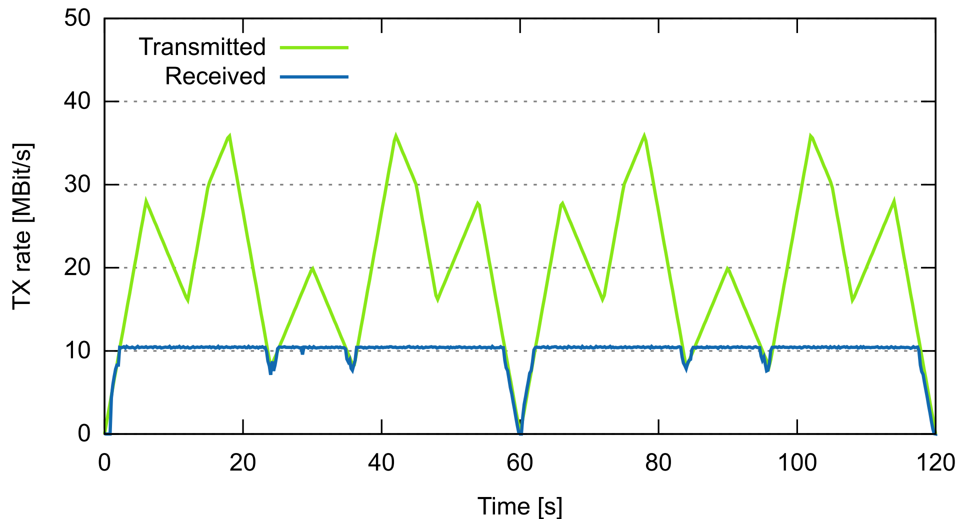


Figure 5.7: Dynamic traffic throughput on original H2S

Unsurprisingly the old design plateaus at around the 10 MBit/s mark whenever the incoming traffic goes beyond that speed, this limit could already be observed in Section 5.2.

Figure 5.8 displays how well our design copes with the same traffic load if it is configured with a buffer of 64KB and a timeout of 10 milliseconds:

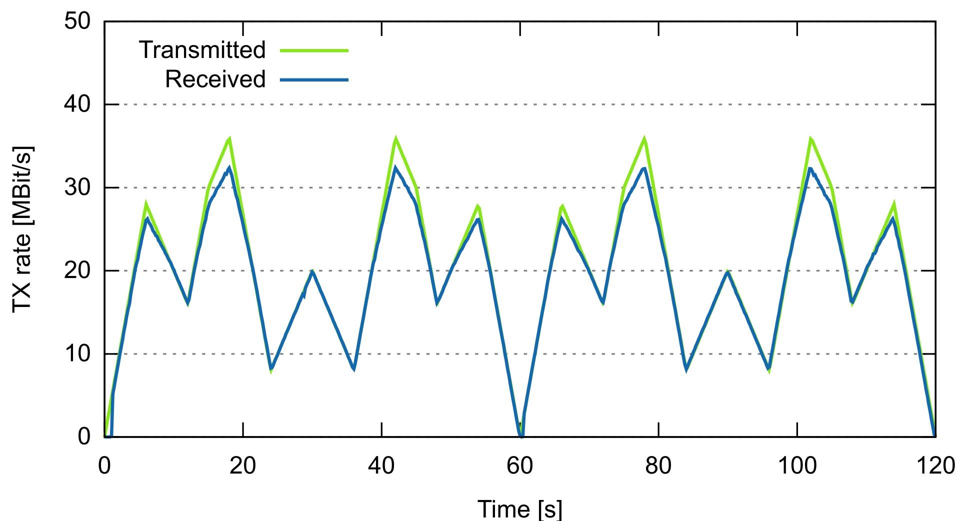


Figure 5.8: Dynamic traffic throughput on new H2S

As shown in Section 5.2: Our H2S interface with a buffer as large as this can easily reach the transmission rates required by this traffic load, however as you approach the transmission rates where the maximum latency is in the same range as the timeout our design can't quite follow the traffic's slope anymore.

Figure 5.9 reinforces this observation with a traffic load that fluctuates between 20 and 40 MBit/s and two different timeouts. Packet and buffer size are the same as in Figure 5.8.

