Kai-Hajo Husmann

Handling Spike Data in an Accelerated Neuromorphic System

Bachelor Thesis
Bachelorarbeit

Handling Spike Data in an Accelerated Neuromorphic System

This bachelor thesis has been carried out by Kai-Hajo Husmann under the supervision of

Prof. Dr. Karlheinz Meier
Kirchhoff Institute for Physics
&
Prof. Dr. Artur Andrzejak
Institute of Computer Science

at the
Ruprecht-Karls-Universität Heidelberg

Handling Spike Data in an Accelerated Neuromorphic System

The EsterProxy Suite

Kai-Hajo Husmann

December 17, 2012

Electronic Vision(s), Kirchhoff-Institut für Physik,
Ruprecht-Karls-Universität Heidelberg

For

Alexandra Tribout

&

our daughter

Zora Anne Laurence Tribout--Husmann

Handling Spike Data in an Accelerated Neuromorphic System

This bachelor thesis deals with the transport of data in a novel hybrid system, which is being developed as part of the BrainScaleS research project of the EU. This system is an innovative combination consisting of neuromorphic hardware and a conventional cluster. For optimal use of the facility an efficient transmission of configuration data and action potentials (spikes) between the control computer and the neuromorphic hardware is important. During experiment operation, a high throughput of configuration data is of great importance, because runtimes in the order of milliseconds are sought. So-called closed-loop experiments, in which the neuromorphic hardware is supplied with sensory data and in turn provides motor data, also make demands on the latency of single transmissions. Given that the neuromorphic part of the facility works with time constants that are shortened in comparison to the biological equivalent by a factor 10^3–10^5, low latencies are required. In this thesis, a software to efficiently transfer spike-data (from the control cluster to the neuromorphic hardware) was developed. Based on this implementation important characteristics (e.g., bandwidth and latency) have been determined, which allowed for evaluating the problems described. The maximal achieved throughput is $88.2 \pm 0.8\,\mathrm{MEv/s}$ (spikes). The minimal available latency is $371 \pm 44\,\mu\mathrm{s}$.

Handhabung von Spikedaten in einem beschleunigten neuromorphen System

Die vorliegende Bachelorarbeit behandelt den Datentransport in einem neuartigen hybriden System, das im Rahmen des BrainScaleS Forschungprojekts der EU entwickelt wird. Dieses System ist eine innovative Kombination bestehend aus neuromorpher Hardware und konventionellem Cluster. Für eine optimale Nutzung des Systems ist eine effiziente Übertragung von Konfigurationsdaten und Aktionspotentialen (Spikes) zwischen Kontrollrechner und neuromorpher Hardware wichtig. Im Experimentbetrieb ist ein hoher Durchsatz der Konfigurationsdaten von großer Bedeutung, da hier Laufzeiten im Bereich weniger Millisekunden angestrebt werden. Sogenannte *Closed-Loop*-Experimente, bei denen die neuromorphe Hardware mit sensorischen Daten versorgt wird und im Gegenzug motorische Daten zurückliefert, stellen allerdings auch Anforderungen an die Latenz einer einzelnen Übertragung. Da der neuromorphe Teil des Systems mit Zeitkonstanten arbeitet, die im Vergleich zum biologischen Pendant um einen Faktor 10^3–10^5 verkürzt sind, sind hier niedrige Übertragungslatenzen erforderlich. Im Rahmen dieser Arbeit wurde eine Software zur effizienten Übertragung von Spike-Daten (vom Kontrollcluster zur neuromorphen Hardware) entwickelt. Anhand dieser Implementierung konnten wichtige Kenngrößen (z.B. Bandbreite und Latenz) ermittelt werden, die die Evaluierung der beschriebenen Problematik ermöglicht haben. Der erreichte maximale Durchsatz liegt bei $88.2 \pm 0.8\,\mathrm{MEv/s}$ (Spikes). Die bestmögliche Latenz bei $371 \pm 44\,\mu\mathrm{s}$.

Contents

1. Introduction **1**
1.1. Motivation: CarverMead, BrainScaleS . . . 1
1.1.1. Hybrid Multiscale Facility . . . 1
HMF Neuromorphic Part . . . 2
HMF Conventional Part . . . 3
The Closed Loop Experiment . . . 4
1.2. Assignment . . . 5
1.2.1. EsterProxy Suite . . . 6

2. Design and Code Presentation **8**
2.1. ShamemIPC . . . 11
2.1.1. Usage . . . 12
2.2. TimeSpanRnd . . . 14
2.3. UserDummy . . . 15
2.4. Test Benches . . . 17
2.4.1. EsterProxyTest . . . 21
2.5. Test Results . . . 21
2.6. ARQ Drain . . . 22
2.7. The `EsterProxy` program . . . 24
2.7.1. Serialization . . . 25
2.7.2. `RCF` Servant . . . 27
2.7.3. SpikesQueue: `DoubleSidedMutexedQueue` . . . 29
2.7.4. The Packager . . . 30

3. Discussion and Results **31**
3.1. `RCF` in General . . . 32
3.2. `ShamemIPC` in General . . . 35
3.3. EsterProxy Evaluation . . . 38
3.3.1. Single Spike Transfer . . . 39
3.3.2. Container Serialization . . . 41
3.3.3. Minimal Latency . . . 43
3.3.4. Maximal Throughput . . . 45
3.3.5. TSR-Simulated Environment . . . 48

4. Conclusion - Outlook **52**
4.1. ShamemIPC Pointer-Access Method . . . 52
4.2. Real ARQ Drain . . . 53

4.3. Multiple Packagers – Multiple FPGAs . . . 53
4.4. Latency-Dependent Packaging . . . 54
4.5. `C++11-ify` . . . 54
4.6. Container Format . . . 55
4.7. `ShamemIPC` Initialization . . . 55
4.8. Sizeof Pitfall / Packed Attribute . . . 56
4.9. `ShamemIPC` Throughput . . . 56
4.10. Operating System Support . . . 57
4.11. Network Protocol . . . 57

Appendices **59**

A. Caipc/ Code Package **59**
A.1. Repository . . . 59
A.1.1. Caipc Directory Environment . . . 59
A.1.2. EsterProxy Directory . . . 60
A.2. RCF Calls . . . 60
A.3. Evaluation Data . . . 62

B. HMF Host-FPGA Communication **64**
B.1. Current State . . . 64
B.2. Current Work and Future Improvements . . . 65
B.3. ARQ . . . 65

C. Acronyms **66**
C.1. Technical Terms . . . 66
C.2. Thesis Projects . . . 67
C.3. External Libraries . . . 68
C.4. Superordinate Project . . . 68

D. List of Figures **69**

E. Bibliography **71**

1. Introduction

1.1. Motivation

Carver Mead In the late 1980s *Carver Mead* developed the concept of *neuromorphic engineering* based on analogue electronic circuits implemented on a Very-Large-Scale Integration (VLSI) system (*Mead and Mahowald*, 1988; *Mead*, 1989).

With his concept Carver Mead attempts to emulate neuro-biological models. Neurons and synapses are implemented *in-silico* and they act as real –though simplified– neurons or synapses. In contrast to software simulation, where the interaction of neurons and synapses is realized using complex and time-consuming integral calculus, analogue hardware has a major advantage:

> Analogue hardware is able to emulate as opposed to simulate neuro-biological models, as the aggregation of electrical currents works in hardware just as in the human brain.

It is the aspect of *emulation* which characterises *neuromorphic* systems.

Institute The Electronic Vision(s) Group is part of the Kirchhoff-Institute for Physics, founded in 1995, at the Ruprecht-Karls-University Heidelberg. The Vision(s) Group works on novel computing paradigms in the field of neuromorphic engineering. Currently the Vision(s) Group takes part in a BrainScaleS research project.
This project is called the Hybrid Multiscale Facility (HMF) (*BrainScaleS*, 2012).

1.1.1. Hybrid Multiscale Facility

The Hybrid Multiscale Facility (HMF) is an advanced implementation of the Carver Mead concept combined with software simulation. It is understood as the combination of neuromorphic hardware with conventional hard- and software leading to *Closed Loop* experiments in simulated software environments. In the following paragraphs the HMF Neuromorphic Part (NP) –the actual research–, the HMF Conventional Part (CP) –the necessarily supportive, though not less important part– as well as the Closed Loop Experiment (`CL-Experiment`) –as a potential "aim"– are outlined.
This bachelor thesis itself centres on the data streams within the CP as well as between the CP and the NP. The following describes the HMF in more detail and contains the rationale why it is important to evaluate these streams.

HMF Neuromorphic Part

We begin with a presentation of the HMF Neuromorphic Part (NP) outlining the state of affairs of this research task. Single chips have been developed and manufactured which implement 512 neuron circuits with 224 synapses each. However, the main system features much higher neuron and connection densities using Wafer-Scale Integration[1]. Connections between the repeating structures –i.e. units of 8 chips– are added in a post-production step to maximize inter-connectivity. This is referred to as the Wafer-Scale Neuromorphic System (*Schemmel et al.*, 2010; *Millner et al.*, 2010). The mentioned chips are called High Input Count Analog Neural Network (HICANN) and a single wafer integrates 384 of those. Therefore, in total –on a single wafer– we have

$$384 \times 512 = 196\,608 \text{ neurons}$$
$$384 \times 512 \times 224 = 44\,040\,192 \text{ synapses}$$

The underlying neuron model is an enhanced *Integrate-and-Fire* spiking neuron model which is called *Adaptive Exponential Integrate-and-Fire* (AdEx) (*Brette and Gerstner*, 2005). Compared to the *biological model* the hardware parameter ranges are optimized –in terms of area consumption and technology-dependent modifications– for the in-silico implementation. Thus, the hardware model parameters are scaled compared to the biological model description. In particular, the time constants are scaled down by a factor of 10^3–10^5 compared to the biological real-time. This operational speed-up is largely configurable by changing the model parameters (e.g., leak conductances or disabling parts of the neuron capacitance), but interdependencies within the model might add further constraints on the usable parameter range.

Communication Spike data has to be conveyed between the various parts of the NP. We can distinguish between: (a) on-chip or on-wafer (i.e. inter-chip using the post-processing connections), and (b) inter-wafer/host-wafer communication. These communication streams have been split into two layers. The lower layer handles on-chip and on-wafer communication. On this layer events themselves only encode the source neuron address; the routing to –possibly multiple– target neurons is determined by a programmable routing grid. The higher layer handles inter-wafer and off-wafer communication. On this layer spike data is mediated through 12 Printed Circuit Boards (PCBs) per wafer which are equipped with one Field Programmable Gate Array (FPGA) each. Through programming the FPGAs one can determine the routing in this layer. Figure 1.1 shows a schematic of the communication layers.

HMF Transmitter The handling of the host-wafer communication –on software side– was discussed in the *HMF Transmitter* internship report (*Husmann*, 2011). In this internship *I* prototyped and evaluated a circular buffer resident in shared memory.

[1] Both, multiple single-chip-based systems and the WSI-system are in active use for debugging purposes and neural network testing.

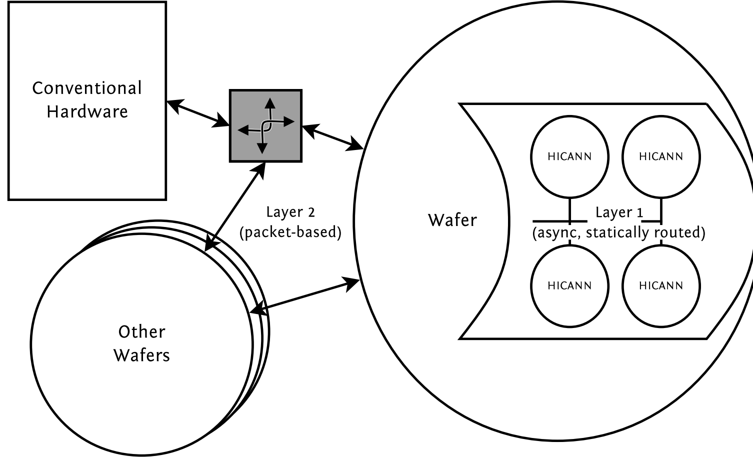


Figure 1.1.: Schematic of the wafer-scale system. On-wafer communication relies on asynchronous events which are statically routed; off-wafer events are wrapped into packets adding the event time. Communication involving the CP adds several more layers (i.e. custom transport layer on top of raw UDP).

This shared-memory buffer technique is expected to be used as primary communication method between the different software parts and the network interface controller, as it is –in interplay with the Remote Direct Memory Access (RDMA) technology– considered to be the fastest solution available.
In parts this bachelor thesis is a continuation of the *HMF Transmitter*.

HMF Conventional Part

The second part, which is called the HMF Conventional Part (CP) is made up of highly interconnected –Commercial Off-The-Shelf (COTS)– computers. They have been equipped with 10 GiB/s Ethernet cards (Intel® NetEffect™ NE020) and host services like the *MappingTool* (*Ehrlich et al.*, 2010; *Wendt et al.*, 2010; *Brüderle et al.*, 2011), software to control peripheral hardware (e.g., power control of the system, recording analogue membrane traces from oscilloscopes) as well as the interface to the simulated "real" world and the user.

The *MappingTool* translates biological experiment specifications using biological units and statistical parameter descriptions into wafer configuration data which in the end defines the neuron parameters and connections between neurons and synapses in the hardware. Thereby each neuron circuit (*denmem*) is configured by 23 individual analogue parameters and each synapse is configured by a 4 bit weight and a 4 bit (*listening*) address. Assuming analogue parameters to be 10 bit precise, i.e. ADC digital resolution, we have (per wafer):

$$196608 \text{ neurons} \times 23 \times 10\,\text{bit} \simeq 5.4\,\text{MiB}$$
$$44040192 \text{ synapses} \times 8\,\text{bit} \simeq 42.0\,\text{MiB}$$

As one can see the total configuration space is approximately 50 MiB.

Application In neuroscience parameter sweeps are a common research method (*Davison*, 2012; *Brette et al.*, 2007). A *parameter sweep* is understood to consist of a set of trials of repeated similar experiments but with in some detail highly varying parameters ("sweeping their range"). In general these parameters are rather independent and the number of trials grows exponentially with the number of parameters: ($\#trials = parRange_1 \times parRange_2 \times \ldots \times parRange_n$). Especially changes to network topology or statistical parameters might require reconfiguration of major hardware parts, and hence might require transfer of large amounts of configuration data. Additionally a single trial might run as short as 100 ms biological time – on the NP this scales down to the order of 10–100 µs. For such a kind of experiment we need to maximize trial *throughput*. And –as the trials itself are of short duration– that considerably depends on the time spend for the configuration.

Also one is interested to look into a running experiment. The available runtime data consists of spikes (resp. their fire time) and membrane traces which can be observed using Analog-to-Digital Converter (ADC) boards or oscilloscopes. The hardware is specified to support up to[2] 2 gigabytes/s. To encode a single event approximately 4 bytes are required (*UHEI and TUD*, 2011, cf. section 1.3.5). This yields[2] 500 megaevents/s. The timely availability (*latency*) of this data is especially important for so called *Closed Loop Experiments (`CL-Experiments`)*.

The Closed Loop Experiment

Vague Definition A Closed Loop Experiment (`CL-Experiment`) is understood as a connection of the NP with a simulated (*real world*) environment such that the NP triggers some action in that environment which in turn triggers back to the NP. In neuro-scientific terms, the simulated environment creates sensory input for the neuronal network which, in turn, creates a motor response. Amidst this loop stands the CP whose job it is to connect the interfaces of the NP with the simulated environment. This is a vague definition as the `CL-Experiment` is still in its infancy.

I write simulated environment here because the hardware is running about 10^4 times faster than the "real world". Therefore a *real* real world environment, presumably, would be to slow for the NP.

An Example In an example of a closed loop the job of the CP is to *translate* the NP output into movements of a *virtual* camera. The resulting camera output is then translated back into neural spikes to again trigger neural activity in the NP. This again leads to further movement of the camera, and so on.
However, interfacing other real-time neuromorphic hardware (e.g., silicon retinas (*Delbrück and Liu*, 2004)) is difficult due to the already mentioned acceleration factor which leads to mismatching time constants. A possible solution to that is a simplified simulated

[2] hardware time domain

environment which calculates response (*sensor data*) to input from the NP (i.e., *motor control*).

In this case, the software has to respond very fast. The slowest reasonable (in terms of parameter ranges) acceleration factor of the present hardware system is approx. 10^3. Thus, latencies of, e.g., 1 ms in the biological time domain are scaled down to 1 µs. Depending on the neuronal model, latencies up to 100 ms in biological real-time (100 µs in hardware domain) might be possible to handle. This is why the system has to be as responsive (low-latency communication) as possible.

Summing Up

For the Closed Loop Experiment (`CL-Experiment`) as well as for the *configuration* we need fast data transport between the NP, the CP and the *simulated environment*. This transfer has strong constraints considering throughput and latency.

This thesis centres on finding a *transfer methodology* which has the potential to satisfy these constraints.

1.2. Assignment

Currently, the HMF is in the debugging/launching phase (cf. Appendix B). Therefore the thesis focuses on standalone tests of present software and software that was developed during my work. The different parts are described below.

Shared Memory Continuing the internship mentioned in the motivation, one aim of this bachelor thesis was to reuse the internship's shared memory circular buffer code to construct a set of fully-fledged C++ classes offering a fast and productive library to easily create and access arbitrary (via templated buffer entries) shared memory circular buffer instances. These must be safe for concurrent access by different threads as well as by different processes (resp. programs). Furthermore the internal data must be packed and cleanly aligned to make it accessible by an RDMA-capable NIC (RDMA-NIC). This is the Shared Memory Inter-Process Communication Library (`ShamemIPC`).

Simulation Secondly it seemed useful to create a fast event dispatcher specialized on dispatching events at randomly generated intervals. The interval generator (e.g., regularly-spaced, Poisson distribution, ...) should be highly configurable. Such a tool was considered to be useful to rudimentarily simulate the NP for various tests. This is the Time Span Randomizer (`TimeSpanRnd`).

Application Last but not least the above should be used to create a tool to emulate the data transfer of a typical situation on the CP. That is a *user program* generating some arbitrary spike data and transferring these to the NP. For this, an intermediate `proxy` should be used which job it is to receive that user data and repack it into NP/hardware-formatted packets queued in a shared memory. Finally there should follow

a drain program which job it is to add the Automatic Repeat reQuest (ARQ) header and offer the data to the RDMA-NIC. This is one direction of a conceptual Closed Loop Experiment. Configuring the NP will relay on such a system as well. This assignment is the lion's share of this bachelor thesis and we call it the *`EsterProxy` Evaluation Suite* (*`EsterProxy` Suite*).

1.2.1. EsterProxy Suite

Ester in this case is the working name for the code base which takes care of:

> "**E**xperiment **st**art – **E**xperiment **r**un"

The aim of the *`EsterProxy` Evaluation Suite* (*`EsterProxy` Suite*) is to evaluate on what terms spikes can be transported from software to hardware. In this function we try to keep close to the requirements of the `CL-Experiment` to which the dependencies between latency and throughput are of considerable importance.

The *`EsterProxy` Suite* thereby gives a conceptual overview of the transferring software part of a general experiment. It does not consist of independent contrived test cases but it rather reflects the complete chain to realise the spike data transportation. In its basics the code can be used for productive use. The *`EsterProxy` Suite* has an integrated measurement setup and is configurable in a sense that one can enable/disable various features and transportations to evaluate latency, throughput, reliability, stability, as well as the interdependence of those.

