IMPERIAL COLLEGE, LONDON

Interim Report

Patrick Engelbert

January 30, 2017

CONTENTS

1 Introduction

Modern datacenters are starting to struggle to balance the requirements of high capacity computation with the ease of manageability, which together with the decreasing rate of *Central Processing Unit (CPU)* improvements have caused datacenter providers to look towards *Field-Programmable Gate Array (FPGA)s* as a means of providing large scale reconfigurable computation for cloud based applications**pub:configurable-cloud-acceleration**

Traditionally, cloud-based systems require a layer of abstraction between the physical hardware and the user application, both to effectively use all of the hardware without the user noticing and to provide isolation between the applications of multiple users. In this sense, running on *FPGAs* is no different and requires a layer of abstraction between the hardware and the applications.

With *DiRCC*, the aim is to create a layer of abstraction between the *FPGA* hardware and the application that is running on top of it. To use the *FPGA* to it's fullest potential, the project is designed around small asynchronous computation nodes that are interconnected by a network to allow them to communicate and perform large and complex tasks by splitting them into many smaller manageable components that can be computed in parallel.

2 Project Description

Currently in development is the *Partial Ordered Event Triggered Systems (POETS)* project at Southampton University, Imperial College, London and other institutions. *POETS* is designed as a system to provide distributed asynchronous computing using interconnected *CPUs*. In essence, it has the same aim as *DiRCC* except to run on a *CPU* based architecture opposed to an *FPGA* based architecture.

To ease the development of *DiRCC*, much of the specification is based on the specification of *POETS*, allowing for interoperability as well as the sharing of ideas between the two projects as well as providing a better abstraction layer that allows the *application* code to be platform agnostic.

2.1 POETS Description

POETS is a software infrastructure project which uses many small *software nodes* to break apart computation of time expensive and complex tasks into many smaller computations that all run at the same time and interact with each other through small *messages*. On receiving such a *message*, a handler is executed on the receiving *software node*, potentially changing it's state and/or sending more *messages*. All of the *software nodes* have a defined set of input and output *ports* which provide a way to logically connect up multiple *software nodes* within the *application*. These *ports* are connected via *channels* which can be seen as a direct link between a *source port* on one *software node* and one *destination port* on another *software node* as shown in fig. 2.1.

In addition, there is the possibility of attaching properties to *channels* which act as fixed data that is passed to the receiving *software node*'s receive handler if the *channel* with attached properties is the one that a *message* has travelled along. For example, this would allow the receiving *software nodes* to know about sending *software node*'s ID without this data having to be sent over the network.

As the *software nodes* are directly connected via *channels* (on the logical layer, although it may use a protocol like *Transmission Control Protocol (TCP)* on the networking layer). This means that data is packaged into a *message* automatically and the application does not need to worry about the source and destination *software nodes* as these are implicit in the system design. Due to the direct logical connection between two *software nodes*, *POETS* also guarantees that any *messages* sent along an *channel* will arrive at the destination *software node* although due to the abstraction away from the hardware layer, it is unable to guarantee the order in which *messages* will arrive at the destination *software node*.

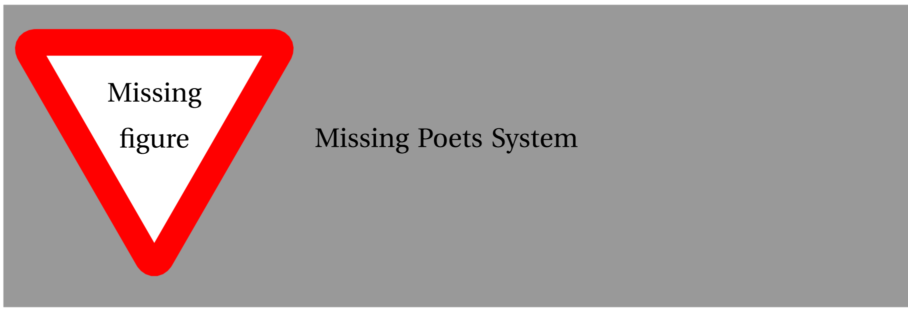

Figure 2.1: System diagram of a simple POETS application

POETS specification

The following *POETS* requirements are applicable to *DiRCC*.

Finish

- POETS Spec 1
- POETS Spec 2

2.2 *DiRCC* Project Description

DiRCC is a project that will be composed of the hardware in the form of *network nodes* and networking infrastructure, as well as the *Application Programming Interface (API)* between this hardware and the *POETS* style application that will be running on it. To maintain compatibility with the *POETS* project, *DiRCC* will be but around the same specifications, with only some minor changes to be compatible with *FPGAs*.

finish

- DiRCC Spec 1

While in the ideal scenario, the system would be infinitely expandable, due to time and resource constraints, my project will be focussing on the generation of a system running on only a single *FPGA*. However, I will be taking into account the fact that the system may have to run, split over two or more *FPGAs* when designing it.