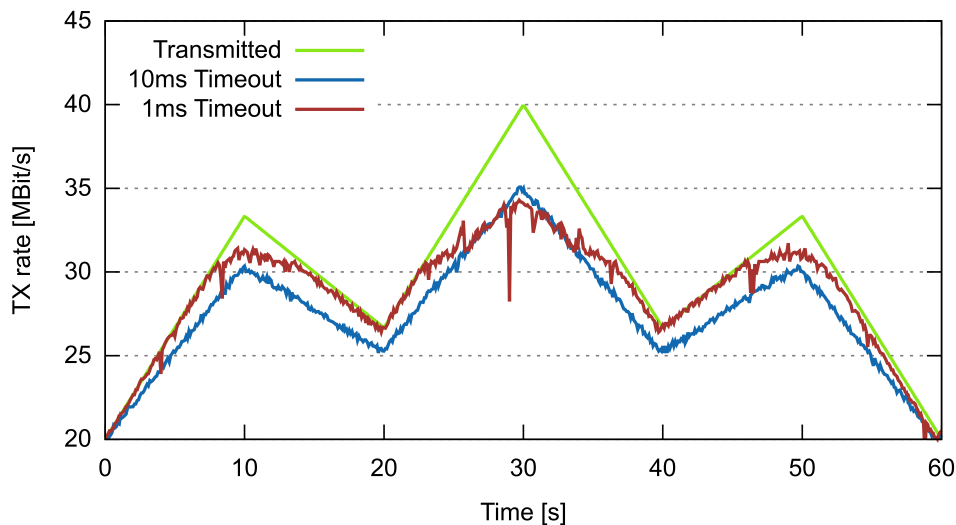


Figure 5.9: Effect of small timeouts on dynamic traffic loads

As you can see with a 10 millisecond timeout our design can't quite match the slope of the incoming traffic starting at around the 20 MBit/s mark. At a much lower timeout of 1 millisecond the interface can follow the slope much more closely but also reaches saturation at around 35 MBit/s.

The effects of large timeouts are invisible for the most part in these traffic traces since the design will easily achieve the transmission rates, the only visible artifact is a lower resolution of the trace at low transmission rates.

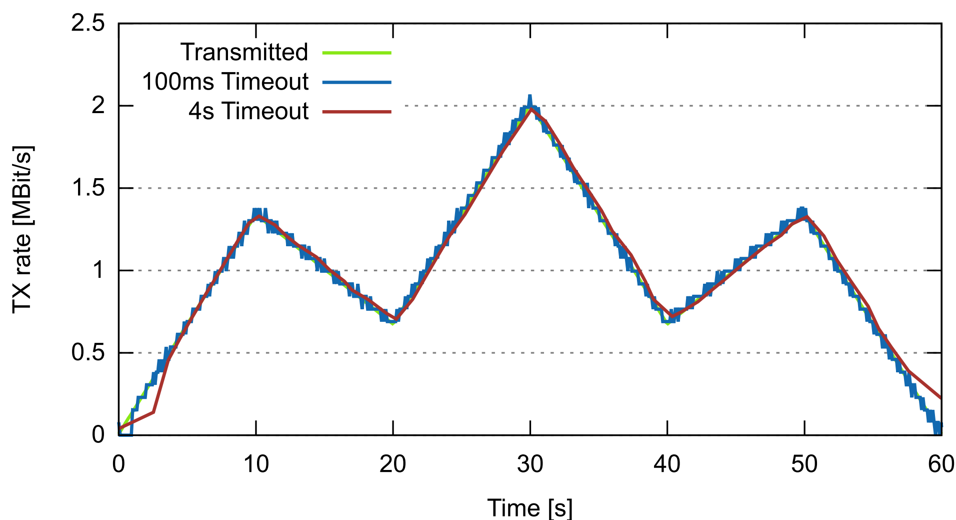


Figure 5.10: Effect of large timeouts on dynamic traffic loads

Figure 5.10 shows that our design is perfectly capable of following a low traffic load with a large timeout, the stepping in the function is due to the low transmission rates since increasing the packet rate by one will increase the transmission rate by 8 Kbit/s. Increasing the timeout further will essentially smooth out the curve and as such act as a low pass filter on the traffic dynamics at low transmission rates since it will average the incoming traffic over larger periods of time.

5.5 Resource Consumption on FPGA

Table 5.1 summarizes the resource consumption of the original design after synthesis.

	FlipFlops Used	LUTs[1] Used	BRAMs[2] Used
H2S	364	670	2
S2H	448	502	2

Table 5.1: FPGA Resource Usage of original design

In Table 5.2 on the other hand you will see the resource consumption of the new design with a 64KB buffer.