As already mentioned it consists of three programs representing various sub-groups of the CP software part of an experiment (e.g. the *Closed Loop Experiment* or the *parameter sweep*)

1. `UserDummy` program (`UserDummy`): data source
2. `EsterProxy` program (`EsterProxy`): data handling
3. `ARQ_Drain` program (`ArqDrain`): data drain[3]

The `UserDummy` and the `EsterProxy` are expected to reside on different machines and to use a transportation based on TCP/IP. But the `EsterProxy` and the `ArqDrain` reside on the same machine and use the `ShamemIPC` to exchange data. The `UserDummy` as well as the `ArqDrain` are simulators only whereas the `EsterProxy` is a conceptual but nonetheless functional implementation handling the path from the `UserDummy` to the `ArqDrain`. The `UserDummy` simulates spike input by generating configurable spike data and the `ArqDrain` simulates the part of transferring data packets to the NP. The `EsterProxy` in the middle is the actual thing to be evaluated. It receives arbitrary spike data and has the job to pack this data into a shared memory in such a manner that a real `ArqDrain` implementation could use that shared memory data –without any understanding of its payload– to transfer that data to the NP. That means that the payload data packed into

[3]For further information on the Vision(s) Group's ARQ implementation see Appendix B.3.

the shared memory is understandable by the NP already and that further data is put into the shamem such that the `ArqDrain` has an easy job transferring this data to the NP. A real `ArqDrain` is supposed to be unaware of the internals –i.e. the expected data structures– of the NP. Of course in this evaluation the `ArqDrain` understands enough to do some evaluation.

2. Design and Code Presentation

As the Vision(s) Group low-level code base is written in the C++ Programming Language it was clear that the programming language to be used for this assignment had to be the same.[1] As a well-known and mature language C++ can unquestionably be called stable. Also C++ promises to potentially produce highly efficient and fast programs. But as a very modular and feature-rich language C++ is also highly dependent on the programmer's knowledge. If not used correctly the last two advantages can be quickly undermined and result in segmentation faults and other hard to track errors as well as undefined behaviour.

In the following paragraphs the used tools and libraries are outlined and then in the following sections an assortment of interesting code pieces and design decisions will be presented. The most important section in this chapter is the one about the `EsterProxy` (2.7) as it puts everything together whereas the other sections look into the elements which form the `EsterProxy`'s environment. For a first schematic overview of the ***EsterProxy* Suite** one may look at the figure 2.1. It shows the physical location of the *EsterProxy Suite*'s parts as well as the communications within.

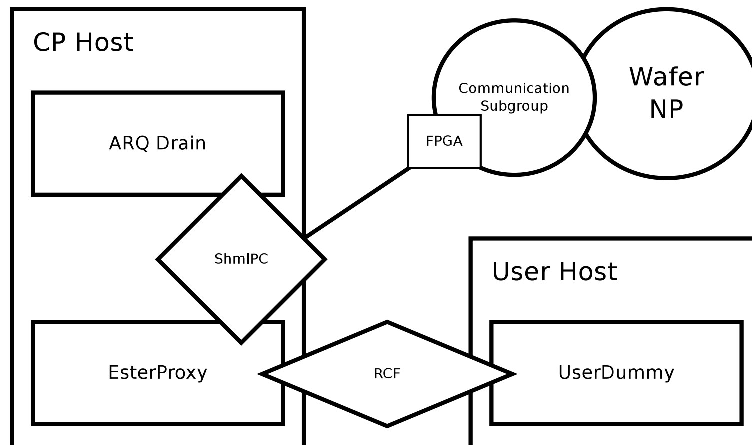


Figure 2.1.: Schematic of the *EsterProxy Evaluation Suite*

C++11 Having mostly programmed using the *Java* programming language the author went through a steep learning curve during this bachelor thesis. Additionally the C++ standard was subject to significant change. The code started out obeying the *C++03 Standard* and ended up using the recent *C++11 Standard (C++11)*. Not only by the author

[1] NB: The high-level (i.e. user interface) is written in Python.

is this new standard seen as a great improvement to C++. It integrates –amongst others– many of the ideas which had been originally introduced by the BOOST C++ Libraries (BOOST). Amongst the innovations are `std::chrono`, `std::unique_ptr`, `std::thread` and various improvements that are relevant for Template Meta-Programming (TMP).
For compilation the cutting-edge version of the GNU Compiler Collection (g++) was used:[2]

```
g++-4.7 (Ubuntu/Linaro 4.7.1-7ubuntu1) 4.7.1 20120814 (prerelease)
Copyright (C) 2012 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

At this point I personally want to thank the contributors of the website `http://en.cppreference.com` which helps migrating to the new C++11 Standard and which I heavily relied on during my work.

RCF For the transportation of spike data between the `UserDummy` and the `EsterProxy` program the Remote Call Framework by Delta V Software (RCF) was chosen. This again is primarily due to the fact that RCF is already in use in the code base. But there are other reasons: RCF is open source and free (GPLv2) for open source usage.[3] As there are such hard constraints in our special case it could be unavoidable to change parts of the remote call library itself; though that will hopefully not be necessary. Having this possibility is a considerable advantage. Nonetheless the RCF code base was accessed only for minor bugfixes. Secondly, it is portable, efficient and scalable as the company's website puts it.[4] One may also look at the RCF throughput evaluation presented in Discussion 3.1 on page 32. Problematic could be the fact that it seems to be primarily developed and tested under Windows. Minor syntactic bugs had to be fixed to achieve a functional build under Linux.

So far using RCF can be considered a good choice. Apart from the rather insufficient documentation –compared to standard Java software packages– everything worked quite well.

ShamemIPC For the data transfer from the `EsterProxy` to the `ArqDrain`, the `ShamemIPC` was developed and put into use. Why this was chosen has been outlined in the introduction and thoroughly discussed in the *HMF Transmitter* report (*Husmann*, 2011).

Event Library/TSR For the development of the `TimeSpanRnd` the Libev Event Library (`libev`) was chosen (*libev*, 2012a). That was simply due to the fact that it offers exactly what was needed and because this library is optimized for modern Unixes (like Linux or FreeBSD). Nowadays, as with the C++11 standard a thread library is available, the

[2]Since September 20, 2012, the most recent stable version is g++ 4.7.2 (`http://gcc.gnu.org/gcc-4.7`).
[3]RCF Licensing: `http://www.deltavsoft.com/buy.html`
[4]RCF Homepage: `http://www.deltavsoft.com/index.html`

use of `std::this_thread::sleep_for` resp. `std::this_thread::sleep_until` should be considered though it might not be flexible enough when e.g., using multiple timers and other event sources concurrently.

Code Location The code to this bachelor thesis is located in the `caipc`[5] repository. It can be found under the url `git@gitviz.kip.uni-heidelberg.de:caipc.git`. Whenever a file or directory location is named –and does not start with "/"– it can be found in this repository. Sole exception is the RCF Serialization Evaluation (`RCFSerEval`) code which was put directly under our `lib-rcf` repository as it fits there better. For the project to build it must be –as all components of *ester*– checked out into the `symap2ic` repository under its `components` directory. Appendix A.1 offers an overview of the `caipc` directory tree.

Building/wscript The *Ester* code base uses the Waf build tool (`waf`) for building. As the `caipc` code was integrated into the existing software framework, the build flow uses the same build tool: `waf`. `Waf` is a very modular and modern Python based build tool which integrates functionality for configuration, build and software deployment. It absolutely overpowers the archaic `make` build utility. In comparison with the GNU build system (*autotools*) the scripts tend to be more readable; at least they are written in pure Python. A Waf build script (`wscript`) that is located in a source directory can be called with the command `waf distclean configure build` which will do a complete and clean rebuild. For further information please refer to the `waf` website: `http://code.google.com/p/waf/`

For the `caipc` source the `wscript` offers some configuration options. Excerpt from the output of `waf -help` called in the `caipc` directory:

```
--set-caipc-deblvl=CAIPC_DEBLVL
                    0(release), 1(beta), 2(debug), 3(alpha), 4(debug0)
--test-ShamemIPC    Build ShamemIPC test objects and programs
--with-tsr          Build TimeSpanRnd objects
--test-TimeSpanRnd  Build TimeSpanRnd test programs
--with-proxy        Compile EsterProxy Evaluation Suite
--with-proxy-testbench
                    Enable compilation of UserDummies with testbench
                    includes
--test-rcf          Compile RCF testclasses
```

For a complete release compilation of the *EsterProxy Suite* one should call

```
CXX=g++-4.7 waf distclean configure
  --set-caipc-deblvl=0 --with-tsr --with-proxy-testbench build
```

If all required libraries and *Ester* sub-projects are available the compiled programs are afterwards available in the `build` directory. Links to the necessary libraries are created

[5] `caipc` that is a coquette concatenation of *Kai* (my name) and Inter-Process Communication (IPC).

in the `lib` folder. Otherwise the configure step will output information about missing dependencies.
Using the above command the compiler is basically called with these flags:

`-std=c++11` using the C++11 Standard

`-Wall` all warnings enabled ...

`-Wno-deprecated-declarations` apart from deprecated declarations warnings, because the `RCF` code at its recent stage has a considerable amount of them

`-O3` highest optimization level

2.1. ShamemIPC

The Shared Memory Inter-Process Communication Library (`ShamemIPC`) realizes a circular buffer in a shared-memory area (shamem) (shared by processes). It can be found in the directory `ShamemIPC`.

The circular buffer can be accessed by a writer and a reader. They are also called *heads* as they are moving around the shared memory similar to a hard disks read-and-write head. The difference is that here we have two separate heads, one for writing and one for reading, and also that not the medium (shamem) is moving but rather the heads. Think of these heads to float at the same height over the medium → one must never get past the other.

Each head has its own flags (position and status) which the other head may only read from and the trick is that position and status changes are constrained in such a way that the other head can handle an old value just like the real one (as long as the updates are in order). And a head can generally only move forward if its next position is smaller than the other head's visible –possibly outdated– position. For now the code works just fine without any memory barriers (on `x86`). It has been thoroughly tested. More on the theory and how memory reordering can be handled was discussed in the *HMF Transmitter* report (*Husmann*, 2011).

Classes The `ShamemIPC` consists of four mayor classes which represent an initializer as well as the two heads:

1. `eipcInitialiser`
2. `ShamemHead` and the two specializations:
   a) `ShamemWriterHead`
   b) `ShamemReaderHead`

2.1.1. Usage

Templated At first one wants to create template wrapper classes as well as an object to be held by the `ShamemIPC`. The interested reader can view an example of these in `EsterProxy/ipc_interface.d/`. The following two listings show an example for the initializer (`ShamemInitialiser`). For the heads (`ShmReader` and `ShmWriter`) it will look analogous.

```
/* initializer.h */
#pragma once
#include <ShamemIPC/eipcInitialiser.hpp>
#include "../ipc_xferTypes.h" // holds the buffered class (arq_shamem_t)

extern template class
    ester::ipc::eipcInitialiser<EsterProxy::arq_shamem_t>;

namespace EsterProxy { // we can put it into an arbitrary namespace
typedef ester::ipc::eipcInitialiser<EsterProxy::arq_shamem_t>
    ShmInitialiser;
}
```

```
/* initializer.cpp */
#include "initialiser.h"
#include <ShamemIPC/eipcInitialiser.cpp>

template class ester::ipc::eipcInitialiser<EsterProxy::arq_shamem_t>;
```

Initialization Now, before the `ShamemIPC` can be used a shamem has to be initialized which is handled by the `eipcInitialiser` which we wrapped in the `ShamemInitialiser typedef`.

Initialization creates an object in shared memory which (on most Linux flavours) can be seen in the `/dev/shm/` directory.

It will be called `ShamemIpc_` with its ID as suffix. The initializer supports RAII[6]-style initialization as well as lifetime-unbounded initialization. If the Resource-Aquisition-is-Initialization (RAII) initializer is used the object will be destroyed and removed from memory as soon as the destructor is called. But if the initialization and destruction are to be done by independent programs we can also use the unbounded initialization, in that the shared memory object must be explicitly removed by a correspondent call to the remover.

```
std::string id("MySharedMemory"); // an identifier of the shared memory
size_t bufsize = 42;
ShmInitialiser shminit(id, bufsize); // RAII style initialization
// ShmInitialiser::init(id, bufsize); // unbounded initialization
// ShmInitialiser::remove(id); // unbounded remover (can be called by any
    program)
```

[6]Resource-Aquisition-is-Initialization

The second step is to connect a writer and a reader to the `ShamemIPC` object. For this we call the factory methods `ShamemWriterHead<BufferT>::wconnect` and `ShamemReaderHead<BufferT>::rconnect` respectively (wrapped in `ShmWriter` and `ShmReader`). Then we start the returned objects.

```
1 // the writer thread/process
2 ShmWriter* writer = ShmWriter::wconnect(id); // shamem must be initialized
3 writer->startWriter(); // synchronizes with reader->startReader()
4
5 // the other thread/process
6 ShmReader* reader = ShmReader::rconnect(id);
7 reader->startReader(); // synchronizes with  writer->startWriter()
```

Run Phase During the run phase we have basically two methods to access the `ShamemIPC`. One accesses the shamem per `memcpy` whereas the other returns a pointer. The pointer access method has been added just shortly and should be further tested. Apart from not relying on a copy it has the advantage of the reader being able to change the underlying object. This will be useful when connected to the RDMA-NICs but more on that can be found in the Outlook 4.1 on page 52.

memcpy The writer's and reader's copy access methods are based upon `ShamemHead::accessShamem()`.

```
bool accessShamem(BufferT& object, const bool blocking=false,
                                   const bool flush=false);
```

Depending on the type of the head this method does either copy from the shamem into the `object` (reader) or vice versa. If `blocking` is `false` it will never enter any wait-loops but instead –if need be– omit the copy and return `false`. The boolean `flush` applies only to the writer. It moves the head to its next position thereby assuring that the reader can access the written element. A consecutive write does flush the prior write implicitly. For the reader a flush is more or less a no-op. It tries to move the head but will not enforce it. In particular if the writer has stopped the reader cannot move ahead after it has read the last element from the buffer as then its next position equals to the writers position (where it stopped). The `Shamem(Writer/Reader)Head` classes offer interface methods to do the access: `tryread/write`, `blocking_read/write` and `tryFlushingWrite`.

pointer The pointer access method actually consists of two single methods, one to acquire the pointer and the other to indicate that it can be released; for now these two methods have to be called in alternating order:

```
BufferT* getShamemPointer(bool blocking = false);
  // returns nullptr if accessShamem(obj, blocking, false) would return
      false
bool releaseShamemPointer(bool blocking = false);
  // if (blocking) returns equivalent of checkContinue()
```

One may switch between the *memcpy* and the *pointer* access methods at any time.

2.2. TimeSpanRnd

The Time Span Randomizer (`TimeSpanRnd`) is a convenient interface to the Libev Event Library (`libev`) which offers to disperse events among fine grained (randomized) time spans. It can be found in the directory `TimeSpanRnd`.

Basically it consists of the class `TimeSpanRandomizer`[7] and the interface `iTimeSpanEmitter`[8]. The `TimeSpanRandomizer` class can be used to start a loop of events. These events will be separated by time spans returned by an implementation of the `iTimeSpanEmitter` interface. An instance of `TimeSpanRandomizer` can either be polled for the next action or an event timer loop calling a callback method can be started:

```
std::unique_ptr<iTimeSpanEmitter> tse(new Any_TSE_Implementation());
TimeSpanRandomizer tester(tse.get());

bool getActionAvailable(); // returns true if an action is available, and
    in that case updates when the next action should occurr.
double startTimer(Callback_fptr tsrCallback, bool loop = false,
                  double _evMinDist = 0.0002);
```

In the following the second usage method, `startTimer()`, will be discussed.

Dispatching At first the event timer library is said to be fast – there are promising benchmarks on the `libev` website (*libev*, 2012b). But if the time spans are very short and the performed action takes too long, the real time might advance ahead of the next event. If that happens the events do still dispatch their linked action (the `performAction` callback function pointer) but the event timer is bypassed and instead we enter a specialized "catch-up" loop.

```
1  double now = getSeconds();                   // get the time
2  while ( nextAction < (now+evMinDist) ) {
3    // we remain in sched_yield loop if nextAction is near (evMinDist)
4    while ( nextAction <= now ) {              // next action already passed
5        updateAvailableAction();               // updates next action
6        if (!performAction(this)) return;      // handles next action
7    } // leaving this loop only when we're early
8
9    sched_yield();                             // yield to OS and then ..
10   now = getSeconds();                        // update now and check if we're
11                                              // close enough/early..
12 };
13 // now start the event timer.. (ev::timer)
14 w.set( nextAction - now, 0 );
15 w.start();
```

This loop will dispatch actions in the inner (faster) loop as long as we're definitely late. As soon as we have caught up we return to the outer `sched_yield()` loop which

[7]Defined in file `TimeSpanRnd/cTimeSpanRandomizer.hpp`.
[8]Defined in file `TimeSpanRnd/iTimeSpanEmitter.hpp`.

returns the control to the operating system. As long as we're not more than evMinDist early we dispatch events from here. Finally if there's enough distance to the next event we restart the event timer. The event timer callback will perform an action and then possibly enter the above listed loop again.

It is the programmers responsibility to offer a callback that can be handled within the average of a timespan as otherwise we will reach a point where the TimeSpanRnd will dispatch only late events.

Usage There are three implementations of the iTimeSpanEmitter interface available: static, Gaussian and burst. But writing a new time span emitter is easy. A contrived example:

```
1  #include "iTimeSpanEmitter.hpp"
2  class TSE_Example : public iTimeSpanEmitter {
3      virtual double getNextTimeSpan()
4          { return 1.0; } // one event per second!
5  };
```

To use the timer loop feature a callback pointer to perform the triggered actions needs to be passed along. It can look like this:

```
1  bool emitSpikeTask(TimeSpanRandomizer *tsr)
2      { return true; } // endless loop
3  // [...], now in main:
4  TimeSpanRandomizer::Callback_fptr taskFunction = &emitSpikeTask;
5  TSE_example tse;
6  TimeSpanRandomizer tester(&tse);
7  // ev::default_loop loop;
8
9  tester.startTimer(taskFunction, true); // sets performAction = taskFunction
10 // having passed true as second parameter the loop is integrated into the
11 // startTimer() call. Otherwise we need to call loop.loop(); here.
```

If the event loop shall stop the callback linked to performAction must simply return false. Therefore in the above example tester.startTimer(...) will block forever as emitSpikeTask(...) never returns false. The callback emitSpikeTask(...) will be called once every second.

2.3. UserDummy

The job of the UserDummy program (UserDummy)[9] is to configure the EsterProxy and to initiate the interconnection of the *EsterProxy Suite*. Therefore it should be started as third program after the EsterProxy and the ArqDrain. Secondly it performs the task of testing by calling test bench implementations. These test benches consist of code which is included at a specific position in the UserDummy. The test bench to include can be specified by the macro ESTERPROXY_TEST_BENCH="testBench.tb". If the test bench file

[9] The main function is located in the file EsterProxy/UserDummy.cpp.

is suffixed with `.tb` (testBench.tb) and placed into the directory `EsterProxy/test_bench.d/` the waf build script will automatically generate a `user-testBench` program[10].

A `UserDummy` program takes as an optional command line parameter the address of the `EsterProxy` it should connect to. The address must be specified in such a way that `RCF::TcpEndpoint()` will accept it. If that is not specified `localhost` is assumed.

Preparations Before the test bench can be started some preparations have to be done:

1. Connect to the proxy
2. Configure the connection
3. Check clock constraints
4. Establish `ArqDrain` connection

When the user program is started it at first tries to connect to its proxy. This will be repeated until a connection could be made. Then the connection will be configured, that is:

```
1 auto clientStub = client.getClientStub();
2 clientStub.getTransport().setMaxMessageLength(PROXY_MSG_SIZE_MAX);
3 clientStub.setRemoteCallTimeoutMs(PROXY_TIMEOUT_ms);
4 clientStub.setSerializationProtocol(RCF::Sp_SfBinary); // == default
```

The values for the maximum message length (242 MiB) and the remote time out (3 min) (`PROXY_MSG_SIZE_MAX` and `PROXY_TIMEOUT_ms`) are taken from the file `EsterProxy/common.h`. The remote time out is set to an unrealistic high value – at least for productive use –, that is because some tests (with rather worse) settings are in need of that. We want each test to finish gracefully no matter how bad its outcome is. The third setting explicitly specifies the archiver to be used, that is the `RCF` Serialization Framework (`RCF-SF`) in binary manner.

Clock constraints Assuring a small distance of the real-time clocks between the `UserDummy` and the `EsterProxy` is of crucial importance for latency measurements. In the *`EsterProxy` Suite* we give spikes their `spiketime` upon creation (unless prepared spike data is used). Latency is then calculated by the difference of the creation time and the time a spike passes by the point of measure. With high deviation between the clocks this value becomes fairly unusable. Therefore `common.h` specifies some constraints which must be met:

```
1 // checking test environment:
2 //    allowed clock divergence of EsterProxy and UserDummy
3 const double USER_CLOCK_INACCURACY_MAX // maximum clock INaccuracy
4                          = 10e-4;        // specified in seconds
5 const double USER_CLOCK_FAILED_BAD_MAX // max ratio of failed clock tests
6                          = 0.00;         // 0.0 (hard) .. 1.0 (don't care)
```

[10]If you configured with the flag `--with-proxy-testbench`.

This test goes as follows:

1. take local time stamp (`before = getSeconds();`)
2. set remote time stamp (`client.setTestTimeStamp(RCF::Twoway);`)
3. take local time stamp (`after = getSeconds();`)
4. fetch remote time stamp (`between = client.getTestTimeStamp(RCF::Twoway);`)

This test is repeated multiple times. If `between` is not in the interval (`before`, `after`) it is considered a "bad time". Additionally the divergence (distance to x) is calculated: $dist_x = abs(((before + after)/2) - between)$.