Similar projects have been done in the past such as Splash/Splash 2**pub:1_arnold_buell_hoang_pryor_shirazi_t** or PRISM**pub:agarwal1994asynchronous** However, these projects tended to be aimed at either a very specific application, typically neural networks and machine learning or relied on custom hardware and would not be suitable for cloud based systems as shown in section 3.1. *DiRCC* however is aimed as a more generic system layer that has applications beyond those that comparative projects have, with a focus on large cloud scale systems.

In addition, with the recent completion of Project Catapult by Microsoft **pub:configurable-cloud-acceleration** in which they added a single Stratix-V *FPGA* to each of the *CPUs* for configurable computing in the cloud, it would be useful to re-purpose these *FPGAs* to act as a massive distributed network. In creating this project, the aim is ultimately allow developers to target only the *FPGAs* as the computational layer without requiring the *CPU* at all.

3 Background

3.1 Similar Research

There are multiple projects focussed around event-based massively-parallel systems, on multiple *CPUs* as well as *Graphics Processing Unit (GPU)s* and even *FPGAs*. For the most part these systems are based around spiking neural network, in other words they are designed to mimic the precesses that the human brain uses, especially the connection of neurons and their interactions.

Update previous paragraph

SpiNNaker

The inspiration for the *POETS* project is a hardware design from The University of Manchester called *SpiNNaker***pub:painkras2012spinnaker** This is a system designed to model the actions of up to a billion neurons using up to 2^{16} software nodes connected together using an asynchronous network, allowing it to perform many calculations at the same time with *messages* being passed between them. Due to the similarity between *SpiNNaker* and *DiRCC*, a large amount of the concepts of *SpiNNaker* can be used as a starting point for it. Specifically, the details of the *SpiNNaker* node and some of the routing infrastructure are of interest.

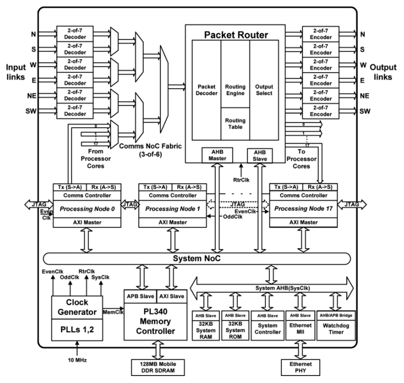

Figure 3.1: System diagram of a single *SpiNNaker* node**pub:painkras2012spinnaker**

SpiNNaker is based around a large number of *SpiNNaker* nodes that are all interconnected. Each of the nodes has two functions: they are the *CPUs* that perform the actual communication, and include a router that forms the backbone of the network infrastructure. This means that for the *SpiNNaker* system, these nodes must simply be connected together once when the hardware is built and the configuration can then be determined on the start of each program run without the hardware having to be changed. The design of the node is based around the *Globally Asynchronous, Locally Synchronous (GALS)* architecture**art:1_plana_furber_temple_khan_shi_wu_yang_2007** to simplify timing closure in the *System on a Chip (SoC)* design and allow for redundancy within the node in case of a faulty processing node.

Each *SpiNNaker* node is laid out as shown in fig. 3.1 with 6 input and output links, an input clock, connection to some off-chip *SDRAM*, an Ethernet interface to the external world and some JTAG interfaces to allow for debugging.

expand

Within each node, there are 18 ARM968 based Processor Nodes, of which one is chosen on start up as a monitor Processor Node to manage the other computing Processor Nodes and act as an interface to the external system with functions such as data aggregation. Each of these Processor Nodes is composed of an ARM9E-S integer core with attached 64KB of dual-banked data memory and 32KB of instruction memory as can e seen in fig. 3.2. In addition, it has the typical timers and interrupt controllers as well as interfaces to the Processor Node's internal network.

All of Processor Nodes within a single *SpiNNaker* node are connected to a networking infrastructure that attaches to the built-in *message* router as well as a block of on-chip RAM, on-chip boot ROM and the off-chip *SDRAM*. In addition, this networking layer connects to the interface for the Ethernet connection. The second component of each *SpiNNaker* node is the *message* router. This component receives the *messages* from all of the 6 input connections and decides into which of the output connections the *message* should travel next. Alternative, the *message* may have reached it's destination and so needs to be routed to the correct Processing Node. In addition, when a Processing Node wishes to send a *message* to a different *SpiNNaker* node, it too must travel through the router so that the correct output connection can be chosen.