	Flip Flops Used	LUTs Used	BRAMs Used
H2S	523	782	16
S2H	484	588	16

Table 5.2: FPGA Resource Usage of new design with 64KB buffer

Both the Flip Flop and LUT counts have increased by amounts that can be regarded as negligible compared to the kind of speed ups we can achieve. The increase in the amount of BRAMs is the only figure one might be concerned about when trying to save FPGA resources, but it is also the one figure you can tweak prior to the design's synthesis.

As seen in Section 5.2 we can achieve speed ups of one or two orders of magnitudes with a buffer of 64KB and we take up just under one order of magnitude of additional resources.

[1] Block RAM
[2] Lookup table

Chapter 6

Conclusion and Future Work

6.1 Conclusion

This thesis aimed to improve the existing design of the Hardware/Software Interface of EmbedNet by increasing its maximum throughput to the point where it could no longer be considered a major bottleneck of the design. Since this increase in throughput is not free and comes at a latency tradeoff we also set out to make this new design adaptive so that the design parameters could easily be tweaked both offline as part of the synthesis and online as part of passing some additional parameters after setup of the delegate threads.

The evaluation of the design showed that we managed to thoroughly increase the maximum amount of throughput and also significantly reduced the negative impact of small packet sizes to the throughput. We achieved speed-ups between one and two orders of magnitudes over the old design at buffer sizes up to 64KB. Furthermore it confirmed our concerns about the amount of latency our design could introduce to some of the packets as such introducing a timeout as a limiting factor certainly seems to have paid off.

The improvements achieved as part of this thesis should certainly prove invaluable for future work on EmbedNet. As Roman Trüb concluded in his thesis [10], the Hardware/Software interface was the most limiting factor of EmbedNet next to the low computing power of the MicroBlaze softcore. Future work on EmbedNet will hopefully confirm that the performance of the softcore is now the only limiting element in latency insensitive applications.

6.2 Future Work

While the interface developed as part of this thesis is adaptive and could change its parameters based on the current traffic profile. The adaptivity we tested was mostly of offline nature i.e. we decided on parameters before we started receiving traffic, since the logic to make these decisions online has not been implemented yet. So an interesting project would be to develop a traffic analyzer that makes decisions on how to tweak the performance parameters of the interface based on the current conditions.

Furthermore we concentrated our efforts on traffic as a whole and did not yet take into account that there might be a variety of packets coming in at the same time with different performance requirements. We might for example expect traffic that consists of high traffic bursts that we want to push through as fast as possible but mixed in is a constant flow of a few highly latency critical packets that might now miss their requirements between the bursts since the interface was set to a high throughput profile. The NoC header features a flag to mark packets like these as latency critical, at the moment the interface just ignores that flag, it would be straightforward to add the ability to flush the buffer immediately whenever such a packet arrives in the interface, but since there was not enough time at the end of the project to ensure that this feature would function properly it was not yet added.

Appendix A

HowTo

This section will give a brief overview on how to use the hardware design and software shipped on the CD-R with this thesis.

A.1 Hardware

Configuration

Since the H2S and S2H interfaces are adaptive and their most interesting parameters can be changed at run time this section is only relevant if you want to perform tests beyond the default maximum buffer size of 64KB or want to take up less area on the FPGA with a smaller synthesized buffer.

The relevant VHDL sources can be found on the CD-R in:
`reconos/demos/protocol_graph_h2s_s2h/hw/edk/pcores/hwt_h2s_v1_00_b/`
and
`reconos/demos/protocol_graph_h2s_s2h/hw/edk/pcores/hwt_s2h_v1_00_b/`
respectively.

For both designs the constant `C_LOCAL_RAM_SIZE` determines the size of the local RAM buffer in words of 4 bytes.

Simulation

If you're interested in simulating the hardware design, testbenches have been provided in `reconos/demos/protocol_graph_h2s_s2h/hw/edk/simulation/` along with a wave configuration file for ISE.

A.2 Software

All of the source code for software executed on the FPGA can be found in `reconos/demos/protocol_graph_h2s_s2h/sw/` on the CD-R, the sources for self-written tools can be found in `tools/`

Compilation

For compilation you need to have the microblaze compiler toolchain in your PATH. After that it is a simple case of running make in the folder for the corresponding applications. Before that you will however also need to have the ReconOS libraries compiled via running make in `reconos/linux`.