```
* indicates the returned remote time (between)
              |   [ * |   ]   |               // OK
              |   [   | * ]   |               // OK
 badTimes     | * [   |   ]   |   badTimes    // critical
              |   [   |   ]   | *             // BAD
          before      x      after            // x = (before+after)/2
```

In the sketch above the `|` denote the bad time boundaries: if a measured `between` lies outside these boundaries the clock inaccuracy is greater than the remote (two-way) call time. This is highly critical. It indicates that the involved machines should do a Network Time Protocol (NTP) update. The rectangle brackets denote the allowed average inaccuracy boundaries. The clock inaccuracy is calculated as follows:[11]

$$clockInAccuracy = mv(dist_xSet) + 2 \times sd_high(dist_xSet)$$

If the calculated inaccuracy is greater than `USER_CLOCK_INACCURACY_MAX` (0.1 ms) or $badTimes/testCount$ is greater than `USER_CLOCK_FAILED_BAD_MAX` (0.0) the program is aborted.

2.4. Test Benches

Test benches in this context are code pieces which are inserted into the `UserDummy` to perform various test tasks. The test benches differ in the spike creation, transfer methods and measurements taken. And generally in the setup (enabled features) of the `EsterProxy` side of the test.

Test benches rely on the class `EsterProxyTest`. This class offers three convenient methods –which encapsulate basic calls to the `EsterProxy`– to structure test cases. Calls to these methods are interlaced with direct calls to the `EsterProxy`. The following presents a simplified but explicit test bench:

[11] sd_high is special standard deviation in which only the variance of values higher than the mean is taken into account, for other values we assume a variance of 0, $sdHigh <= sd$.

```
1  /* vim: set filetype=cpp : */
2
3  // onerun.tbb
4  // This test is performed prior to any other test bench, it is generally
       included by the user dummy unless ESTERPROXY_TRIAL is specified.
5  // [2012-11-12 00:12:29] v2.0 New version for thesis include
6
7  { // we begin a test bench with opening a scope
8
9  // The client object has been defined by by the UserDummy and offers the
       access to the EsterProxy remote calls.
10 // Also available here is clockInAccuracy.
11
12 std::string testName("One Run To Rule Them All"); // Yes, I like The Lord
       of the Rings ;) - the book!
13 size_t spikesCount = 25 * HMF::FPGAPulsePacket::capacity();
```

Listing 2.1: onerun.tbb (intro)

```
15 // Latency measurements slow down the transfer so we can enable or disable
       them
16 bool measureProxyReceiveLatency = true;
17 bool measurePackagerSendLatency = true;
18 bool measureArqReceiveLatency   = true;
19
20 // Timeout Settings (in seconds)
21 double packagerTimeout  = std::numeric_limits<double>::infinity(); // never
22 double receiveTimeout   = 3e-3;
23
24 // Transfer settings
25 bool enableDropping = false;
26 bool onewayCalls    = true;
27 //size_t batchSize    = 16 * 1024 * 1024; // bytes
```

Listing 2.2: onerun.tbb (settings)

```
29 // Instantiate a basic test object (prepares the proxy)
30 EsterProxyTest test(client, testName, measureProxyReceiveLatency,
       spikesCount);
31
32 // Drop late data before anything else is done with it
33 if (enableDropping) client.enableReceiveDropping(receiveTimeout);
34
35 // Enable packaging of ARQ ready packets
36 client.enablePackaging(measurePackagerSendLatency, packagerTimeout);
37
38 // Enable transfer of constructed packets to the ARQ_Drain
39 //+ this must be polled until ARQ_Drain is ready
40 while ( ! client.enableArqXfer(measureArqReceiveLatency, enableDropping) )
41     std::this_thread::sleep_for(std::chrono::milliseconds(150));
42
43 double now = 0;
44 fpga_dummy_event_t tmp;
45 tmp.label = 42; // The Hitchhiker's Guide to the Galaxy is not bad either.
46 tmp.spiketime = 0;
47
48 // Spike creation specialization
49 double expectedDuration = 10;
50 double maxDuration = expectedDuration * 1.1; // 10% plus allowed
51 double avgSpikeDist = expectedDuration / test.spikes;
52 double nextSpikeTime = 0;
```

Listing 2.3: onerun.tbb (config)

```
54 // ------ TEST BEGIN ------------------------------------->
55 double pzOffset = test.beginTest(onewayCalls /*, batchSize*/);
56
57 // Now transfer the spikes as you wish
58 client.swallow(tmp);
59 for (int i = 1; i < test.spikes; i++) {
60
61     nextSpikeTime += avgSpikeDist;
62     do {
63         now = MicroCounter::getSeconds(pzOffset);
64     } while ( now < nextSpikeTime );
65
66     // calculate correspondant FPGA clock time
67     tmp.spiketime = now * CYCLES_PER_SECOND; //defined in common.h
68
69     client.swallow(tmp); // transfer the spike!
70 }
71
72 test.endTest();
73 // <------ TEST END ---------------------------------------
```

Listing 2.4: onerun.tbb (test)

```
75  // Finally we can access the test duration results
76  assert( test.durationClient <= test.durationProxy + clockInAccuracy );
77  assert( test.durationProxy  <= test.durationArq + clockInAccuracy );
78  assert( test.durationArq    <= maxDuration );
79
80  // And lots of other result data, @see class TestResult.
81  // Some of which is not available under all circumstances!
82  if (!enableDropping) {
83      assert( test.res.arqLastSpikeAvailable );
84      assert( test.res.arqLastSpike.spiketime == tmp.spiketime );
85      assert( test.res.arqLastSpike.label == tmp.label );
86  }
87
88  std::cout << " * [UD] Functionality test passed (" << test.name << "): " <<
        test.durationArq << " sec.\n" << std::endl;
89
90  } // finally close the test bench scope
```

Listing 2.5: onerun.tbb (finish)

This test bench[12] will be performed prior to any other test bench passed to the `UserDummy` at compile time. What the above code does should be clear enough with its comments. A few words to the class `EsterProxyTest` can be found at the end of this section. All available remote calls are listed in the Appendix A.2.

One should note that a potential `NDEBUG` flag will be disabled for the `UserDummy` as for test benches it is convenient to use assertions. Therefore assertions within the time critical phase (between `BEGIN` and `END TEST`) should be omitted or commented out if they are not explicitly wanted during final tests. Outside of this phase there is no need to optimize for speed.

The implemented test benches can be found in the directory `EsterProxy/test_bench.d`. The most important of them will be described in the Discussion 3 on page 31ff. For others one has to keep to the code.

`EsterProxy/test_bench.d`
- `debug.tb` .... was used for debugging
- `maxthp-serpck.tb` .... find maximal transfer using compressed containers
- `maxthp.tb` .... older test bench to find throughput maxima
- `maxthp-vector.tb` .... find maximal transfer using a vectors
- `measure-full.set` .... subset for measure.tb
- `measure-sgl.set` .... subset for measure.tb
- `measure.tb` .... various tests esp. for serialization evaluation
- `measure-vec.set` .... subset for measure.tb
- `minlat.tb` .... find minimal possible latency
- `onerun.tbb` .... the big listing above
- `trial.tbb` .... used during development
- `tsr.tb` .... use case evaluation

[12]The original can be found in `EsterProxy/test_bench.d/onerun.tbb`.

2.4.1. EsterProxyTest

The class `EsterProxyTest`[13] consists of three methods which we have all seen in the listing 2.1. There is (a) the constructor `EsterProxyTest`, (b) `beginTest`, (c) `endTest`, which surround a test case.

The constructor (a) forwards its input to `client.prepareTest` thereby instantiating a test on the proxy. Following this call one enables various features for the test by calling the `enable`*`Something`* remote methods on the `client` instance. After the configuration a test begins with a call to `beginTest` (b). This internally calls `client.startupTest` which plans the start of the prepared test case on the proxy and returns its point zero (start time). The parameters of `beginTest(oneway,batchSize)` setup the client side remote call semantics – that is using `oneway` calls, and whether `RCF` batching should be enabled (if $batchSize > 0$). Then it will block until point zero is reached on the client side hereby assuring that the test will not commence in advance. Now a test is running and one has to let the proxy swallow its spikes by calling any of the `client.swallow(`*`Something`*`)` methods. Finally after all spikes have been transferred a call to `endTest` stops the test. At first it stops the client side duration and then after setting the remote call semantics back to normal (two-way, no batching) it calls three remote methods on the `client` object: (1.) `shutdownTest` (2.) `finalizeTest` (3.) `fetchTestResult` which respectively initiate the shutdown, wait for it to be completed and then fetch the test result object. With the call to `client.fetchTestResult` the proxy is also set back to "ready" state expecting a new test. The next section will list the available results.

2.5. Test Results

The measured durations can be directly accessed from the `EsterProxyTest` instance:

durationClient all swallows have been performed but data might still be on its way (from `UserDummy` to `EsterProxy`)

durationProxy all data has been received by the `EsterProxy` (no outstanding one-way calls left)

durationArq all data has now passed the `ArqDrain` - the test is finished!

Additional results are offered by the `TestResult` member `EsterProxyTest.res`:

proxyReceiveLatency latency of reception of spikes by the proxy (`RCF` latency)

packagerSendLatency latency of first spike in hardware packet when it is ready to leave the `EsterProxy`

packagerPacketConstructionCount number of packets constructed by the packager

packagerPacketDropCount number of packets that have been dropped because of a full `ShamemIPC`.

[13] It is defined in `EsterProxy/esterProxyTest.h`.

packagerShamemBlockCount total time the packager was blocked because the `ShamemIPC` was full

packagerLastTransferredSpiketime spike time of last spike of last packet that was put into the `ShamemIPC`

arqReceiveLatency latency of reception of packets (first spike) by the `ArqDrain`

arqReceivedPackets number of packets the `ArqDrain` received

arqReceivedEntries number of entries the `ArqDrain` received (an entry is either a spike or an overflow)

arqReceivedSpikes number of spikes the `ArqDrain` received.

arqLastSpikeAvailable only if the packets have been read (`measureArqReceiveLatency`) the `ArqDrain` knows of spikes

arqLastSpike last spike received by the `ArqDrain`.

2.6. ARQ Drain

The `ARQ_Drain` program's[14] job is to simulate the data delivery to the RDMA-NIC.

The data is not actually passed to the RDMA-NIC but –after an eventual latency measurement– simply thrown away. This is due to time constraints for this thesis as well as due to the fact that the receiving side has not jet advanced far enough that such a test would give us a lot of additional information. But to implement a real `ARQ Drain` is considered not to be very problematic. The Outlook will present some information about that, see section 4.2.

Nonetheless the `ArqDrain` receives its data formatted in a way that it should be easy to move on. The received payload is wrapped in instances of `arq_shamem_t`[15] queued in a `ShamemIPC` buffer. The wrapper contains as first element the byte length of the *wire* data and as a second element the ARQ header as well as the payload itself. The payload is an FPGA Pulse Packet (`fpga_pulse_packet_t`)[16]. which may contain arbitrary bytes at the end if it is not full. Therefore only as much bytes as specified by the first element must be transferred to the FPGA. A working `ArqDrain` has to fill in the ARQ header as the `EsterProxy` does not know about the ARQ protocol apart from its header size. With that done one has to pass to the RDMA-NIC only a pointer to the first and second element. Figure 2.2 shows the struct `arq_shamem_t`.

[14]The main function is located in the file `EsterProxy/ArqDrain.cpp`.
[15]The struct `arq_shamem_t` can be found in the file `EsterProxy/ipc_xferTypes.h`.
[16]`fpga_pulse_packet_t` specifications: *UHEI and TUD* (2011, cf. sections 1.3.5 and 1.3.5).

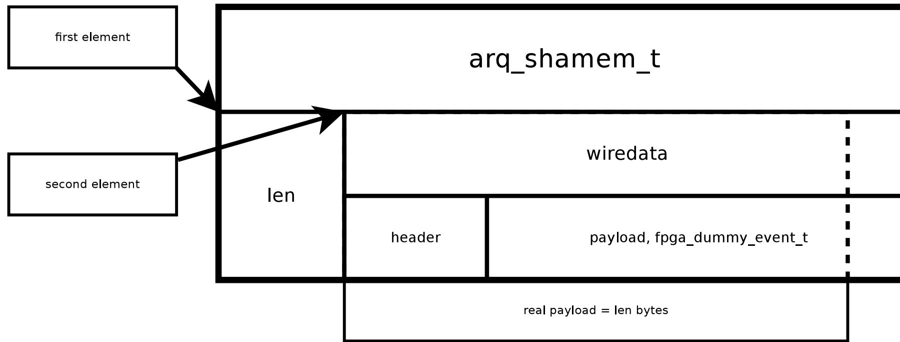


Figure 2.2.: The `arq_shamem_t` struct

Code In the *`EsterProxy` Suite* the `ArqDrain`'s first job is to initialize the `ShamemIPC`. For this a command line parameter specifying the `ShamemIPC` buffer size can be given.

```
1 ShmInitialiser shminit(ARQ_IPC_SHAMEM_ID, shamemIpcCbSize);
2 ShmReader *reader = ShmReader::rconnect(ARQ_IPC_SHAMEM_ID);
3 reader->startReader();
4 // BLOCKS until user calls client.establishArqConnection()
```

In line 1 the shamem is initialized with the identifier `ARQ_IPC_SHAMEM_ID` and a size of `shamemIpcCbSize`. The identifier specifies the shamem and must be the same for the `EsterProxy`. If no command line parameter was given the size defaults to `ARQ_IPC_SHAMEM_DEFAULT_SIZE` (33). The ID as well as the default size are specified in the file `EsterProxy/common.h`. Line 4 blocks until the `EsterProxy` starts the writer which is done as soon as the `UserDummy` connects to the `EsterProxy` and calls `client.establishArqConnection()`.

After this an `RCF` connection to the `EsterProxy` is established which is used to fetch test settings and to submit results. For the actual work of the `ArqDrain` this `RCF` connection is not necessary, it is used solely for the evaluation procedure.

Now the `ArqDrain` enters an active wait loop polling the `EsterProxy` until a test with enabled ARQ transfer has been planned.

```
1 while ( ! client.checkArqWakeup() ) { /*sleep a bit*/ }
2 // fetch test settings:
3 client.getArqConfig(testName, measureArq, spikesExpected);
4 do {
5     pointZero = client.getPointZero();
6     // we need to get this fast, its crucial! (no sleep)
7     std::this_thread::yield();
8 } while ( pointZero <= 0 ); // with a point zero available the test has
      been fully configured (and planned)
```

Finally the test has begun and the shamem is read until a stop packet with a length of 0 is received.

```
1 while (true) {
2     if ( reader->tryread(data) ) {
3         if (data.len == 0) break;
4         // if measurement is enabled:
5         // a) compare spiketime of first spike in packet with now (latency)
6         // b) loop through the data to update the last spiketime
7     } else { std::this_thread::sleep_for( ARQ_EMPTY_SHAMEM_YIELD ); }
```

When the test is done the results are returned to the proxy and we enter the `while ( ! client.checkArqWakeup() )` loop again. Note that the above listings have been shortened and do not reflect the actual code.

2.7. The `EsterProxy` program

The `EsterProxy` program (`EsterProxy`)[17] is the core of the *`EsterProxy` Suite* and the main issue of this thesis. It is the part which is closest to a real implementation and contains the algorithms which are under evaluation. In opposite to the other two programs it is not a simulation – it performs actual work. Therefore this section is also the most important of this chapter and puts it all together. It does not look at the `EsterProxy` as an isolated object but outlines the design of the whole *`EsterProxy` Suite* thereby concentrating on how the `EsterProxy` gets its job done. One may take a quick look back to figure 2.1 on page 8 which showed a schematic overview of the *`EsterProxy` Suite* and the physical location of its parts before moving on.
In the following the design is discussed alongside the path spike data takes from the `UserDummy` to the NIC. For orientation figure 2.3 may help. The `EsterProxy` does not

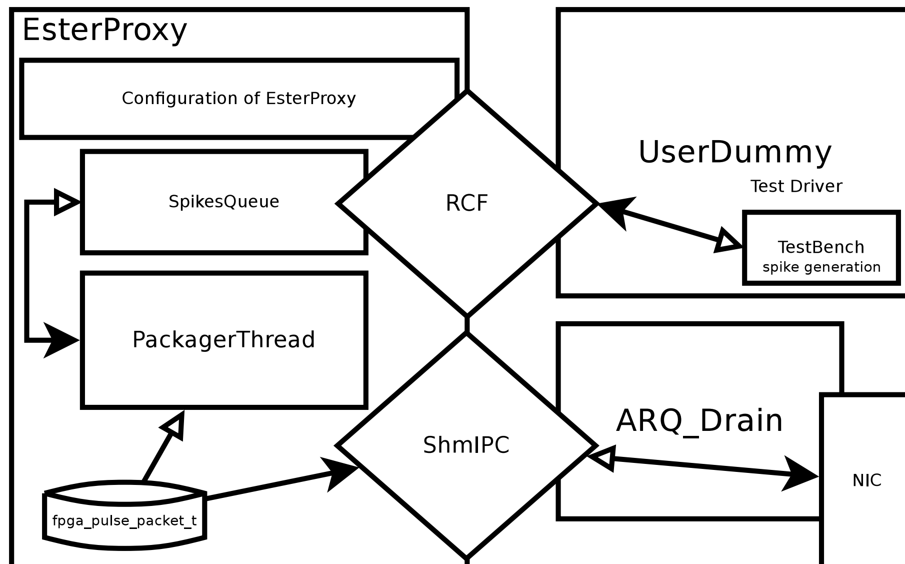


Figure 2.3.: Schematic of EsterProxy Logic

[17]The main function is located in the file `EsterProxy/EsterProxy.cpp`.

operate on its own, it is instead completely driven by the UserDummy. In the *EsterProxy Suite* the UserDummy serves as the "Test Driver".[18] It performs the initial EsterProxy setup as well as it configures the test cases and produces the test data. The sole configuration that can be passed to the EsterProxy directly is a command line parameter specifying the listening address where it expects the UserDummy to connect through. For easy development it defaults to localhost.

Main function The code of the EsterProxy main function itself is therefore rather small –it is only an RCF server wrapper– and can be summed up as follows:

```
1  RcfServant proxy; // create RCF servant object
2  RCF::RcfInitDeinit rcfScopeInitializer; // setup RCF
3  RCF::RcfServer server; // create an RCF server
4
5  // bind addresses/ network interfaces
6  server.addEndpoint( RCF::TcpEndpoint( bind_address, PROXY_DEFAULT_PORT ));
7  if ( bind_address != "localhost" ) server.addEndpoint( RCF::TcpEndpoint("
       localhost", PROXY_DEFAULT_PORT) );
8
9  server.getServerTransport().setMaxMessageLength( PROXY_MSG_SIZE_MAX );
10 server.bind<I_EsterProxy>(proxy); // bind the servant object to the
       I_EsterProxy RCF interface
11 server.start(); // start RCF server thread(s)
12 while( !proxy.shutdown ) {doSomeOccasionalReport();sleepAbit();}
```

There is not much to add to the listing above.[19] Line 7 assures that the RCF server is always at least bound to localhost as the ArqDrain does connect from there. The most important is line 10 where the servant object is bound to the I_EsterProxy RCF interface. The RCF Servant object (RcfServant) is responsible for the UserDummy remote calls. But before data arrives there it has to be serialized. So we continue with the different spike data serialization methods.

2.7.1. Serialization

The Dummy Spike Data struct (fpga_dummy_event_t) serves as basic object for spike transfer between the UserDummy and the EsterProxy. It is located in EsterProxy/rcf_-xferTypes.h where all the *EsterProxy Suite*'s RCF-SF serializable classes are declared.

[18] How test benches can be implemented was shown back at section 2.4.

[19] For details on the RCF methods one may check their documentation at http://www.deltavsoft.com/doc/index.html.