The router is attached of several multiplexers to combine all of the possible input *message* sources, whether on-chip or off-chip into a single *message* stream that then feed into the input of the router. Within the router there are three pipeline stages to decode the message, decide on the route of the *message* and push the *message* into the appropriate output queue. In addition, there is a 1024 word routing table that is preloaded with the routing information at the application start**pub:navaridas2009understanding**

As can be seen in fig. 3.3, once a *message* has been decoded, there is an error checking stage that also steers the *message* to the correct routing engine for the second stage. This second stage uses either a multicast router for, algorithmic router or point to point router to determine the next step that the *message* should take**art:1_plana_furber_temple_khan_shi_wu_yang_2007** based on both the message's source address as well as the data stored in the routing table. In the case of *SpiNNaker*, the multicast router is the default choice for most *messages* with point-to-point routing used as little as possible**pub:navaridas2009understanding**

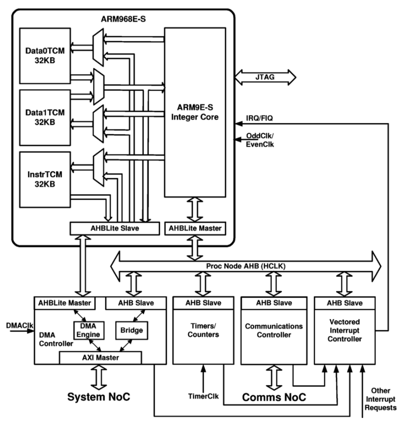

Figure 3.2: Details of a *SpiNNaker* Processor Node**pub:painkras2012spinnaker**

This is done to reduce the number of *messages* that are traversing the network, especially if a large portion of the *messages* travel the same path.

The second routing engine, the algorithmic routing, has two main modes. The first is called 'default routing'**art:1_plana_furber_temple_khan_shi_wu_yang_2007** which occurs when the source address is not in the routing table and causes the *message* to be passed to the opposite output connection, e.g. North to South. This saves space on the routing table and also reduces the complexity of the calculations required for most *messages*. As such, it is a useful feature that should be included in the routing of *messages* for my project as much as possible. The second mode for the algorithmic routing is one of neighbours, which routes the *message* to all neighbours of the processing node. As this is heavily inspired by the way that synapses communicate, it is useful to the application that *SpiNNaker* is designed for but is not too useful for my application.

The last routing engine, which is also the one most applicable to the one in this project, is the point-to-point routing. It is typically used for control *messages* in the *SpiNNaker* system but fits the description of the *POETS* system the closest.

The final part of the router is an emergency routing pipeline stage. When an output connection has been chosen, it attempts to send it along this path by placing it into a *First In, First Out (FIFO)* buffer to be sent as soon as it is ready. If this is not possible due to the buffer being full (thus the connection is congested) or if the connection is broken, the emergency routing is put in effect. In this case, the system routes the *message* to one of it's neighbours to bypass the problem zone**pub:navaridas2009understanding** If this is not possible, the *message* is dropped after a certain time to prevent deadlocks. While the dropping of *messages* is not supported by *POETS*, the other aspects of the emergency routing is very much applicable to my project as it has the can have the same problems that will need to be overcome.

One thing that the routing in *SpiNNaker* does is manage the flow of *messages* and congestion very well. Due to the nature of the default routing, most *messages* travel along a straight line with at most a single point of inflection within it **pub:2_khan_lester_plana_rast_jin_painkras_furber_2008** thereby reducing congestion. In addition, this allows for a high throughput due to low processing cost as well as a lower memory requirements for the routing table.

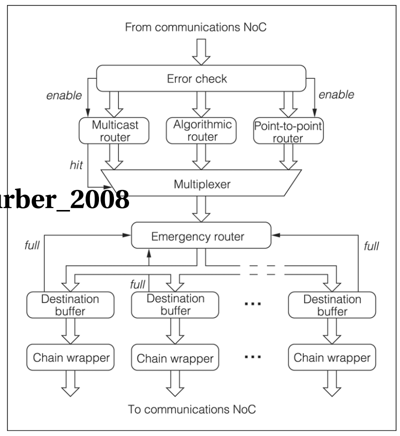

Figure 3.3: Details of a *GALS* Router**art:1_plana_furber_temple_khan_shi_**

As the *SpiNNaker* project is very similar to this one, with the exception of the *FPGA* component, many of the design principles can be used as a starting point for this project. However, due to my project being an adaptation of *POETS* and designed for a broader range of applications than *SpiNNaker* was, some of the design choices cannot be used while others must me modified to fit new criteria if they are to be taken into consideration.

PRISM

Another similar project is PRISM**pub:agarwal1994asynchronous** which is a small-scale proof of concept project for reconfigurable asynchronous computing. This project uses hardware based *CPUs* with an attached *FPGA* for hardware acceleration of commonly executed instruction groups. However, it had issues due to the arbitrary nature of the application code, most noticeably the execution of loops and if/else code constructs on the *FPGA*.