Each application also comes with a debug folder which lets you compile the program for your current machine by running make inside it. Although these debug builds won't have the necessary hardware to interact with and as such won't provide meaningful measurements, they are nevertheless useful to debug the data and control flow of the program.

app_h2s

This application will receive packets sent to the H2S interface after configuring it with the specified buffer size and timeout and will print out the measured performance across the number of packets received.

Calling the application with `-h`, `--help` or `help` will print out information on the usage of the application:

```
Usage: ./app_h2s [buffer_size] [timeout] [num_packets]
        buffer_size:    buffer size in KB (default value 64)
                        (note:) 1 turns on single packet buffering.

        timeout:        timeout in ms (default value: 10)
        num_packets:    how many packets to receive (default value: 16384)
```

app_h2s_trace

This application will receive packets sent to the H2S interface after configuring it with the specified buffer size and timeout and will print out the measured throughput in Kbit/s for each time step specified across the duration specified.

Calling the application with `-h`, `--help` or `help` will print out information on the usage of the application:

```
Usage: ./app_h2s_trace [buffer_size] [timeout] [duration] [timestep]
        buffer_size:    buffer size in KB (default value 64)
                        (note:) 1 turns on single packet buffering.

        timeout:        timeout in ms (default value: 10)
        duration:       duration of trace in s (default value: 180)
        timestep:       timestep between measurements in ms (default value: 500)
```

app_s2h

This application will send packets using the S2H interface after configuring it with the specified buffer size and timeout and will print out the measured performance across the number of packets sent. It is worth mentioning that generally the throttling of the data rate won't be accurate if you're throttling close to the actual limit of the hardware.

Calling the application with `-h`, `--help` or `help` will print out information on the usage of the application:

```
Usage: ./app_s2h [buffer_size] [timeout] [packet_size] [data_rate] [num_packets]
        buffer_size:    buffer size in KB (default value 64)
                        (note:) 1 turns on single packet buffering.

        timeout:        timeout in ms (default value 10)
        packet_size:    packet size in Bytes (default value 64)
        data_rate:      data rate in KBit/s (default value: -1, unlimited)
        num_packets:    how many packets to send (default value 16384)
```

send_pkts_simple

This tool will send packets on the Ethernet interface specified in the constant `ETH_INTERFACE` prior to compilation. It will send two super positioned triangle waves with the same amplitudes but different periods both starting at the lower packet rate specified. It is based on a tool provided to me by Dr. Markus Happe that sends a single triangle wave.

Calling the application with `-h` will print out information on the usage of the application:

```
program usage
  -s    hash value (64 bit)
  -p    packet length (in bytes)
  -l    lower packet rate (pps)
  -u    upper packet rate (pps)
  -i    interval between lower and upper packet rate for triangle 1 (in sec.)
  -I    interval between lower and upper packet rate for triangle 2 (in sec.)
```

Appendix B

Task Description

See following page.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für
Technische Informatik und
Kommunikationsnetze

Semester Thesis

An Adaptive Hardware/Software Interface for EmbedNet

David Salvisberg

Advisor: Dr. Markus Happe, markus.happe@tik.ee.ethz.ch
Professor: Prof. Dr. Bernhard Plattner, plattner@tik.ee.ethz.ch

23 March 2015 - 22 June 2015

1 Introduction

Nowadays the diversity in networked devices, communication requirements, and network conditions vary heavily, which makes it difficult for a static set of protocols to provide the required functionality. Therefore, dynamic protocol stack (DPS) architectures are investigated in which protocol stacks can be built dynamically. In contrast to the static protocol stacks that are used in today's Internet architecture, the DPS architecture splits up the networking functionality into functional blocks, which can be dynamically linked with each other to form arbitrary protocol stacks. The execution environment called EmbedNet is an FPGA-based implementation of the dynamic protocol stack architecture that allows for a dynamic mapping of such functional blocks to either hardware or software.

The hardware/software interface is the major bottleneck of the EmbedNet platform which limits the packet throughput. The current version of the interface only supports to send single packets across the hardware/software boundary. This results in a low overall performance for packet processing. It is the goal of this thesis to improve the packet processing performance of EmbedNet by developing a new adaptive hardware/software interface.

2 Assignment

This assignment aims to outline the work to be conducted during this thesis. The assignment may need to be adapted over the course of the project.

2.1 Objectives

The goal of this thesis is to design and implement a new hardware/software interface for the EmbedNet platform, which can transfer multiple packets at a time. The expected outcome of this semester thesis is a new version of the hardware-to-software (H2S) and software-to-hardware (S2H) interfaces, which improve the packet processing performance of the EmbedNet platform.