```
1  struct fpga_dummy_event_t {
2    // ------ TYPES -----------------------------------------
3    typedef HMF::FPGAPulsePacket::spiketime_t spiketime_t;
4    typedef HMF::FPGAPulsePacket::label_t label_t;
5    // ------ MEMBER VARIABLES ------------------------------
6    spiketime_t      spiketime;  // global time in cycles
7    label_t          label;
8    // lifecycle: constructer, etc
9     [...]
10 }
```

Listing 2.6: Dummy Spike Data struct (`fpga_dummy_event_t`)

There are basically two methods to serialize: member-wise or byte-wise (listings 2.7 resp. 2.8). The first of which suffers from a considerable call overhead[20] (esp. for small elements), whereas the second is undoubtedly faster, but cannot be used under all circumstances. E.g., complex classes with pointers and variable sized objects cannot be trivially serialized as a whole using the byte-wise method.

```
1  void serialize(SF::Archive &ar) {
2     simple but very slow
3    ar
4       & spiketime
5       & label;
```

Listing 2.7: member-wise

```
1    if (ar.isWrite()) {
2      ar.getOstream()->writeRaw(reinterpret_cast<char*>(this), sizeof(
           fpga_dummy_event_t));
3    } else {
4      ar.getIstream()->read(reinterpret_cast<char*>(this), sizeof(
           fpga_dummy_event_t));
5    }
6  }
```

Listing 2.8: byte-wise

For the `fpga_dummy_event_t` the byte-wise serialization was chosen. But serializing or transferring single spikes is still far from efficient. Even with enabled `RCF` batching it does not improve. See chapter Discussion 3.3.1 on page 39 for the rationale.

As the size of a `fpga_dummy_event_t` is with 12 bytes rather small the only reasonable approach is to use a container format. As possible containers a `std::vector<fpga_pulse_packet_t>` as well as various self-made ones have been evaluated.

The simplest available container format is the vector. But in its default implementation this is still not very fast as the serializer must loop through the vector and serialize its elements one by one; it does not understand that the vector's elements can be byte-wise

[20] This has been investigated using the `valgrind` tool `callgrind`.

serialized. To solve this this a RCF-SF vector specialization[21] can be specified:

```
1  template<typename A>
2  inline void serializeVector(
3    SF::Archive & ar,
4    std::vector<EsterProxy::fpga_dummy_event_t, A> & det,
5    boost::mpl::false_ *)
6  {
7    size_t sz = 0;
8    size_t const minSerializedLength = sizeof(EsterProxy::fpga_dummy_event_t)
       ;
9    if (ar.isRead()) {
10     det.clear();
11     ar & sz;
12     if (ar.verifyAgainstArchiveSize(sz*minSerializedLength))
13       det.resize(sz);
14     else throw std::runtime_error("unexpected size!");
15     ar.getIstream()->read(reinterpret_cast<char*>(det.data()), sz*
         minSerializedLength);
16   } else if (ar.isWrite()) {
17     sz = det.size();
18     ar & sz;
19     ar.getOstream()->writeRaw(reinterpret_cast<char*>(det.data()), sz*
         minSerializedLength);
20   }
21 }
```

Listing 2.9: Vector Specialization

But better still performed a self-made container using delta compression upon insertion (i.e. the data is compressed already before it gets to the serializer). For details on this container format one should look at the code.[22] The performance of the different containers is discussed in section 3.3.2 on page 41. The Outlook also has some information on how this container could be made even more efficient, see 4.6 on page 55.

2.7.2. RCF Servant

The RcfServant handles all RCF calls which arrive at the EsterProxy. These conduct the general setup of the EsterProxy and the preparation, configuration, finishing, evaluation of test cases. A complete list of the RCF calls can be found in the Appendix A.2.
Last but not least there are the swallow calls which receive the spike data. There are five swallow methods handling various data types. They all forward their data to an inlined handleSpike resp. handleVector method.

handleVector This method receives the de-serialized spike data as a vector of fpga_pulse_packet_t. After de-serialization –on the path down to the packager– the vector's data is never copied again. Instead of looping through the vector and measuring

[21] The code of this specialization is –due to include constraints– not placed with the other RCF-SF serializable classes but in the file EsterProxy/rcf_interface-base.h.
[22] The class FpgaDummyEventSerializer can be found in the file EsterProxy/rcf_xferTypes.h.

the latency for each spike we take the spike at position 1/3 of the vector as a probe for the whole vector. So for the first third of the vector we get a better latency and for the following two third we get a worse latency for the evaluation. This latency calculation is considered “fair” as it returns a latency close to the expected average. Under normal circumstances (about equally distributed spikes within the packet) the estimated latency will never look better than the actual spike-wise latency. One is always interested in latencies rather close to the worst-case latency as this it the pivotal question.

```
1  inline void RcfServant::handleVector(std::vector<fpga_dummy_event_t> &
     vector) {
2    // [...] some #ifdeffed debug code, disabled during evaluation tests
3
4    t->handledSpikes += vector.size(); // all received spikes are counted
5
6    // we take the event at position 1/3 of the vector as latency equivalent
       for the whole vector.
7    // this can be considered "fair enough".
8    double spiketime_dbl = vector[(vector.size() / 3)].spiketime /
       CYCLES_PER_SECOND;
9
10   // drops the whole vector!
11   if ( t->isSpikeDropping
12     && ( MicroCounter::getSeconds( - t->pointZero ) > (spiketime_dbl + t->
         spikeDropTime) )
13   ) {
14     t->spikeDropCnt += vector.size();
15     return; // drop spike!
16   }
17
18 /* FUTURE: With multiple packagers this gets tricky.. We must seperate the
     vector for the various packagers. We could do something like a shared
     pointer to a vector and every packager loops through it or we must
     return to single spike handling, looping through the vector and moving
     spike per spike into the correct packager.. */
19
20   if (packager) packager->insert(std::move(vector)); // vectors are always
       moved -> no copy!
21
22   if ( t->latencyReceiveMeasureEnabled ) {
23     t->latencyReceiveMeasure.push_back(MicroCounter::getSeconds( - t->
         pointZero ) - spiketime_dbl); // receive - "creation" of spike
24   }
25 }
```

Listing 2.10: `RcfServant::handleVector`

The `handleSpike` method looks analogous. These two methods contain everything which the `RcfServant` does during the run phase of a test. But there is also a lot of configuration, etc. that has to be handled beforehand. How a test case is configured has been described in section 2.4 and a complete list of all available `RCF` calls can be found in the Appendix A.2. The following outlines the environment in which these `RCF` calls are handled.

Life cycle To control the life cycle of the `EsterProxy` program and the test cases the `RcfServant` has been implemented as a state machine. Figure 2.4 shows the components of the `RcfServant` and its state machine. Important to note is that the instances of the `testdata_t` struct as well as the `SpikePackager` class are wrapped into `std::unique_ptr` objects which will be instantiated anew for each test case. This assures that there are no side effects of one test to another. The `ShmWriter` pointer on the other hand points throughout the program's lifetime to the same object. It is reused during tests; this is not necessary the best choice, see Outlook 4.7.

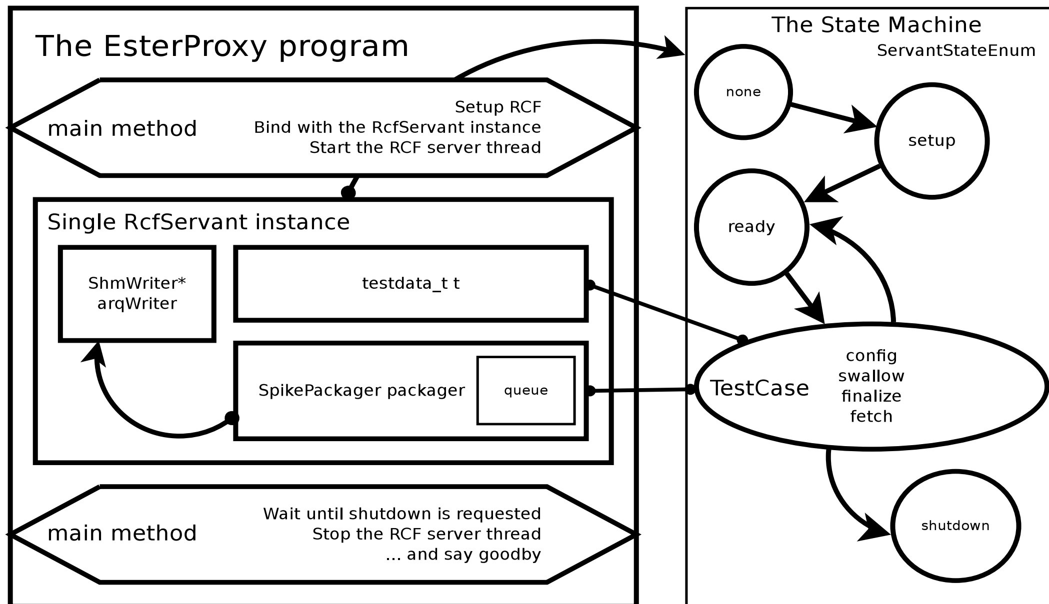


Figure 2.4.: Schematic of EsterProxy Internals

2.7.3. SpikesQueue: `DoubleSidedMutexedQueue`

In listing 2.10 on line 20 the received spike data is inserted into the packager's queue. The initial queue (at this time only single spike handling was implemented) was a simple `std::queue<fpga_pulse_packet_t>`. This queue suffered severely from the synchronization of the multi-threaded access (insertion by the `RcfServant` thread and removal by the packager thread). For each spike inserted or removed it was necessary to acquire a mutex.

To improve these circumstances the class `DoubleSidedMutexedQueue`[23] was developed. It is called such because it is a compound of two queues specialized for "mutexed" multi-threaded access. The trick is that the initial queue is split into two queues, one for insertion and one for removal. When the queue for removal is empty both queues are exchanged.

[23]The class `DoubleSidedMutexedQueue` is located in the file `EsterProxy/doubleSidedMutexedQueue.h`.

A mutex must now be acquired only for insertion and when we need to switch both queues. So the removal can most of the time work unsynchronized. For single spike transfer a speed up of about 20 % was observed.[24]

With the support for vectors this queue is now holding `std::vector<fpga_pulse_packet_t>` objects and insertion as well as removal are less frequent. This means that the importance of this queue has been reduced. But with the support for multiple FPGAs there might be the necessity to go back to single spike handling in the `swallow()` methods. See Outlook 4.3 on page 53. This would make the `DoubleSidedMutexedQueue` interesting again.

2.7.4. The Packager

The FPGA Spike Packager class (`SpikePackager`) defined in the file `EsterProxy/spikePackager.h` finally takes care of preparing the hardware-formatted delta-compressed `fpga_pulse_packet_t` packets. It also offers them to the `ArqDrain`. The packager works in its own thread sedulously packing spike after spike. The packager thread is separated into three parts/loops.

The first loop waits on the first spike to be inserted into a new packet. After the spike is inserted the packet time-out (`xferwalltime`) is calculated. This time-out is the wall time when the packet must at the latest be written into the `ShamemIPC`. It is the spiketime of the packet plus the `packetBufferTime` which was specified by the packager's constructor.

The second loop packs more spikes into the packet until it is either full or reaches its time-out. It is to much overhead to check if the packet has expired after every spike. So this test is only done every 32 spikes and whenever the spikes queue is empty. If this system is supposed to be kept up a clock thread could be helpful to move the time spend on fetching the wall time out of the packaging thread (see Outlook 4.4.

The third loop now takes care for writing the newly build packet into the `ShamemIPC`. This task can only be performed if the `ArqDrain` is not running behind (i.e. if the `ShamemIPC` buffer is not full) If the write fails more spikes are put into the packet until it is full. After each insertion the packager thread `yield`s and retries the `ShamemIPC` write. If the `arqXfer` feature was enabled (see Appendix A.2) with packet dropping the packet will be dropped after a specific time spend on retries which is defined in `common.h`. This dropping feature (packet dropping) could not be very well tested as the `ArqDrain` is just a simulator. However this dropping design is considered useful as it moves the dropping to the `EsterProxy` in case of an overloaded network (in this case the `ShamemIPC` buffer will be full). It must be paired with a reasonable-sized `ShamemIPC` buffer size dependent on the packets the RDMA-NIC can handle concurrently and the average time it spends per transfer.

[24]There is no record for this observation as it happened very early during this project.

3. Discussion and Results

This chapter presents all performed tests as well as their outcome. It begins with a discussion of `RCF` in general and its possible speed on a local network. It shows how `RCF` calls can be made most efficient, considering throughput. Secondly, the `ShamemIPC` using a *`EsterProxy` Suite*-specific buffer type, but other than that being independent, is examined.
Finally it concludes with the evaluation of the suite: serialization, maximal throughput, minimal latency and a use case simulation.

Definitions For the following evaluations some definitions should be stated. All graphs show the standard deviation (sd) multiplied by two (i.e. the 95 % confidence interval). Often the standard deviation has been given two colours, the smaller darker area then represents one sd. Note that the standard deviation is very small in many of the tests and in most of the graphs not easily visible. Frequently data "sizes" have been normalized due to packet overhead or data compression, yielding an *effective* size that corresponds to the payload (i.e. spikes or other user data).

All plots and tables that speak of spikes use *normalized bytes* for the presentation of throughput values. For normalization spikes have been multiplied by 10 to return a representative value in bytes. This is due to the fact that a packed Dummy Spike Data struct (`fpga_dummy_event_t`) has a size of 10 bytes. There had been some problems with this which are described in the Outlook 4.8. Note that a spike packed into a `fpga_pulse_packet_t` packet needs less bytes: about four to eight bytes. This is due to the delta-compression. If we assume an average biological firing rate of 1 Hz and a required spike precision of 1 ms this yields $log_2(\frac{1000}{\text{s}}/1\text{Hz}) \approx 10\,\text{bits}$ to encode a spike on average. The current specification provides 15 bits for timestamp encoding (and a precision similar to 1 ms); longer time periods are handled using *overflow* entries that hold 31 bits of *silence*.

A given throughput is generally calculated as follows: divide the total of transferred data by the duration of a test. Depending on the viewpoint there can be multiple measuring points considered as the end of a test.

For evaluation purposes a spike gets as spiketime a representation of its creation time. The creation of a spike usually takes place just before it is transferred; or, in the case of batched transfer, just before the spike is put into its container (i.e. the transfer object). In other words, the creation time specifies the time at which the spike is available to the `UserDummy`. Latency is then measured by comparing the spiketime to the wall time at certain points. I.e. it is the time consumed by the transmission to a specific point. The available measuring points are specified later alongside the results/plots.

All tests have been performed on the *HMF Cluster*. This computer cluster forms the HMF Conventional Part. It consists of sixteen fast nodes connected with 10 GbE Ethernet. Important characteristics are:

Fast Network – 10 GbE network interface cards (Intel NetEffect NE020) connected to one 10 GbE 24-port ToR network switch (Hewlett Packard ProCurve Switch 6600-24XG); the 8 unused ports are planned to be used by multiple wafer-scale systems (connecting to one 24-port 1 GbE to 2-port 10 GbE aggregating switch (Hewlett Packard ProCurve Switch 2910AL-24G) for every wafer-scale system).

Control Network – 1 GbE, Intel on-board NICs with built-in KVM (Intel AMT)

CPU – Intel Core i7-2600, Sandy Bridge architecture @ 3.40 GHz

RAM – 16 GiB DDR3 memory, 2× dual channel equipped

HDD – 4 out of 16 nodes equipped with two OCZ Vertex 3 each: Fast RAID 0 storage for spike recording at wire speed (supporting 1 GB/s)

OS – Debian Wheezy (GNU Linux distribution running Linux `3.2.0-1-amd64`)

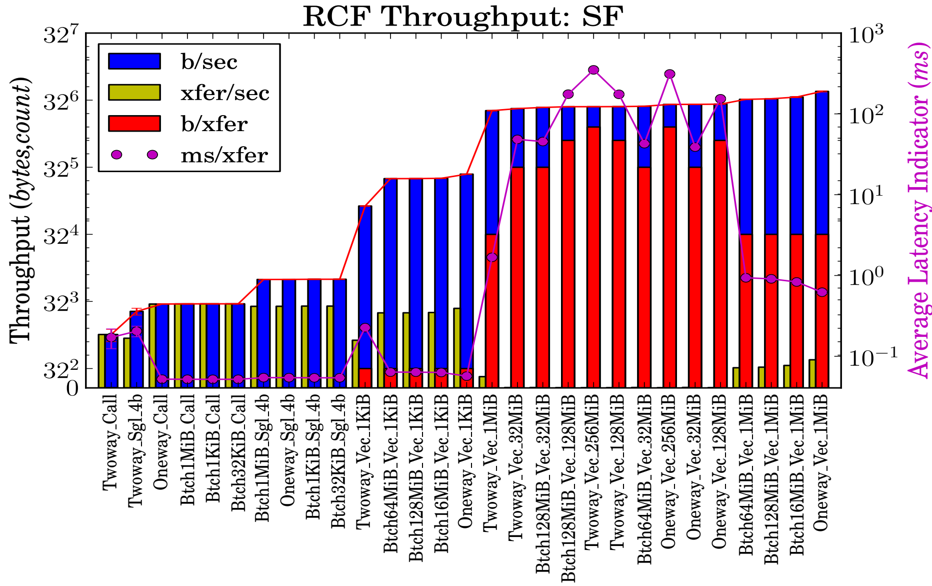
3.1. `RCF` in General

Before we talk about the ***EsterProxy** Suite* we must discuss the `RCF` potential in general. `RCF` offers the use of three different serialization frameworks:

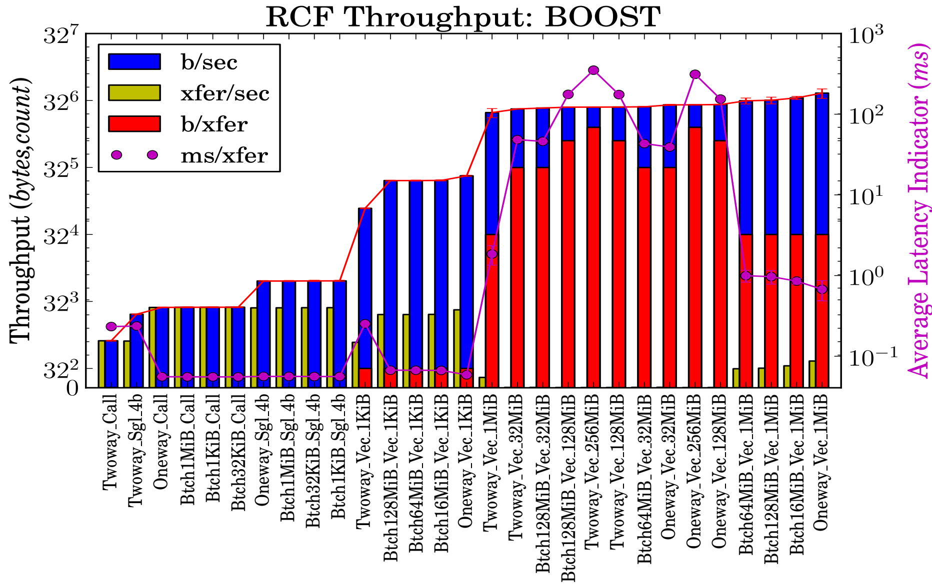
1. `RCF` Serialization Framework (`RCF-SF`) – built-in
2. `RCF` `BOOST` Serialization (`RCF-BOOST`)
3. `RCF` Google Protobuf Serialization (`RCF-PROTO`)

All three of them have been subject to testing. The primary goal of `RCF-PROTO` is clearly not the fast transport of specialized data as in our case. To make data serializeable by Google Protobuf Library (`protobuf`) one has to write a `.proto` file in a specific syntax and then call the `protobuf` compiler on it (`protoc myprotofile.proto`). This will generate a verbose class with getters and setters and other convenient stuff. What is interesting about this code generator is that it can generate outputs not only for C++ but also for *Java* and *Python*. However it was quickly seen that `RCF-PROTO` will not suffice to cover all our needs as it concentrates more on convenient data handling as well as usability than on speed and strong typing. Therefore no further advancement in direction `RCF-PROTO` was made.

Figures 3.1a resp. 3.1b show the evaluation of `RCF-SF` and `RCF-BOOST` serialization. Primarily they can be grouped into three transfer types: (a) RCF-call (no transfer, suffix `_Call`) – measuring call overhead, (b) single-int (a single `int` per call, suffix `_sgl_4b`), (c) vec-int (`vector<int>`, suffix `_vec_*`). The `*` in the vec-int transfers stands for the size



(a) RCF-SF throughput



(b) RCF-BOOST throughput

Figure 3.1.: RCF throughput dependent on different call semantics and transfer objects. The blue bars show the throughput in bytes per second and the red bars the correspondent size of the transfer objects. The yellow bars denote the count of performed RCF method calls (i.e. transfers) per second. The Average Latency Indicator shows the average duration of a single transfer. Note that for RCF batching the actual number of transfers is unknown – here we count the number of method calls. For the other semantics these are equivalent.

of the transferred object. For the RCF-call throughput calculation the transfer object size was defined as 1 byte.

The prefix specifies the *remote call semantics* used for the transfer, which are (A) Two-way (wait for the remote call to return), (B) One-way (just set off the call and do not wait for it to return). (C) Batched (like one-way but also use batching of size `*`, prefix `Btch*`). Combining the above with various transfer object sizes and batch sizes we obtain the "`RCF` Throughput" figures. The total transferred data for the vec-int tests was 2 GiB. For the single-int and RCF-call tests it was reduced by a factor of 1024 as these tests otherwise became to lengthy. Please keep in mind that the RCF-call did not actually transfer any data, it just did one remote call per "pseudo-byte". On the receiving side the data was not used but per received `int` resp. call (RCF-call) a counter was increased. I.e. the vector-receiving methods looped through the vector counting all `int`s contained. Each test was finalized by a two-way call which assures that all prior remote calls have been handled before we then measure the duration of the test. All tests have been performed 9 times for standard deviation calculation. Listing 3.1 shows the remote call interface used for the *RCF Serialization Evaluation* (`RCFSerEval`).[1]