2.2 Tasks

This section gives a brief overview of the tasks the student is expected to perform towards achieving the objective outlined above. The binding project plan will be derived over the course of the first three weeks depending on the knowledge and skills the student brings into the project.

2.2.1 Familiarization

- Xilinx Design Tools (XPS, SDK, Isim, ChipScope)
- EmbedNet architecture, ReconOS execution environment and corresponding APIs and libraries
- In collaboration with the advisor, derive a project plan for your semester project. Allow time for the design, implementation, evaluation, and documentation.

2.2.2 Architecture and hardware/software design

- Develop a hardware architecture for the new hardware-to-software (H2S) interface in hardware and a corresponding software architecture. The hardware and software blocks should buffer multiple packets at a time before they forward the packets to software. The packets should be transferred whenever the buffer becomes full (i.e. cannot store another packet).
- Develop a hardware architecture for the new software-to-hardware (S2H) interface in hardware and a corresponding software architecture. The hardware and software blocks should buffer multiple packets at a time before they forward the packets to hardware. Again, the packets should be transferred whenever the buffer becomes full (i.e. cannot store another packet).
- Optional: The packets should be automatically transferred across the hardware/software interface after a user-defined timeout to avoid starvation.
- Optional: The packets should be automatically transferred across the hardware/software interface whenever a time-critical packet arrives.

2.2.3 Implementation

- Determine an appropriate version control system and set it up for further use. You might consider using git and branch the official ReconOS git repository into your git repository.
- Implement the H2S and S2H hardware blocks (in VHDL).
- Implement the H2S and S2H software blocks (in C).
- Implement the generic hardware and software functional blocks on a Xilinx Virtex-6 ML605 board.

2.2.4 Validation

- Validate the correct operation of your implementation after each implementation step. Use for your evaluation different packet sizes (short, long, even or odd number of bytes, etc.).
- Quantify the maximum throughput of the H2S/S2H interfaces for selected packet sizes.
- Check the resilience of the implementation, including its configuration interface, to uneducated users.

2.2.5 Evaluation

- Do a performance evaluation of your implementation. This evaluation should include a stress test, in order to verify that your hardware thread does not introduce any instabilities into the overall system.
- Compare the performance of the new hardware/software interface to the old interface.

2.2.6 Documentation

- Provide appropriate source code documentation.
- Write a step-by-step how to that describes the compilation of your code, the loading of the code into the hardware and the execution of your code.
- Write a documentation about the design, implementation, validation and evaluation of your work.

3 Milestones

- Provide a project plan, which identifies the milestones.
- One intermediate presentation: Give a presentation of ten minutes to the professor and the advisor. In this presentation, the student presents major aspects of the ongoing work including results, obstacles, and remaining work.
- Final presentation of 15 minutes in the Communication Systems Group meeting, or, alternatively, via teleconference. The presentation should carefully introduce the setting and fundamental assumptions of the project. The main part should focus on the major results and conclusions from the work.
- Any software and hardware modules that is produced in the context of this thesis and its documentation needs to be delivered before conclusion of the thesis. This includes all source code and documentation. The source files for the final report and all data, scripts and tools developed to generate the figures of the report must be included. Preferred format for delivery is a CD-R.
- Final report: The final report must contain a summary, the assignment, the time schedule and a declaration of originality. Its structure should include the following sections: Introduction, Background/Related Work, Design/Methodology, Validation/Evaluation, Conclusion, and Future work. Related work must be referenced appropriately.

4 Organization

- Student and advisor hold a weekly meeting to discuss progress of work and next steps. The student should not hesitate to contact the advisor at any time. The common goal of the advisor and the student is to maximize the outcome of the project.
- The student is encouraged to write all reports in English; German is accepted as well.
- The core source code will be published under the GNU general public license.

5 References

[1] Ariane Keller, Daniel Borkmann, Stephan Neuhaus, and Markus Happe. Self-Awareness in Computer Networks. In International Journal of Reconfigurable Computing (IJRC), Article ID 692076, 2014, Hindawi.

[2] Andreas Agne, Markus Happe, Ariane Keller, Enno Lübbers, Bernhard Plattner, Marco Platzner, and Christian Plessl. ,,ReconOS – An Operating System Approach for Reconfigurable Computing". In IEEE Micro 34(1), Jan/Feb. 2014.

[3] Git Repository: `https://github.com/ReconOS/reconos/tree/v3.0_dev`

[4] Xilinx User Guide 360: Virtex-6 FPGA Configuration (v3.8) `http://www.xilinx.com/support/documentation/user_guides/ug360.pdf`

Webpages: `http://www.epics-project.eu` `http://www.reconos.de`

Appendix C

Declaration of Originality

See following page.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

An Adaptive Hardware/Software Interface for EmbedNet

Authored by (in block letters):
For papers written by groups the names of all authors are required.

Name(s):	First name(s):
Salvisberg	David

With my signature I confirm that
- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

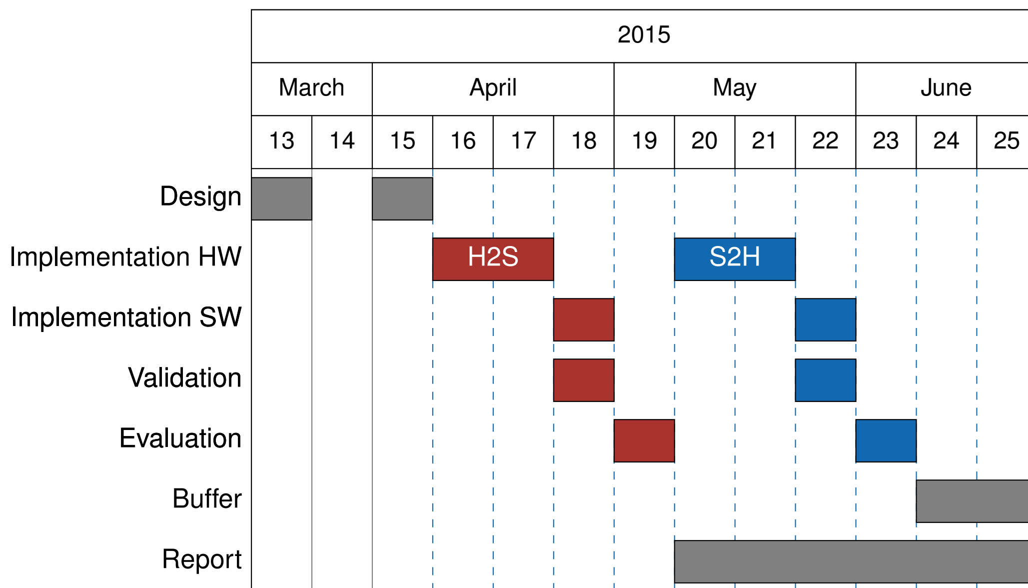
Place, date

Signature(s)

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.

Appendix D

Timetable



Bibliography

[1] D. L. Tennenhouse and D. J. Wetherall. Towards an active network architecture. *Computer Communication Review*, 26, 1996.

[2] A. Keller, D. Borkmann, S. Neuhaus, and M. Happe. Self-awareness in computer networks. *International Journal of Reconfigurable Computing (IJRC)*, Article ID 692076, 2014.

[3] N. Feamster, J. Rexford amd E. Zegura. The road to SDN: an intellectual history of programmable networks. *ACM SIGCOMM Computer Communication Review*, 44(2), Apr 2014.

[4] A. Agne, M. Happe, A. Keller, E. Lübbers, B. Plattner, M. Platzner, and C. Plessl. Reconos – an operating system approach for reconfigurable computing. *IEEE Micro*, 34(1), Jan/Feb 2014.

[5] M. Happe, A. Traber, and A. Keller. Preemptive hardware multitasking in reconos. *International Symposium on Applied Reconfigurable Computing (ARC)* p. 12, Apr 2015.

[6] R. Huber. A Dynamic Hardware Architecture for Future Networks. Master Thesis, ETH Zurich, Jun 2012.

[7] F. Deragisch. Network Protocols for Embedded Devices *with Dynamic Hardware/Software Mapping*. Master Thesis, ETH Zurich, May 2012.

[8] Y. Yang. Hardware Encryption for Embedded Systems. Semester Thesis, ETH Zurich, Feb 2013.

[9] S. Kronig. Intrusion prevention for flexible Protocol Stacks. Master Thesis, ETH Zurich, Sep 2013.

[10] R. Trüb. Generic Functional Blocks for FPGA-based Network Nodes. Semester Thesis, ETH Zurich, May 2015.

[11] Xilinx, Microblaze soft processor core, 2013. http://www.xilinx.com/tools/microblaze.htm

[12] Srivats P. , Ostinato traffic generator and analyzer. https://code.google.com/p/ostinato/

[13] The Wireshark Foundation, Network protocol analyzer. https://www.wireshark.org/