```
RCF_METHOD_R0(size_t, reset) // once at the end of each test (returning and
    resetting the counter)
RCF_METHOD_V0(void, rcfcall) // (a)
RCF_METHOD_V1(void, swallow, const int &) // (b)
RCF_METHOD_V1(void, swallow, const intvec &) // (c)
```

Listing 3.1: `RCFSerEval` remote call interface

Results The `Oneway_Vec_1MiB` tests performed best with both serialization frameworks.

- `RCF-SF`: `Oneway_Vec_1MiB` with $1.688 \times 10^9 \pm 2.41 \times 10^7\ {}^{bytes}/_{seconds}$
  Latency indicator: $0.621\ {}^{ms}/_{transfer}$ (2048 transfers)
- `RCF-BOOST`: `Oneway_Vec_1MiB` with $1.575 \times 10^9 \pm 1.89 \times 10^8\ {}^{bytes}/_{seconds}$
  Latency indicator: $0.676\ {}^{ms}/_{transfer}$ (2048 transfers)

Also the latency indicator being less then a millisecond is considerably good. It is the total test duration divided by the number of RCF calls[2] (transfers) that have been made whereas the throughput are the total bytes divided by the test duration – so both values depend on the same measure. We can also see that the `RCF` batching is not very good. In fact it does never win against one-way vectors. The only tests where the batching wins are RCF-call (always) and single-int (mostly). There is no justification to use `RCF` batching for data transfer[3]. Two-way calls are –as expected– the slowest option. Interesting though is the observation that as bigger the transfer objects get the lesser the

[1]The code of this evaluation is located in the `rcf-lib` repository.
[2]For one-way calls this is the only way to rate the duration of a single call (on the caller side) as they do not block.
[3]Note that on other computers batching performed even worse. But all final tests have been performed on the CP cluster, therefore only these results are presented here.

negative effect of two-way calls compared to their one-way counterpart; this is expected as for big objects the time spent on the actual transfer of the data outweighs the call overhead. That said it might be useful to occasionally intersperse two-way calls because this assures that all prior one-way calls have been handled; a two-way call does block the client side until the call was handled by the server. This will not happen before all prior (one-way) calls have been handled as well. Otherwise the caller cannot expect that its call will be handled quickly as the one-way calls will stack if they come in faster then the callee can handle them. Therefore the *EsterProxy Suite*'s minimal latency test used two-way calls (see section 3.3.3).

This test has been performed on the *CP Cluster* with the caller and callee connecting through localhost. The network protocol was used but the network controller was bypassed. This was intentional as an unbiased RCF serialization overhead measurement was sought.

3.2. ShamemIPC in General

The connection from the EsterProxy to the ArqDrain is realized using a ShamemIPC with arq_shamem_t buffer entries holding FPGA Pulse Packets (fpga_pulse_packet_ts). The ShamemIPC test program[4] implemented for this evaluation therefore uses the same buffer entries. Other than that it is independent from the *EsterProxy Suite*. As we are –in the end– interested in spike throughput, figure 3.2 shows the throughput of the ShamemIPC in normalized bytes; i.e. 12 bytes per spike (see this chapter's introduction).

This evaluation distinguishes between two different methods of accessing the ShamemIPC entries: (a) pointer, (b) memcpy. Their difference is outlined in Design 2.1. For the final ArqDrain implementation it is expected that reading through the payload is not needed. Its job will only be the preparation of the ARQ header and to offer the packets to the RDMA-NIC. Any knowledge of the data therein which the ArqDrain might need should have been extracted by the EsterProxy beforehand; and saved alongside the payload in the ShamemIPC buffer. But for measuring the latency the *EsterProxy Suite*'s version can optionally read through the payload, accessing every single spike and thereby updating the last spiketime. As the fpga_pulse_packet_t uses delta compression and does not contain a global offset time stamp this is necessary if one wants to know the actual spiketime of any single spike.

Table 3.1.: ShamemIPC Throughput parameters

Entry Access		Spike Access		Buffer Size
pointer – direct		whole packet only		4
	×		×	to
memcpy – indirect		reading every single spike		125000

[4]The code of the ShamemIPC evaluation can be found in EsterProxy/ShamemThroughput.cpp.

Therefore the second test parameter (Spike Access) specifies exactly that. In the figure the two lines on the bottom are those which read (rd) through the packet, accessing every single spike.

The third and final *sweep* parameter was the buffer size as it was expected to have an effect on the outcome. The buffer size (number of `arq_shamem_t` entries) was increased by a factor of 1.25 per test, ranging from 4 to 125 k.

Results The results being straight lines for all four basic test parameters are unexpected as they show no relation between the buffer size and the throughput. This allows for a more flexible use of the `ShamemIPC`.

Table 3.2.: `ShamemIPC` top results

Test Case	Buffer Size (*entries*)	Throughput (*norm. bytes*)
pointer (rd)	4038	$9.100 \times 10^{+08} \pm 1.19 \times 10^{+06}$
memcpy (rd)	5048	$8.619 \times 10^{+08} \pm 1.11 \times 10^{+06}$
pointer	46	$1.288 \times 10^{+09} \pm 2.49 \times 10^{+06}$
memcpy	24074	$1.221 \times 10^{+09} \pm 1.90 \times 10^{+06}$

The throughput of about 2 GiB normalized bytes is promising though it is not as good as expected considering the *HMF Transmitter* evaluation (*Husmann*, 2011). The *HMF Transmitter* evaluation promised about 8 GiB of real data throughput for one pair of a writer and reader head. Probably this is due to the additional code overhead in making the `ShamemIPC` a fully-fledged and stable release. But it could also originate in the packaging of the spikes. This should be taken into consideration once again as there seems to be some room for enhancement (see Outlook 4.9).

Also note that the pointer access method was implemented just shortly after the ***EsterProxy*** *Suite* had been declared "stable" and –being a draft implementation– it is therefore not used in the ***EsterProxy*** *Suite*. More on that can be found in the Outlook 4.1.

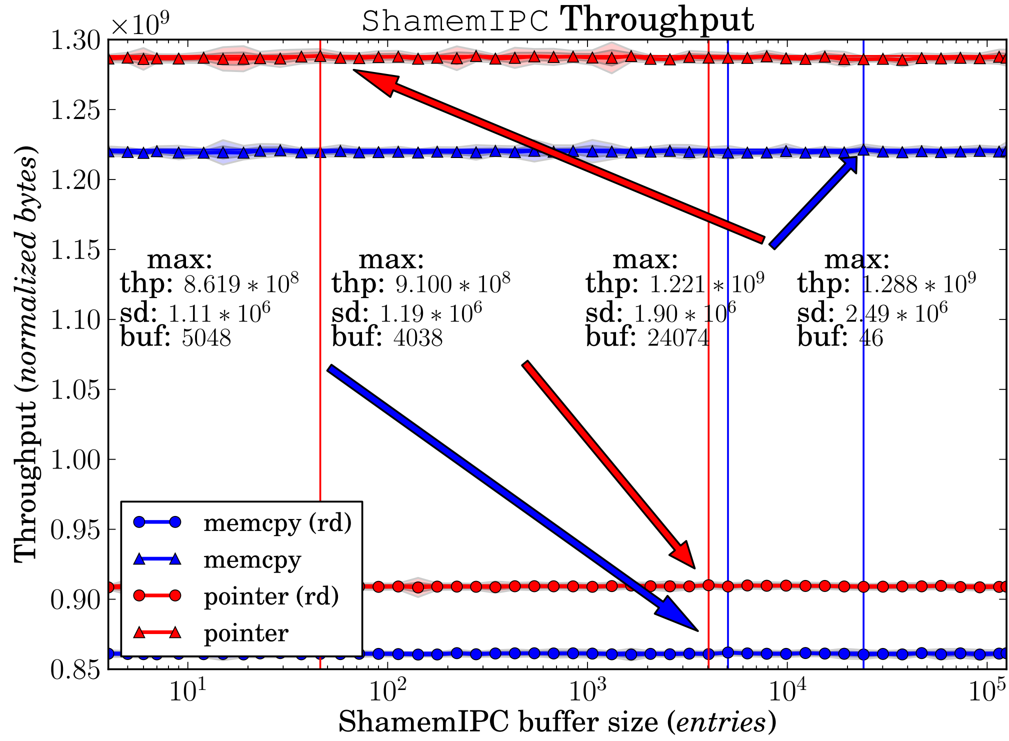


Figure 3.2.: Throughput of ShamemIPC using different access methods and looking for an applicable buffer size. The red lines display the pointer access method and the blue lines refer to the memcpy access method. Also we distinguish between reading through the received packet (bottom lines, rd) and accessing only the packet as a whole (top lines).

3.3. EsterProxy Evaluation

This section discusses various tests that have been performed on the *EsterProxy Suite*.

1. Single spike transfer and batching;
2. Container formats: esp. vector and delta-compressed packet;
3. Maximal throughput using prepared spike data;
4. Minimal latency using two-way calls and single spikes;
5. Use case evaluation using the `TimeSpanRnd`;

Before these tests are described and visualized in the following subsections a quick overview of the available measurements seems to be useful. See figure 3.3.

Throughput For each test on the *EsterProxy Suite* there are three potential measurements concerning the throughput. They all base on the total duration of a test divided by the total normalized (see page 31) byte count; They differ in when (resp. where in the code) the duration is measured.

thpC measure the duration after the last `swallow()` method was called.

thpP measure the duration on the `EsterProxy` just after all `swallow` methods have been handled.

thpA measure the duration as soon as the `ArqDrain` has received the `len==0` packet (i.e. the test has ended).

The thpC measurement (also called client throughput) can be understood as what the `UserDummy` “feels” as throughput whereas the thpP value (proxy throughput) is the simple unbiased `RCF` TCP/IP throughput. The thpA measurement (ARQ throughput) finally specifies the throughput of the complete suite with all features (packager and arqXfer) enabled. If the *packager feature* is enabled the packager thread is started and `fpga_pulse_packet_t` packets are prepared. If the *arqXfer feature* is also enabled (it depends on the packager feature) these packets are also transferred to the `ARQ_Drain` program. As long as all three values have been measured we can assert that $thpA < thpP < thpC$ (of course within the bounds of measurement precision).

Latency There are also three different latency measurements but they are not exactly analogous.

prx-recv latency measured when the `EsterProxy` receives a bunch of spikes (`swallow()`)

pkg-send latency measured just after a packet has left the packager

arq-recv latency measured when a packet arrives at the `ArqDrain`

For single spike transfer the prx-recv value is measured for each spike. This introduced to much overhead for multi-spike (container) transfer therefore it uses the latency of the spike at position 1/3 of the packet as an estimation. The other two latencies are "packet-wise" latencies, i.e. the latency of the first spike (highest value) in a packet specifies that packets latency. The pkg-send latency is also measured if the arqXfer feature is disabled. It then simply represents the packet latency when the packager has finished packing the `fpga_pulse_packet_t` packet as there is no transfer. In general we can say prx-recv < pkg-send < arq-recv.

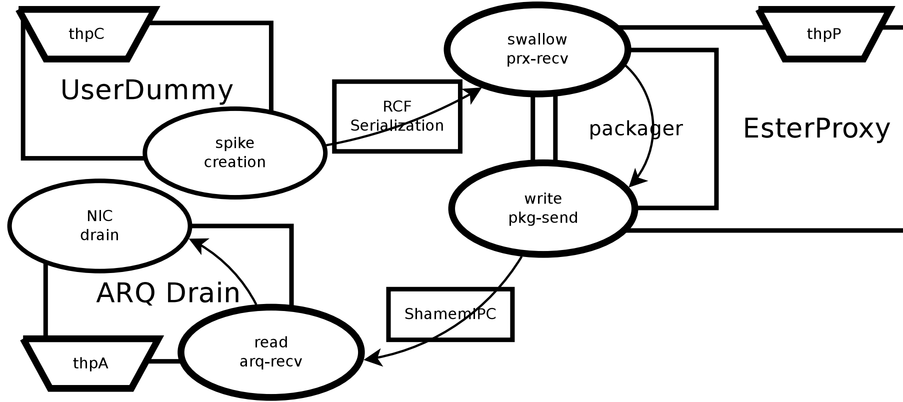


Figure 3.3.: A schematic of the measuring points and how the spikes pass through them.

3.3.1. Single Spike Transfer

This evaluation looks into the serialization and `RCF` throughput of single spikes. To get unbiased results the packager and arqXfer features have been disabled. Figure 3.4 compares byte-wise and member-wise serialization of the `fpga_dummy_event_t` as specified in Design 2.7.1. Note that the byte-wise serialization substantially equals to a memcpy method call, see listing 2.8. As second test parameter it evaluates the different remote call semantics available.

The plot shows quite clearly that the `RCF` batching is ineffective. Throughput could not be improved and the latency became worse with batching enabled. The only effect of batching is that the time spent on a single call by the client (i.e. `UserDummy`) is lowered which is why the thpC value increases. But the client side test duration is measured before the final batch is flushed. We can also see that the byte-wise serialization (blue boxes) performed better than the member-wise alternative (red boxes). Table 3.3 shows the most interesting results of the byte-wise serialization: One-way transfer is the best considering throughput whereas two-way performs best in latency. The best result for batching is also shown. It is worse in latency and throughput compared to the simpler one-way call. Latency is defined here as the distance between the creation of the spike and its reception at the `EsterProxy` (see prx-recv in section 3.3).

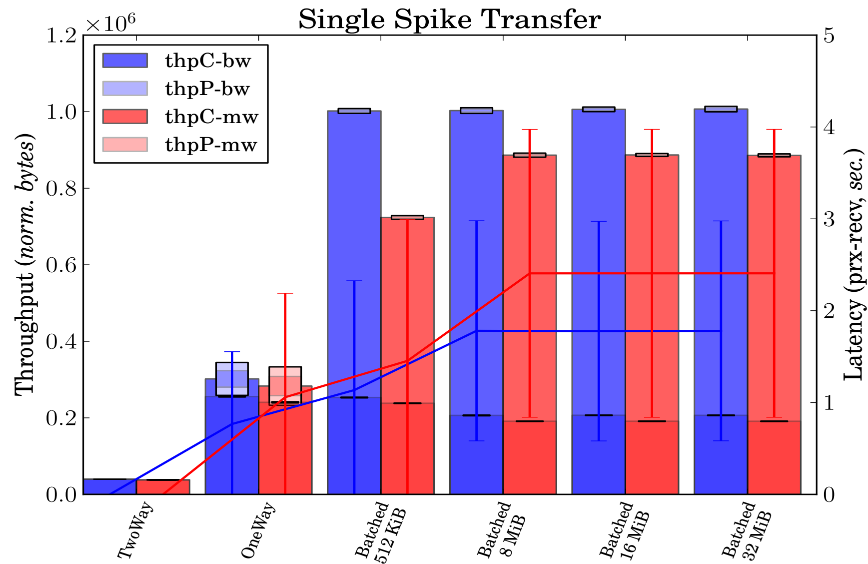


Figure 3.4.: Comparison of byte-wise (bw) and member-wise (mw) serialization of single spikes. The dark coloured smaller boxes are the important ones as they specify the actual `RCF` throughput whereas the light ones measure the client side throughput – which disregards the piling of unprocessed calls on the remote side. The blue and red lines show the correspondent latency (distance between `UserDummy` creation and `EsterProxy` reception).

Table 3.3.: Single Spike Transfer

CallSemantics	Throughput (thpP, *norm. bytes)*	Latency (prx-recv, *sec.*)
OneWay	$2.557 \times 10^{+05} \pm 5.38 \times 10^{+02}$	$7.666 \times 10^{-01} \pm 3.94 \times 10^{-01}$
Batched 512 KiB	$2.531 \times 10^{+05} \pm 4.24 \times 10^{+02}$	$1.136 \times 10^{+00} \pm 5.95 \times 10^{-01}$
TwoWay	$3.986 \times 10^{+04} \pm 4.21 \times 10^{+01}$	$1.680 \times 10^{-04} \pm 3.61 \times 10^{-06}$

3.3.2. Container Serialization

The single spike transfer evaluation showed that it is not fast enough and that `RCF` batching does not increase the throughput. So container formats had to be tested. All implemented containers have been probed with three different sizes (11 k, 33 k, 99 k). Figure 3.5 visualises them all and table 3.4 presents the top five results. The *EwV* test on the left is the default vector serialization (element-wise). Far better performs the *Vec* test which uses a byte-wise serializing vector specialization (basically a memcpy, see listing 22 on page 27). The best results provides the `FpgaDummyEventSerializer` container (denoted as *Pb* test, at the right of the figure). The `FpgaDummyEventSerializer` employs delta-compression for inserted spikes and then byte-wise serializes the whole chunk of data it holds. Interesting is how bad the *Pcp* test performs. It uses delta-compression but serializes the compressed spikes member-wise.
Byte-wise serialization is a must!

Table 3.4.: Container Formats

Container	Size (*spikes*)	Throughput (thpP, *norm. bytes*)
Precompressed Packet (Pb)	33,333	$1.992 \times 10^{+08} \pm 1.05 \times 10^{+06}$
Precompressed Packet (Pb)	99,999	$1.972 \times 10^{+08} \pm 9.53 \times 10^{+05}$
Precompressed Packet (Pb)	11,111	$1.902 \times 10^{+08} \pm 9.18 \times 10^{+05}$
Specialized Vector (Vec)	11,111	$1.883 \times 10^{+08} \pm 7.37 \times 10^{+06}$
Specialized Vector (Vec)	33,333	$1.819 \times 10^{+08} \pm 2.66 \times 10^{+05}$

Test Bench `measure.tb`

The two above measurements (single spike transfer and container serialization) had been performed using the general `measure.tb` test bench. It generates spikes at the `UserDummy` and –depending on sub-settings– puts them into containers up to a specific size or directly calls the `swallow` method on single spikes (possibly batched). Tests are repeated with all latency measurements enabled and disabled. The tests are also be repeated with and without the packager and arqXfer features enabled.

The results used are those with the packager and arqXfer features disabled. For the

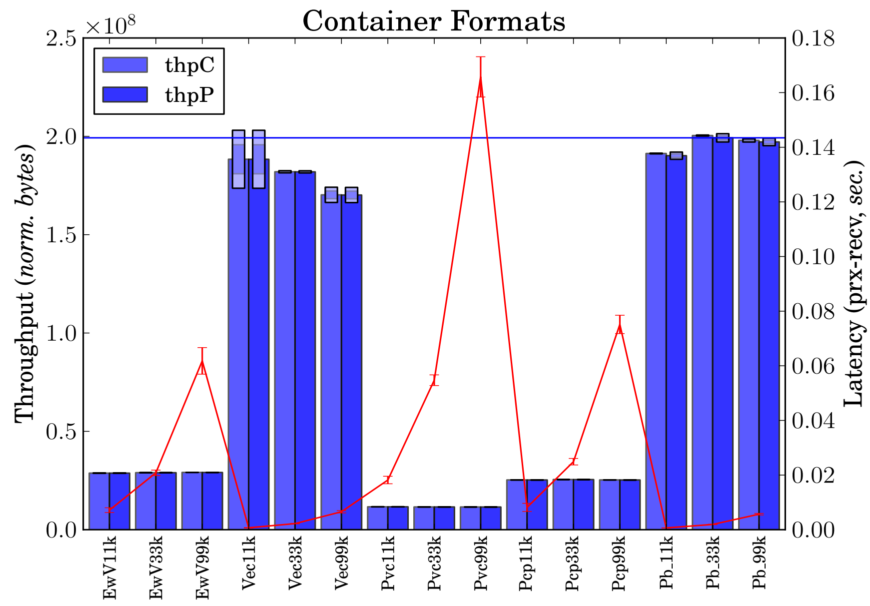


Figure 3.5.: `RCF` Throughput of different container formats. The most interesting ones are the vector (Vec) and the compressed packet (Pb). The red line plots the latency of the `RCF` transfer and the boxes represent the measured throughput. The blue line denotes the maximal throughput (thpP) measured.

throughput examination the results from the tests with disabled latency measurement were used – they report better results as the latency measurement has a negative effect on the throughput.

For more details one should look at the code which can be found alongside the other test benches in the appropriate folder: `EsterProxy/test_bench.d`.

3.3.3. Minimal Latency

As an important characteristic the minimal/best possible latency was evaluated. For this a special test case has been implemented[5] which transfers spikes (resp. smaller batches of spikes) thereby not caring for throughput at all. Figure 3.6 shows the results of this evaluation.
Minimal latency between the `UserDummy` and the `EsterProxy` was assured using two-way calls. Figure 3.4 already allows to hypothesize that two-way calls promise better latencies. This is due to the fact that a two-way call assures that it is handled completely before it returns. Therefore a subsequent call will be executed immediately instead of being stacked on top of the `RCF` dispatching queue. Secondly the packager is started with a packet timeout equal to the minimal positive value a double can hold. This leads to the immediate transfer of packets containing exactly one spike. This assures a minimal latency between the `EsterProxy` and the `ArqDrain`.

Table 3.5.: Minimal latency

Serializer	Size (*spikes*)	Latency (arq-recv, *sec.*)	ThpA (*norm. bytes*)
Compressed	1	$3.713 \times 10^{-4} \pm 4.4 \times 10^{-5}$	$3.754 \times 10^{+4} \pm 9.3 \times 10^{+1}$
Vector	10	$3.809 \times 10^{-4} \pm 4.4 \times 10^{-5}$	$3.805 \times 10^{+5} \pm 3.3 \times 10^{+3}$
Single	1	$3.826 \times 10^{-4} \pm 4.2 \times 10^{-5}$	$3.922 \times 10^{+4} \pm 8.0 \times 10^{+1}$
Vector	1	$3.913 \times 10^{-4} \pm 4.2 \times 10^{-5}$	$3.845 \times 10^{+4} \pm 7.0 \times 10^{+1}$
Vector	100	$3.936 \times 10^{-4} \pm 4.0 \times 10^{-5}$	$3.515 \times 10^{+6} \pm 4.6 \times 10^{+3}$
Compressed	10	$4.618 \times 10^{-4} \pm 5.4 \times 10^{-5}$	$3.725 \times 10^{+5} \pm 4.2 \times 10^{+3}$
Compressed	100	$5.976 \times 10^{-4} \pm 5.7 \times 10^{-5}$	$3.549 \times 10^{+6} \pm 8.8 \times 10^{+3}$

Latencies for packets of 1 k and 10 k spikes have been tested as well, but with latencies above 365 ms they are out of the interesting range and have been removed from the plot. Such higher packet sizes break the one-spike-per-packet optimization (see sentence Secondly... above). Table 3.5 shows the seven best measured latencies. The best five of these all lie within one sd of each other, i.e. they can be considered approximately equal. Interesting is that apart from the best result (which is probably due to deviation) the vector generally performs better, considering latency, than the delta-compressed container whereas the compressed packets offered a better throughput.

[5]See file `EsterProxy/test_bench.d/minlat.tb`.

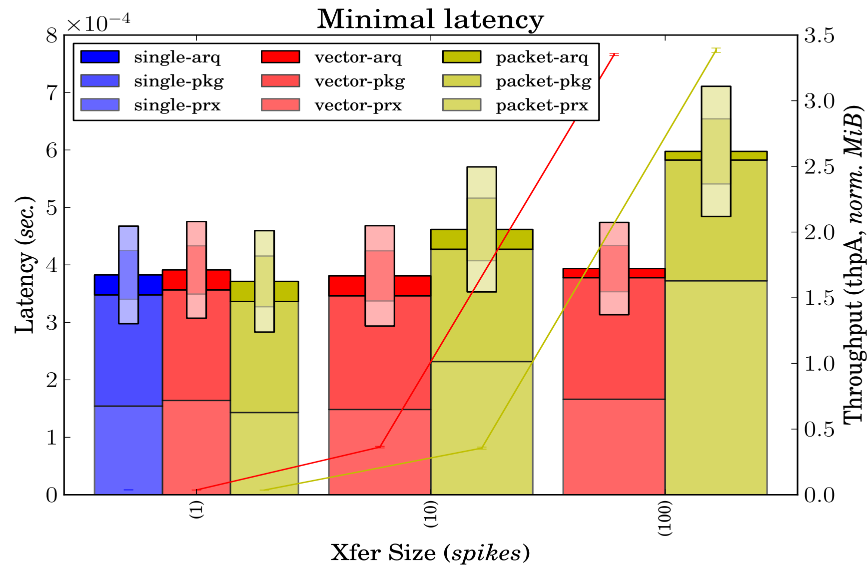


Figure 3.6.: This figure shows an evaluation of the minimal possible start-to-end latency. The boxes denote the measured latency stack, i.e. they show all measured latencies. One can see how the latency increases from measuring point to measuring point. The sd is only drawn for the most important, the arq-recv, latency. For comparison the red and yellow line show the correspondent total throughput (thpA).

3.3.4. Maximal Throughput

Another important characteristic is the maximal throughput. As this is highly dependent on the size of the transferred objects a special test bench[6] to sweep a reasonable range of object sizes (aka *XferSize*) was implemented.

About 1 GiB (norm.) of data was transferred for each test of a specific XferSize. The XferSize parameter range was swept from one to about 40 k (i.e. 143 MiB, norm.) full `fpga_pulse_packet_t` packets. After each test the XferSize was increased by a factor of 1.15. Each test was repeated 13 times with disabled latency measurement, concentrating on throughput; and one additional time to retrieve the latency indicator (arq-recv measuring point).

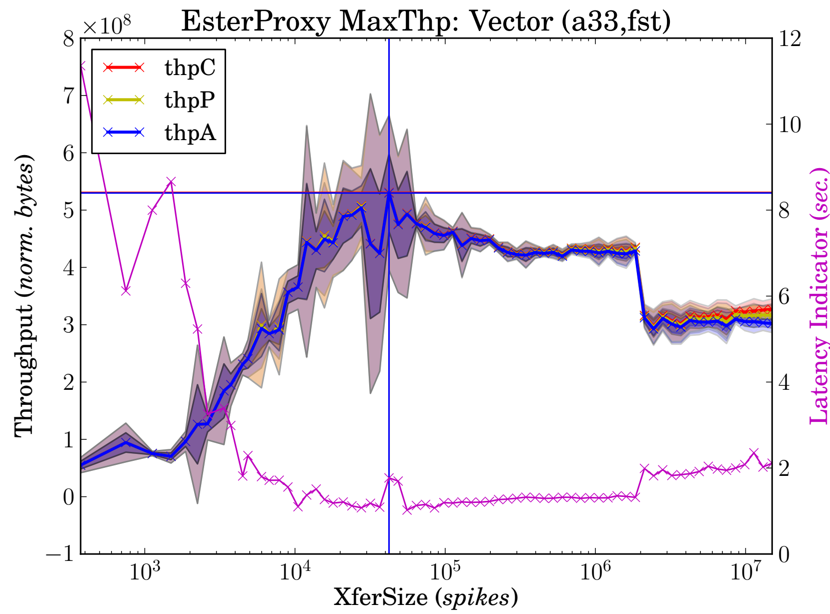
To archive a maximal throughput, the data that was transferred during these tests, was constructed before the tests started. In other tests the spikes are created during the test –shortly before they are transferred– and are initialized with a spiketime resembling the wall-clock time of their creation. Here, such a creation time point was not available due to the early preparation of the data. The spikes' spiketimes were simply increased by one for every new spike. Therefore, the latency measurement results of this test should be merely seen as an indicator. Though a higher resulting latency measure surely indicates just that, the actual value of a latency measured can not be accepted as a realistic representation – they simply do not resemble the time spent on transfer.

The packagers packet buffer timeout was specified as infinity (a special `double` value). This orders the packager to always construct full packets. But with the actual design of the packager (Design 2.7.4) a packet's timeout is still checked occasionally (though the infinite timeout will never force the transfer of an incomplete packet).

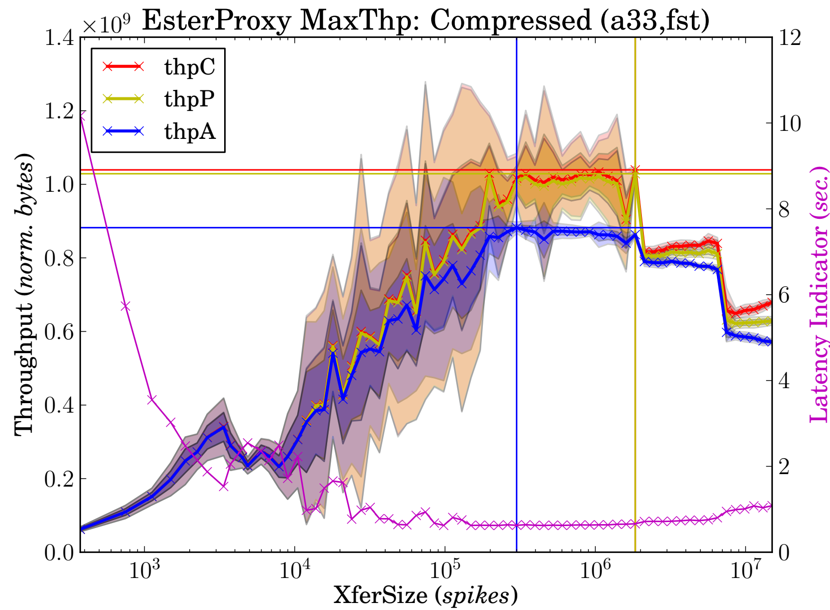
The aim of these tests was to find the "XferSize", i.e. the size of a single transfer object, which serves best a high throughput. Figures 3.7a resp. 3.7b show how the two best available container formats (section 3.3.2) performed on the fast network interface (10 GbE, *fst*) with a `ShamemIPC` buffer size of 33. As the `ShamemIPC` evaluation (section 3.2) showed no dependency between the buffer size and the throughput the 33 –as a rather small value– was chosen.

A second run using a `ShamemIPC` buffer size of 1 k was performed on all available network connections, i.e. additionally to the fast network interface the performance using a standard network interface (1 GbE, *slw*) and a localhost connection was tested. The corresponding figure 3.8 shows all results combined. The selected best results are presented in the table 3.6. One can see that the fst_a33_spk and the fst_a33_vec are fighting until an XferSize of about 2×10^4 spikes. Then the compressed packet lifts off (red with triangles) whereas the vector begins to weaken. From the range of 2×10^5 to 1×10^6 the container (spk) performs best.

[6] See file `EsterProxy/test_bench.d/maxthp-{vector,serpck}.tb`.



(a) vector



(b) compressed packet

Figure 3.7.: These figures show the maximal throughput of the complete *EsterProxy Suite* using the fast network controller and a `ShamemIPC` buffer size of 33 entries. They differ in the `RCF` serialization method used (specialized vector and compressed packet). The blue-coloured line denotes the suite's total throughput; its maximum is indicated by the blue cross. The magenta-coloured line indicates an approximation of the latency. For information on the other lines refer to section 3.3.

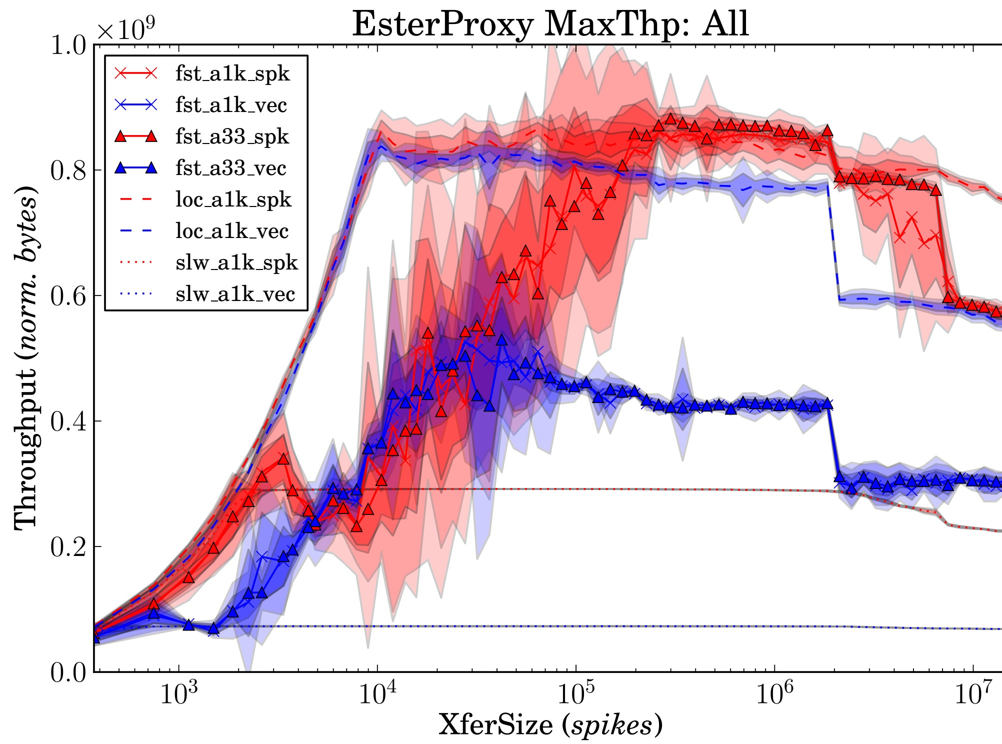


Figure 3.8.: Comparison of the best transfer methods, vector and compressed packet (*spk*), considering throughput, under various test environments, i.e. network devices: localhost (*loc*), 10 GbE (*fst*) and 1 GbE (*slw*). The best throughput reached at an XferSize of 3×10^5 spikes corresponds to 88.2 ± 0.8 MEv/s . Please note that 10 GbE saturates at 0.625 normalized gigabytes/s (wire-speed) for the uncompressed data (in blue).

Table 3.6.: Maximal Throughput (thpA) of the *EsterProxy Suite*

Label	XferSize (*spikes*)	Throughput (thpA, *norm. bytes*)	ThpA (*spikes*)
fst_a33_spk	3.00×10^5	$8.821 \times 10^8 \pm 7.78 \times 10^6$	8.82×10^7
loc_a1k_spk	6.43×10^4	$8.657 \times 10^8 \pm 9.81 \times 10^6$	8.66×10^7
fst_a1k_spk	2.61×10^5	$8.613 \times 10^8 \pm 1.43 \times 10^7$	8.61×10^7
loc_a1k_vec	1.05×10^4	$8.378 \times 10^8 \pm 1.19 \times 10^7$	8.38×10^7
fst_a33_vec	4.23×10^4	$5.298 \times 10^8 \pm 6.73 \times 10^7$	5.30×10^7
fst_a1k_vec	2.77×10^4	$5.261 \times 10^8 \pm 9.53 \times 10^7$	5.26×10^7
slw_a1k_spk	4.86×10^4	$2.917 \times 10^8 \pm 1.23 \times 10^4$	2.92×10^7
slw_a1k_vec	1.49×10^5	$7.300 \times 10^7 \pm 1.67 \times 10^3$	7.30×10^6

3.3.5. TSR-Simulated Environment

Figure 3.9 shows a test of the *EsterProxy Suite* in a `TimeSpanRnd`-simulated environment, looking for a mixture of high throughput and low latency. The code to this test can be found in the file `EsterProxy/test_bench.d/tsr.tb`.

The TSR-test tries to transfer a fixed amount of spikes per second, 20 MEv/s. These spikes are dispersed in bunches of 25 spikes (`innermostBurst`, line 13) using the `TimeSpanRnd` callback described in listing 3.2.

The timespan emitters (TSE) used in this test are configured such that they will generate on average 800 k events per second, i.e. $25 \times 800,000$ Ev/s = 20 MEv/s. The total number of spikes (initialization value of `spikesRemaining`, line 25) is chosen such that the test duration will be about 30 seconds.

```
1  // single spike handler
2  inline bool emitSpike(TimeSpanRandomizer * tsr) {
3    double realnow = tsr->getLastAction(); // NB.: if the TSR lags real
         realtime will be later already, for now this is not handled.
4
5    // calculate a spiketime representing "now"
6    tmp.spiketime = ( (realnow+pzOffset) * CYCLES_PER_SECOND );
7
8    if (first) { // first spike in the actual transfer object
9      xferWallTime = realnow + batchTimeout; // transfer timeout
10     first = false;
11   }
12
13   int fin = (innermostBurst > spikesRemaining ? 0 : spikesRemaining -
         innermostBurst);
14   for (; spikesRemaining > fin; spikesRemaining--)
15       dser.insert(tmp);
16
17   // it the first spike in the transfer object is to old or the maximal
         reasonable object size is reached
18   if ( (realnow > xferWallTime) || (dser.getInsertCount() > batchMaxSize)
         || (!spikesRemaining) ) {
19     //std::cout << "Packet sending: " << dser.getInsertCount() << std::endl
          ;
20     client.swallow(dser); batchTransfers++; // transfer!
21     dser.clear(); // clear the object to reuse its memory thereafter
22     first = true;
23   }
24
25   return (spikesRemaining); // returns true as long as there are more
         spikes to dispatch
26 }
```

Listing 3.2: `TimeSpanRnd` Callback

Tested was the behaviour of 3 different constraints on the latency, *f, m, s*, shown in table 3.7. These constraints have been combined with three different TSE implementations, namely TSE-Static, -Gaussian, and -Burst. Each pair was then executed two times, with enabled dropping and without dropping. The `batchTimeout` specifies the maximum latency of the first spike in a transfer object thats under construction on the `UserDummy`. If this timout is reached or the batch size reaches its maximum (`batchMaxSize`, 65535 spikes), a transfer is initiated, see line 18ff. TSE-Static disperses its events equally distributed (static distance) and TSE-Gaussian uses, as expected, a Gaussian distribution. Figure 3.10 shows a schematic of the TSE-Burst output.

We can see clearly that the settings with higher constraints produce a lower latency. Throughput is –per definition– not effected. With the static and Gaussian distribution this worked quite well and no dropping was necessary. However, the burst distribution shows another picture. Here without dropping the *m* and *f* test failed. Spikes could not be transferred in due time and the packager algorithm failed. When the packager lags it happens that the successive spikes received are already to late → and the packager

Table 3.7.: TSR-test constraints

Short	Name	batchTimeout	packagerTimeout
f	fast	0.7 ms	5.0 ms
m	medium	1.0 ms	7.0 ms
s	slow	2.0 ms	`infinity`

begins transferring single spikes to the `ArqDrain`. This, of course, makes the situation even worse.[7] Without dropping there is no way out of this vicious circle. But with dropping enabled we can see that the constraints can be met again. And less than 1.5 % of the spikes needed to be dropped. With dropping enabled, if the first spike of a packet is already late, it is dropped immediately, thus breaking out of mentioned vicious circle.

[7] These tests could not even achieve the defined throughput, as spikes piled up in the `EsterProxy`'s spike-queue.

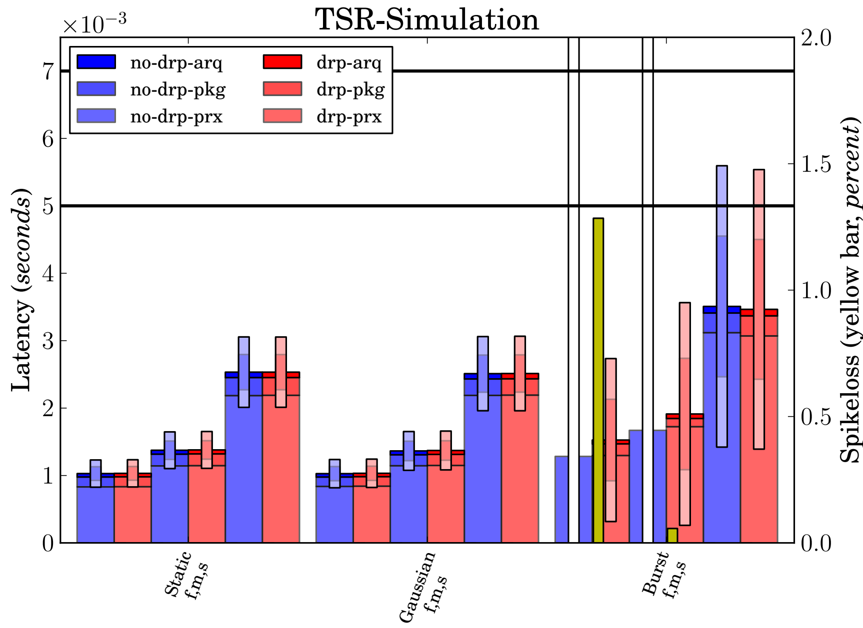


Figure 3.9.: Usability test with a TSR-simulated environment. In this test the `EsterProxy` *Suite* is configured such that it tries to satisfy specific latency constraints. These are denoted by the horizontal black lines. The upper one is for the m test and the lower one for the f test. The third test s does not use any latency constraints. The 3 groups of bars consist each of 3×2 bars, representing the reached latencies of the tests f, m, s with disabled (blue) and enabled (red) dropping. The latency shows the start-to-end latency (arq-recv). The yellow bar denotes the percentage of dropped spikes.

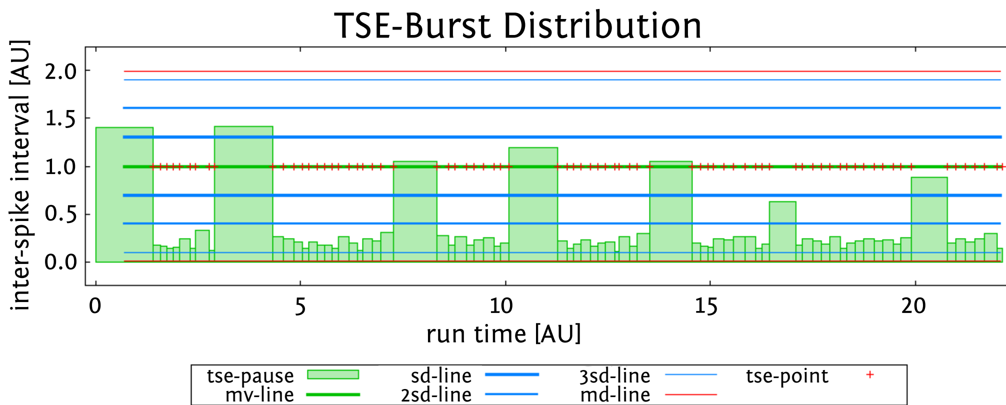


Figure 3.10.: This figure presents a possible output of the Burst TimeSpanEmitter implementation. The bigger the green boxes the longer the pause between to events.

4. Conclusion - Outlook

The final results,

maximal throughput $\rightarrow 88.2 \pm 0.8\,\text{MEv/s}$ (spikes, cf. table 3.6 and figure 3.8)

minimal latency $\rightarrow 371 \pm 44\,\mu\text{s}$ (start-to-end latency, cf. table 3.5)

show that the planned throughput –saturating 10 GbE– was reached for uncompressed data. Throughput results for compressed data easily surpass the raw wire-speed by up to 40%.

The following sections list various outlooks, considerations and conclusions that have been derived from the evaluation (chapter Discussion 3) and are concerning the software part.

4.1. ShamemIPC Pointer-Access Method

Just shortly the `ShamemIPC` has been extended by a pointer access method. But it should be considered a draft only.

Apart from not relying on a copy it has the advantage of the reader being able to change the underlying object. This will be useful when connected to the RDMA-NICs. The `ArqDrain` can such change the ARQ `ShamemIPC` buffer entry (`arq_shamem_t`) before it passes a pointer to it to the RDMA-NIC. With the memcpy method a copy of the `arq_shamem_t` object needs to be made, changed and passed to the RDMA-NIC. This additional copy can be spared using reader-side pointer access. The `arq_shamem_t` contains uninitialized memory for the ARQ header and it is the `ArqDrain`'s responsibility to assign its contents.

As the RDMA-NIC is able to handle more than one ARQ packet at a time it should be possible to pass multiple `arq_shamem_t` objects/pointers to the NIC. Using the memcpy access method this is not a problem, but with the pointer access and the actual `ShamemIPC` design offering no read-ahead – we simply can't.

So a read-ahead method should be added to the `ShamemIPC` read head. Fortunately with the `ShamemIPC` design being as it is, this is a simple task. The reader only needs to delay the update of its read head's position in the shared memory and to introduce a new local variable holding the actual position. Thus the writer can't see how far the reader has advanced and will not overwrite the elements which are still used by the reader resp. the NIC.

The pointer access method could return a C++11 `std::shared_ptr` and hold internally another – actually a FIFO queue of all shared pointers that are still accessible from the

outside. The tail of this queue points at the actual position of the read head. Whenever a shared pointer at the tail becomes unique (i.e. no other references left) it can be released. Note that *releasing* in this context means only that the read head can move on; the shared pointer's deleter must be specialized as the shared memory it points to must not actually be released. The tail of the queue can, for example, be checked whenever the read head is accessed.

Alternatively this queue could also be incorporated into the specialized deleter method. It must then hold `std::weak_ptr` objects. As soon as the last shared pointer to a specific position goes out of scope its deleter is called. The deleter must then check the queue and remove all weak pointers at the queue's tail which have been released. It must also update the read head position in the shared memory accordingly.

The second variant looks cuter. But it also is more complicated as it needs synchronization of the queue. Also it would forbid any update of the read head's position other than through the deleter method. Finally note that the same design is applicable for the write head as well.

It is planned to add this access method during the next weeks.[1]

4.2. Real ARQ Drain

To change the `ArqDrain` into a program that really transfers its data to the RDMA-NIC (i.e. to the NP's FPGA), one should at first implement the `ShamemIPC` pointer access method described in section 4.1.

The `ArqDrain` must access the `arq_shamem_t` objects from the shared memory and fill in the ARQ header. The header now encodes, inter alia, the packet size (i.e. a length field). Afterwards it must set a readiness field, indicating to the NIC that the data is ready for transfer.

The `ShamemIPC` shared buffer is packed such that the (address) distance from `arq_shamem_t` to `arq_shamem_t` is exactly equal; this is also valid for all elements therein. So the address of the first buffer entry and its size, as well as the inner offsets of the readiness, length and Ethernet payload fields should be sufficient for the RDMA-NIC to do its job. The total buffer size must be known to the NIC as well, of course.

With the pointer access method (Outlook 4.1) implemented, the `ArqDrain` now waits until the NIC is done with a transfer. Finally, it must release the shared pointer to the correspondent buffer entry, indicating to the read head that it can move on (releasing its position to the write head).

4.3. Multiple Packagers – Multiple FPGAs

The `EsterProxy` is designed such that multiple FPGAs could be addressed by using multiple packagers. One packager per FPGA is necessary, as we cannot merge spikes for different FPGAs into one `fpga_pulse_packet_t`. However, multiple packagers can use

[1] As I am personally quite interested in this I will contribute it in my free time ahead.

the same ShamemIPC instance. The packager's linkArqShamem() method allows to pass a pointer to a mutex along with the ShamemIPC pointer. If this mutex pointer is not a null-pointer it is used to synchronize the access to the ShamemIPC.

Apart from the speed this design works flawlessly using the single-spike handler[2] on the receiving EsterProxy. Dependent on the label it can decide into which packager's queue the spike should be inserted. For this a routing map must be specified (label-to-packager).

Problem with this design is that the RCF swallow() method then needs to loop through the received vector and copy it element-wise. This is contradictory to the design decision to spend as few time as possible in the RCF servant thread. Also it is contradictory to the effort put into the no-copy (std::move) design.

This problem should be considered when implementing a final RCF serializable container format, see Outlook 4.6.

4.4. Latency-Dependent Packaging

At the moment the packager packs spikes into the fpga_pulse_packet_t packet until it is full or its first spike reaches a timeout. Repeating this timeout check after every single spike turned out to introduce too much overhead (because fetching the wall time is too costly). So the packet timeout is checked only every 32 spikes. This, however, is not a final solution. I can see two possible solutions that should be considered:

1. Check the timeout only when the spikes-queue is empty. If after packaging the packet is too old – just drop it.
2. Implement an additional thread which takes care of getting the wall time and offers that data cheaply to the packager.

When using the second possibility it should be practical to let the clock thread calculate a clock cycle value that can directly be compared to a spike's spiketime.

4.5. C++11-ify

In the introduction to chapter Design (see page 8) it was mentioned that the code of the caipc project uses mixed standards. The complete code should eventually undergo a revision with only the harmonization in mind; thereby std::shared_ptr and other C++11 features should be introduced where they are applicable. Also the wall time was read using gettimeofday(&tim, 0 /*NULL*/); and passed around as a double value. The use of the std::chrono library (e.g., std::chrono::duration and std::chrono::steady_clock) should be considered.

[2]See the vector-spike handler, which is analogous, at section 2.7.2.

4.6. Container Format

The best developed container for spike transfer between the `UserDummy` and the `EsterProxy` was the Dummy Event Serializer class (`FpgaDummyEventSerializer`).[3]

It tries to minimise memory usage by delta-compressing spikes upon insertion and saving only that packed version of the data. However, as the FIFO queue holding spikes for the `SpikePackager` (`SpikesQueue`) and the `SpikePackager` expect vectors of `fpga_pulse_packet_t`s the container is completely decompressed and changed into such a vector upon deserialization. During the decompressing loop each spike must be touched, loaded into CPU memory and finally copied into a new vector. Now each spike uses its full 12 bytes.

As the packager must loop through the data again, performing similar calculations, the above loop could be spared if the data is put compressed into the `SpikesQueue` and decompressed just upon removal from the queue.

So, what should be done is to

1. implement a FIFO spikes-queue container which compresses its contents upon insertion and decompresses upon removal
2. serialization and deserialization must be handled byte-wise and should not access the spikes individually
3. the `SpikePackager` must be revised to work with such a container

The container can unfortunately not be simply build upon a `std::queue` of delta-compressed data as the queue offers no access to its internal data. It could be implemented as a vector of pointers to fixed size arrays of delta-compressed data. Upon insertion a spike is compressed and put into the array at the back of the vector. If the array is full a new one is added to the vector. The serializing process can now loop through the vector and byte-wise serialize the arrays. The deserialization could optionally put all data into a single array as it already knows the expected size.

4.7. `ShamemIPC` Initialization

The `ShamemIPC` initialization is done by the `ARQ_Drain` program. The buffer size is setup once when the `ArqDrain` is started. This means that multiple test cases can not differ in the `ShamemIPC` buffer size during one run of the *`EsterProxy` Suite*. This makes buffer size sweeps needlessly complicated.

It should be a simple revision to change the `ArqDrain` such that it initializes a new shared memory per test case. The ID and buffer size of this `ShamemIPC` should be specified by the `EsterProxy`.

[3]Note that the `Fpga` in its class name is a misnomer as this packet is never transported to the FPGA itself.

If the `ShamemIPC` is not initialized during the `EsterProxy` setup, but during test cases, it would also allow us to start tests without the `ArqDrain` running. This, e.g., would be convenient for `RCF` serialization tests as these do not necessarily need the `ArqDrain`.

The Discussion 3.2 showed that the buffer size has no effect on the `ShamemIPC` throughput but if the `ArqDrain` is really connected to a NIC this will look different. Especially when the packager is used with packet dropping enabled (see Design 2.7.4).

4.8. Sizeof Pitfall / Packed Attribute

For the calculation of normalized bytes the `sizeof` operator was used:

```
normBytes = sizeof(fpga_dummy_event_t) * spikesCount;
```

During development phase `sizeof(fpga_dummy_event_t)` returned 12 bytes which was considered a fair value for the evaluation. Unfortunately on the CP cluster a `fpga_dummy_event_t` uses 16 bytes. When this was observed all tests had been performed. The size of a packed `fpga_dummy_event_t` (i.e. the actual bytes used) should be a better factor. It is 10 bytes.

Fortunately this could be fixed as it is a simple static factor. All such normalized data has been revised by multiplying it by 10/16 to get, e.g., a throughput normalized on 10 bytes per spike (i.e. transferred `fpga_dummy_event_t`). The data files have not been touched, but the values presented in tables and plots.

The use of the `packed` attribute for the `fpga_dummy_event_t` should be tested. It is expected that this will enhance the throughput of the byte-wise (memcpy) serialization of the `RCF` throughput between the `UserDummy` and the `EsterProxy`.

```
struct fpga_dummy_event_t {
// [...]
} __attribute__ ((packed));
```

It has been added to the code and it compiles and runs (i.e. no problems with the `RCF` internals). There is a simple `#if 1` flag in `EsterProxy/rcf_xferTypes.h` which can be used to enable/disable this feature.

4.9. `ShamemIPC` Throughput

The results of the `ShamemIPC` evaluation (Discussion 3.2) are not as good as the *HMF Transmitter* evaluation promised. This is probably due to the additional code introduced to make a fully-fledged library for a templated shared-memory circular buffer out of the singular programs. But it could also originate in the buffer entry used for the evaluation. An evaluation using raw and simple data (e.g., without compression) should be executed to retrieve actual unbiased raw data throughput. Then the esp. the `accessShamem()` method should be revised.

4.10. Operating System Support

The measurements were performed on the CP Cluster running *Debian Wheezy*'s default Linux kernel, i.e. without any added real-time functionality. However, latency results in the range of the operating system's scheduler latency[4] indicate that real-time extensions, e.g. Linux' `CONFIG_PREEMPT_RT`(*Real-Time Linux*, 2012) or Xenomai(*Xenomai*, 2012), could further decrease the average latency and latency jitter.

4.11. Network Protocol

All test were performed using the TCP/IP protocol. If, for `CL-Experiment`s, spike loss is acceptable and programmatic dropping is used the User Datagram Protocol (UDP) protocol could as well be used. For parameter-sweep experiments, though, the TCP/IP protocol should be preferred. Losing data during config phase of a parameter-sweep trial is suboptimal.

[4] cf. Linux `latencytop`, `http://www.latencytop.org`

Appendices

A. Caipc/ Code Package

A.1. Repository

The code to this thesis can be found in the `caipc` repository. It is accessible through `gitviz.kip.uni-heidelberg.de:caipc`. To compile the *EsterProxy Suite* and its accessory parts it has to be checked out into the `symap2ic` components directory. Various other parts/repositories have to be checked out alongside `caipc`, too.

```
1 git clone git@gitviz.kip.uni-heidelberg.de:symap2ic.git
2 cd symap2ic/components
3 git clone git@gitviz.kip.uni-heidelberg.de:caipc.git
4 git clone git@gitviz.kip.uni-heidelberg.de:{halbe,hicann-system,etc.}.git
5 cd caipc
6 CXX=g++-4.7 waf configure
```

The `waf configure` command will announce other missing libraries.

A.1.1. Caipc Directory Environment

The following directory tree outlines the caipc environment.

```
symap2ic ........................................ build support
└─ components
   ├─ caipc
   │  ├─ EsterProxy ........................ EsterProxy Evaluation Suite
   │  │  └─ contents ........................ see below
   │  ├─ TimeSpanRnd ........................ Time Span Randomizer
   │  ├─ ShamemIPC ......... Shared Memory Inter-Process Communication Library
   │  ├─ TestApp ............................ smaller test applications
   │  ├─ Utilities .............. various utilities like the MeanCalculator
   │  ├─ dat ................................ results are placed here
   │  ├─ lib ................................ build libraries are linked here
   │  ├─ build ............ this is where the compiled files will be put
   │  └─ wscript ................. build script of the caipc repository
   ├─ halbe ................................. software backend
   ├─ hicann-system .......... hardware near backend: fpga_pulse_packet_t
   ├─ lib-boost-patches
   ├─ lib-rcf ............... RCF library with our minor adjustments
   │  └─ eval .......................... RCF Serialization Evaluation
   ├─ pyhmf
   └─ ztl
```

A.1.2. EsterProxy Directory

```
EsterProxy ....................................... EsterProxy Evaluation Suite
├── UserDummy.cpp
├── EsterProxy.cpp
├── ArqDrain.cpp
├── ShamemThroughput.cpp
├── common.h
├── doubleSidedMutexedQueue.h
├── esterProxyTest.h
├── ipc_interface.d
├── ipc_xferTypes.h
├── rcf_interface-base.h
├── rcf_interface.h
├── rcf_Servant.{h,cpp}
├── rcf_support.h
├── rcf_xferTypes.h
├── spikePackager.{h,cpp}
├── test_bench.d
│   ├── onerun.tbb
│   └── *.tb
└── wscript
```

A.2. RCF Calls

The following gives an overview of the EsterProxy's RCF interface. The interface is defined in the files EsterProxy/{rcf_interface.h,rcf_interface-base.h}. The implementation can be found in the servant class: EsterProxy/rcf_Servant.{h,cpp}.

An RCF interface method description looks basically like this:

```
RCF_METHOD_<retS><paramCnt>(returnType, methodName, paramTypes [,..])
// retS: R returns something, V returns void
// returnType: void if retS is V, otherwise a serializable type
// methodName: name of implementing method
// An example:
RCF_METHOD_V3(void, prepareTest, const std::string &, const bool &, const
    int &)
// And the declaration:
void prepareTest(const std::string & testName, const bool & enableMeasure,
    const int & spikes);
```

establishArqConnection connect with the ShamemIPC resp. the ArqDrain

setTestTimeStamp set time stamp on the EsterProxy

getTestTimeStamp get last set time stamp

enableReceiveDropping enable dropping of late spikes before they are even enqueued
specify the seconds (as a double) of allowed "packet latency"

enablePackaging enable the packaging feature
specify if the packaging latency should be measured and the maximum first-spike latency before a packet must be send

enableArqXfer enable arqXfer feature
specify if the `ArqDrain` should measure latency and if `ArqDrain` dropping should be enabled (drops packets if the `ShamemIPC` blocks the transfer of a ready packet).

prepareTest prepare a new test on the proxy, enters config state.

startupTest plans a test (it will be started shortly thereafter)
specify a test name and the expected spikes
returns the `EsterProxy`-side point zero at which the `UserDummy` is allowed to start transferring data

shutdownTest measure the `EsterProxy` duration (=> thpP)
then indicate to the packager that the test is over

finalizeTest returns false until the `ArqDrain` has submitted it's results to the `EsterProxy`

fetchTestResult returns a test result object (see Design 2.5)

swallow swallow dummy event data and pass it on to handleSpike resp. handleVector.

- fpga_dummy_event_t (transfers a single dummy event)
- std::vector<fpga_dummy_event_t> (good speed)
- FpgaDummyEventSerializer (best speed)
- FpgaDummyEventVector1 (deprecated)
- FpgaDummyEventVector2 (deprecated)

`ArqDrain` **interface** this part of the interface should be used by the `ArqDrain` only:

checkArqWakeup checks for a flag indicating that the `ArqDrain` should wake up
this flag is set by a `UserDummy` call to `enableArqXfer()`
returns true if the flag is set.

getArqConfig after been woken up the arq requests its configuration data
get the testname, if latency should be measured and how many spikes are expected

submitArqResults submits all possibly interesting test results to the `EsterProxy` (see Design 2.5)

Basic interface This part of the interface is available to both, the `UserDummy` and the `ArqDrain`

echo passes very simple commands to the `EsterProxy` as strings
returns a string as an answer, an example: `"ping" --> "pong"`

getServerTime returns the `EsterProxy`'s wall time as a double (gettimeofday).

getPointZero returns either 0 or the point zero of a planned/started test.

A.3. Evaluation Data

All plots have been build using the Python Matplotlib. The data files as well as the plot decriptions can be found in the `bachelorthesis-khusmann` repository. As the other repositories it is available under the url `gitviz.kip.uni-heidelberg.de:bachelorthesis-khusmann`.

I do not list all data files content's here – for these one should look into the repository. The following just shows the data ant the plot files available.

```
dat/
 ├─ maxthp-fst-a1k-{spk,vec}.dat
 ├─ maxthp-fst-a33-{spk,vec}.dat
 ├─ maxthp-loc-a1k-{spk,vec}.dat
 ├─ maxthp-slw-a1k-{spk,vec}.dat
 ├─ measure-full-lat.dat
 ├─ measure-full-thp.dat
 ├─ measure-sgl_mw-{lat,thp}.dat
 ├─ measure-vec_ew-{lat,thp}.dat
 ├─ minlat.dat
 ├─ rcf/
 │   ├─ testBOOST-1.dat
 │   ├─ testBOOST-*.dat
 │   ├─ testBOOST-9.dat
 │   ├─ testSF-1.dat
 │   ├─ testSF-*.dat
 │   └─ testSF-9.dat
 ├─ shamemthp-memcpy.dat
 ├─ shamemthp-memcpy-reader.dat
 ├─ shamemthp-pointer.dat
 ├─ shamemthp-pointer-reader.dat
 ├─ tsr.dat
 ├─ wireshark_ACK_latency.csv
 ├─ wireshark_ACK_latency.py
 ├─ wireshark_all.csv
 └─ wireshark_all_in_range.csv
```

```
plot/
├── matrix2latex -> ../../tools/matrix2latex
├── maxthp-all.py
├── maxthp-detail.py
├── methods.py
├── minlat.py
├── rcfThroughput.py
├── sglxfer.py
├── shmthp.py
├── tsr.py
├── vecxfer.py
└── wscript_build
```

B. HMF Host-FPGA Communication

B.1. Current State

Figure B.1 depicts a *RAM test* in terms of network communication between host computer and FPGA on the HMF wafer-scale system. Packets are shown in different colours to mark packets to the FPGA in orange and packets from the FPGA in white. The strictly alternating colours and the small packet sizes –Ethernet supports 1500 bytes[1]– are an indication of the very simple transport layer protocol.

File Edit View Go Capture Analyze Statistics Telephony Tools Internals Help

Filter: (ip.addr eq 192.168.1.10 and ip.addr eq 192.16 Expression... Clear Apply

No.	Time	Source	Destination	Protocol	Length	Info
516	0.000027	192.168.1.10	192.168.1.21	UDP	250	Source port: 1702 Destination port: 1702
517	0.000260	192.168.1.21	192.168.1.10	UDP	60	Source port: 1702 Destination port: 1702
518	0.000059	192.168.1.10	192.168.1.21	UDP	150	Source port: 1702 Destination port: 1702
519	0.000144	192.168.1.21	192.168.1.10	UDP	60	Source port: 1702 Destination port: 1702
520	0.000018	192.168.1.10	192.168.1.21	UDP	74	Source port: 1702 Destination port: 1702
521	0.000061	192.168.1.21	192.168.1.10	UDP	60	Source port: 1702 Destination port: 1702
522	0.000026	192.168.1.10	192.168.1.21	UDP	266	Source port: 1702 Destination port: 1702
523	0.000274	192.168.1.21	192.168.1.10	UDP	60	Source port: 1702 Destination port: 1702
524	0.000021	192.168.1.10	192.168.1.21	UDP	150	Source port: 1702 Destination port: 1702
525	0.000144	192.168.1.21	192.168.1.10	UDP	60	Source port: 1702 Destination port: 1702
526	0.000016	192.168.1.10	192.168.1.21	UDP	50	Source port: 1702 Destination port: 1702
527	0.000048	192.168.1.21	192.168.1.10	UDP	60	Source port: 1702 Destination port: 1702
528	0.000021	192.168.1.10	192.168.1.21	UDP	150	Source port: 1702 Destination port: 1702
529	0.000144	192.168.1.21	192.168.1.10	UDP	60	Source port: 1702 Destination port: 1702
530	0.000017	192.168.1.10	192.168.1.21	UDP	74	Source port: 1702 Destination port: 1702
531	0.000061	192.168.1.21	192.168.1.10	UDP	60	Source port: 1702 Destination port: 1702
532	0.000027	192.168.1.10	192.168.1.21	UDP	250	Source port: 1702 Destination port: 1702
533	0.000259	192.168.1.21	192.168.1.10	UDP	60	Source port: 1702 Destination port: 1702
534	0.000164	192.168.1.10	192.168.1.21	UDP	150	Source port: 1702 Destination port: 1702
535	0.000146	192.168.1.21	192.168.1.10	UDP	60	Source port: 1702 Destination port: 1702
536	0.000020	192.168.1.10	192.168.1.21	UDP	74	Source port: 1702 Destination port: 1702

```
▷ Frame 519: 60 bytes on wire (480 bits), 60 bytes captured (480 bits)
▷ Ethernet II, Src: a3:97:a2:55:53:be (a3:97:a2:55:53:be), Dst: 00:1b:21:d4:7c:d7 (00:1b:21:d4:7c:d7)
▷ Internet Protocol Version 4, Src: 192.168.1.21 (192.168.1.21), Dst: 192.168.1.10 (192.168.1.10)
▷ User Datagram Protocol, Src Port: 1702 (1702), Dst Port: 1702 (1702)
▷ Data (4 bytes)

0000  00 1b 21 d4 7c d7 a3 97  a2 55 53 be 08 00 45 00   ..!.|... .US...E.
0010  00 20 00 fd 40 00 40 11  b6 60 c0 a8 01 15 c0 a8   . ..@.@. .`......
0020  01 0a 06 a6 06 a6 00 0c  62 e7 0c 33 00 00 00 00   ........ b..3....
0030  00 00 00 00 00 00 00 00  00 00 00 00               ........ ....
```

File: "/tmp/wireshark_eth1_20121214144825_cRSqzK" 286 KB 00:00:16 P... Profile: Default

Figure B.1.: A screenshot of *Wireshark* (*Orebaugh et al.*, 2006) displaying the network dump of a simple *RAM test* on the HMF wafer-scale system: Data packets originating from the host computer (IPv4 address: `192.168.1.10`, highlighted in orange) get answered by the FPGA `192.168.1.21`, white).

[1] up to 16KiB if *Jumbo* frames are supported

Figure B.2 plots raw data throughput versus wall-clock time. The average throughput of 1.1 MB/s is several orders of magnitude lower than the specified maximum throughput. This is due to the fact that all communication in the HMF wafer-scale system is currently based on Standard Test Access Port and Boundary-Scan Architecture (JTAG). Simply put, JTAG commands are encapsulated into Ethernet/UDP frames, transported to the FPGA and executed on the FPGA. If execution fails, a timeout on the host computer triggers a retransfer of the command, thus providing some kind of *transport layer* protocol (ISO/IEC 7498-1:1994). However, the JTAG standard is designed for easy test access but not high-throughput or low-latency applications.

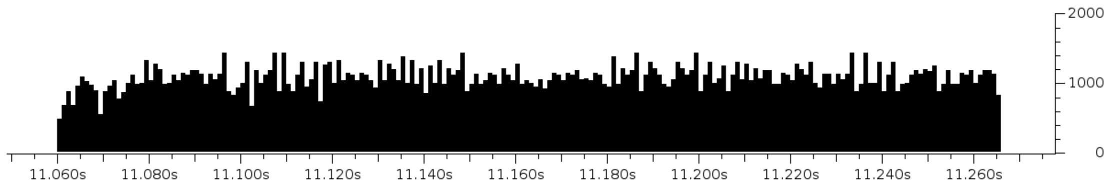


Figure B.2.: I/O graph of data transfers between host computer and a single HICANN/ chip on the HMF Wafer-scale system: The displayed section is part of a longer-running *RAM test*. At the time of writing, the current system still uses the JTAG (*IEEE*, 2001) test access interface for all communication. The test was selected due to the homogeneous access pattern (write, read, write, ...) and the neglectable on-chip latency (register access). Average throughput: 1.1 MB/s for homogeneous access pattern (i.e. from 11.080 s – 11.260 s); average access latency: 186.6 μs ± 9.3 μs.

B.2. Current Work and Future Improvements

At the time of writing, it is expected that the communication channel between FPGA and HICANN will migrate to non-JTAG-based access at the end of 2012. This will mark a first step to higher performance as then the host computer may *batch* several commands into single Ethernet frames, thus reducing computational and protocol overhead. The second step will follow in the first half of 2013 (planned) when the communication channel between host computer and FPGA will migrate to ARQ.

B.3. ARQ

State of the art transport layer protocols –e.g., the Transmission Control Protocol (TCP) (*Braden*, 1989)– employ the Automatic Repeat reQuest (ARQ) (*Fairhurst*, 2002) error-control method to automatically re-transmit dropped data.

The implementation developed in the Vision(s) Group features variable-length packets and a sliding window algorithm[2]. FPGA and software implementation is described in (*Philipp*, 2008; *Schilling*, 2010).

C. Acronyms

C.1. Technical Terms

ADC Analog-to-Digital Converter

AdEx Adaptive Exponential Integrate-and-Fire

ARQ Automatic Repeat reQuest

COTS Commercial Off-The-Shelf

FIFO First In – First Out

FPGA Field Programmable Gate Array

IPC Inter-Process Communication

JTAG Standard Test Access Port and Boundary-Scan Architecture

KVM Keyboard, Video and Mouse

NIC Network Interface Controller

NTP Network Time Protocol

PCB Printed Circuit Board

RAII Resource-Aquisition-is-Initialization

RDMA Remote Direct Memory Access

RPC Remote Procedure Call

TCP Transmission Control Protocol

TMP Template Meta-Programming

ToR Top-of-Rack

UDP User Datagram Protocol

VLSI Very-Large-Scale Integration

WSI Wafer-Scale Integration

C++	C++ Programming Language
C++03	C++03 Standard
C++11	C++11 Standard
g++	GNU Compiler Collection
KiB	kibibyte: in this document a KiB is defined as $2^{10} = 1024$ bytes
MiB	mebibyte: in this document a MiB is defined as 2^{10} KiB
GiB	gibibyte: in this document a KiB is defined as 2^{10} MiB
mv	mean value
sd	standard deviation
shamem	shared-memory area
RDMA-NIC	RDMA-capable NIC

C.2. Thesis Projects

***EsterProxy* Suite**	*EsterProxy Evaluation Suite*: consisting of the programs `EsterProxy`, `UserDummy`, `ArqDrain`
`TimeSpanRnd`	Time Span Randomizer
`TSE`	Time Span Emitter
`ShamemIPC`	Shared Memory Inter-Process Communication Library
`RCFSerEval`	RCF Serialization Evaluation
`EsterProxy`	`EsterProxy` program: The Core
`UserDummy`	`UserDummy` program: The Driver
`ArqDrain`	`ARQ_Drain` program: The Endpoint
`RcfServant`	`RCF` Servant object
`fpga_dummy_event_t`	Dummy Spike Data struct
`SpikesQueue`	FIFO queue holding spikes for the `SpikePackager`
`SpikePackager`	FPGA Spike Packager class
`arq_shamem_t`	ARQ `ShamemIPC` buffer entry
`FpgaDummyEventSerializer`	Dummy Event Serializer class

C.3. External Libraries

BOOST	BOOST C++ Libraries: A collection of C++ libraries for general use, website: http://www.boost.org
BOOST::Interprocess	BOOST Interprocess Library
BOOST **Serialization**	BOOST Serialization Library
protobuf	Google Protobuf Library: Another serialization libray, website: http://code.google.com/p/protobuf
RCF	Remote Call Framework by Delta V Software: RCF is an IPC/RPC framework tailored for C++ applications, website: http://www.deltavsoft.com
RCF-SF	RCF Serialization Framework
RCF-BOOST	RCF BOOST Serialization
RCF-PROTO	RCF Google Protobuf Serialization
waf	Waf build tool: A flexible as well as powerfull tool for building and compiling code in general, website: http://code.google.com/p/waf
wscript	Waf build script
libev	Libev Event Library: A library for dispatching timed events, website: http://software.schmorp.de/pkg/libev.html

C.4. Superordinate Project

Kirchhoff-Institute	Kirchhoff-Institut für Physik
Vision(s) Group	Electronic Vision(s) Group
HMF	Hybrid Multiscale Facility
NP	HMF Neuromorphic Part
CP	HMF Conventional Part
HICANN	High Input Count Analog Neural Network
mainPCB	Main PCB
CL-Experiment	Closed Loop Experiment
fpga_pulse_packet_t	FPGA Pulse Packet

D. List of Figures

1.1. Schematic of the wafer-scale system 3

2.1. Schematic of the `EsterProxy` *Evaluation Suite* 8
2.2. The `arq_shamem_t` struct 23
2.3. Schematic of EsterProxy Logic 24
2.4. Schematic of EsterProxy Internals 29

3.1. `RCF` Throughput 33
3.2. Throughput of `ShamemIPC` 37
3.3. Measure Points 39
3.4. Single Spike Serialization 40
3.5. Container Serialization 42
3.6. Minimal Latency of the `EsterProxy` *Suite* 44
3.7. Maximal Throughput of the `EsterProxy` *Suite* – details 46
3.8. Maximal Throughput of the `EsterProxy` *Suite* – overview 47
3.9. TSR-Simulation 51
3.10. TSE-Burst 51

B.1. RAM test on the HMF wafer scale system 64
B.2. I/O host computer to single HICANN 65

E. Bibliography

IEEE Standard Test Access Port and Boundary-Scan Architecture, *IEEE Std 1149.1-2001*, pp. i –200, doi:10.1109/IEEESTD.2001.92950, 2001.

Braden, R. T., RFC 1122: Requirements for Internet hosts — communication layers, 1989.

BrainScaleS, Research, `http://brainscales.kip.uni-heidelberg.de/public/index.html`, 2012.

Brette, R., and W. Gerstner, Adaptive exponential integrate-and-fire model as an effective description of neuronal activity, *J. Neurophysiol.*, *94*, 3637 – 3642, doi:NA, 2005.

Brette, R., et al., Simulation of networks of spiking neurons: A review of tools and strategies, *Journal of Computational Neuroscience*, *23*(3), 349–398, 2007.

Brüderle, D., et al., A comprehensive workflow for general-purpose neural modeling with highly configurable neuromorphic hardware systems, *Biological Cybernetics*, *104*, 263–296, 2011.

Davison, A. P., Automated capture of experiment context for easier reproducibility in computational research, *Computing in Science and Engineering*, *14*, 48–56, 2012.

Delbrück, T., and S. C. Liu, A silicon early visual system as a model animal., *Vision Res*, *44*(17), 2083–2089, 2004.

Ehrlich, M., K. Wendt, L. Zühl, R. Schüffny, D. Brüderle, E. Müller, and B. Vogginger, A software framework for mapping neural networks to a wafer-scale neuromorphic hardware system, in *Proceedings of the Artificial Neural Networks and Intelligent Information Processing Conference (ANNIIP) 2010*, pp. 43–52, 2010.

Fairhurst, G., RFC 3366: Advice to link designers on link Automatic Repeat reQuest (ARQ), 2002.

Husmann, K., Internship – HMF transmitter, *Internship rep.*, Kirchhoff-Institut für Physik, Ruprecht-Karls-Universität Heidelberg, 2011, [Online]. Available: `http://www.kip.uni-heidelberg.de/cms/fileadmin/groups/vision/Downloads/Internship_Reports/report_khusmann.pdf`.

ISO/IEC 7498-1:1994, Information Technology — Open Systems Interconnection — Basic Reference Model: The Basic Model, *ISO/IEC 7498-1:1994*, ISO, Geneva, Switzerland, 1994.

libev, Website, http://libev.schmorp.de/, 2012a.

libev, Website – benchmarks, http://libev.schmorp.de/bench.html, 2012b.

Mead, C. A., *Analog VLSI and Neural Systems*, Addison Wesley, Reading, MA, 1989.

Mead, C. A., and M. A. Mahowald, A silicon model of early visual processing, *Neural Networks*, *1*(1), 91–97, 1988.

Millner, S., A. Grübl, K. Meier, J. Schemmel, and M.-O. Schwartz, A VLSI implementation of the adaptive exponential integrate-and-fire neuron model, in *Advances in Neural Information Processing Systems 23*, edited by J. Lafferty et al., pp. 1642–1650, 2010.

Orebaugh, A., G. Ramirez, J. Burke, and L. Pesce, *Wireshark & Ethereal Network Protocol Analyzer Toolkit (Jay Beale's Open Source Security)*, Syngress Publishing, 2006.

Philipp, S., Generic arq protocol in vhdl, *Internal FACETS documentation.*, 2008.

Real-Time Linux, Website, https://rt.wiki.kernel.org/index.php/Main_Page, 2012.

Schemmel, J., D. Brüderle, A. Grübl, M. Hock, K. Meier, and S. Millner, A wafer-scale neuromorphic hardware system for large-scale neural modeling, in *Proceedings of the 2010 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 1947–1950, 2010.

Schilling, M., A highly efficient transport layer for the connection of neuromorphic hardware systems, Diploma thesis, University of Heidelberg, HD-KIP-10-09, http://www.kip.uni-heidelberg.de/Veroeffentlichungen/details.php?id=1999, 2010.

UHEI, and TUD, Implement the FPGA firmware for routing of the layer 2 network, BrainScaleS Deliverable D3-3.1, 2011.

Wendt, K., M. Ehrlich, and R. Schüffny, GMPath - a path language for navigation, information query and modification of data graphs, in *Proceedings of the Artificial Neural Networks and Intelligent Information Processing Conference (ANNIIP) 2010*, pp. 31–42, 2010.

Xenomai, Website, http://www.xenomai.org/, 2012.

Statement of Originality (Erklärung):

I certify that this thesis, and the research to which it refers, are the product of my own work. Any ideas or quotations from the work of other people, published or otherwise, are fully acknowledged in accordance with the standard referencing practices of the discipline.

Ich versichere, daß ich diese Arbeit selbständig verfaßt und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Heidelberg, December 17, 2012

....
(signature)