Distributed *FPGA* Research Platform

There is also a similar project at Bucknell University in 2011**pub:2_su_2011** that attempted to provide a purely *FPGA* based distributed system for multiple application types. They were using a configuration with multiple *FPGAs*, each acting as a single *software node* although the board I/O limited the performance gain that they could gain, especially for file heavy applications.

3.2 Networking

According to R. Francis**rpt:francis2013exploring** an issue with for *Network on a Chip (NoC)* on an *FPGA* is the difficulty of getting good latency without paying the price in terms of area. As such, she recommends a design that incorporates fast and low cost circuit switched *network nodes* for local networking with some *soft microprocessors* as a message-switched routers for longer distance packages, reducing the latency of the number of hops that the *message* would usually have to pass. This design also reduces the need for complex circuitry in the local circuit switched routers, while at the same time reduces the need for large look-up tables for distant destinations as shown in fig. 3.4.

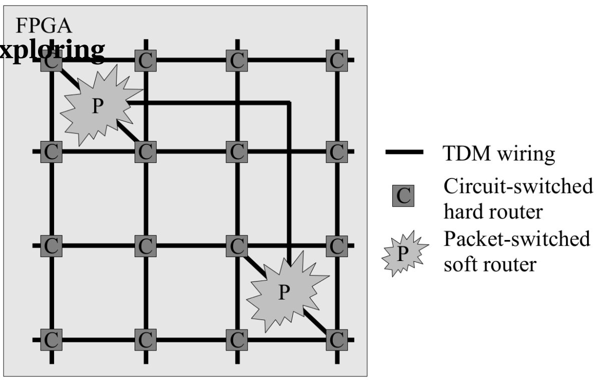

Figure 3.4: Design of a circuit-switched local routing and message-switched distant routing**rpt:francis2013exploring**

3.3 Hardware

Field-Programmable Gate Array

To complete this project, an *FPGA* is required on which the project can be developed and evaluated. The main competitors in this category are the Xilinx Spartan series and Altera Cyclone series of *FPGAs*. Both are the low cost and low power *FPGA* categories from their respective companies and as such the ones that are most readily available.

Out of those two, the Altera Cyclone series of *FPGAs* is the more familiar require less time to develop on it. Within this series, the *FPGAs* that were readily available were the Cyclone III mounted on a Terasic DE0 board**web:terasic_de0** and the Cyclone V mounted on a Terasic

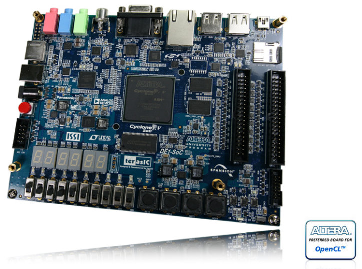

Figure 3.5: Image of a DE1-SoC development board by Terasic**img:terasic_fpga**

DE1-SoC board**web:terasic_de1** While both would have worked as an *FPGA* for this project, as the Cyclone V *FPGA* has a built in ARM Cortex-A9 processor and is supported by a newer version of the Quartus developments software, it is the better choice for *DiRCC* development. While an older version of the development software would have been usable, the newer version, 16.1, has a better compiling and fitting algorithm built in which allows for faster and better optimised compilation compared to the older version, 13.1. Another major issue with the older version of the Quartus software is that it does not run on my machine, meaning that I have to either spend a large amount of time working around assorted compatibility bugs or find a different machine to work on.

In addition, the Altera Stratix V and Stratix X *FPGAs* are the ones targeted by the Microsoft *FPGA* cloud**pub:configurable-cloud-acceleration** which are both supported on only the newest version of Quartus, with only the Stratix V being supported on older versions.

4 Design

4.1 Design Decisions

Based on the project requirements and the research of similar projects and components, a design similar to that of the *SpiNNaker* project (section 3.1) appears the most promising as the architecture and project requirements are very similar. Of course, some of the system will have differences due to *DiRCC* being on an *FPGA* and so will have to be adapted.

Out of the available network architectures, the mesh-based network provides the best design in terms of redundancy, ease of creation and scaling at the expense of increasing the routing complexity and increased latency. However, the alternatives suggested by R. Francis**rpt:francis2013exploring** while useful, appear too complicated to implement properly in the time available. In addition, during this initial development of *DiRCC*, it is unlikely that it will get large enough to warrant the long-range message-switched routing.

For the *network nodes*, the design will be in the form of a single *software node* with attached memory and router. Initially, the *software node* will run on a single *NIOS* processor although these are expensive in area and have a low throughput so the aim, is to replace them with a custom Verilog block that is equivalent to the running *POETS* code without the need for a full *soft microprocessor* to run the code.

4.2 Deliverables

Based on the previous research and the project description, the following deliverables are required to deem the project a success.

- Create *network node* design and implementation
- Create *API* between the *network nodes* and *software nodes*

Reword all deliverables

5 Project Plan

5.1 TIMELINE

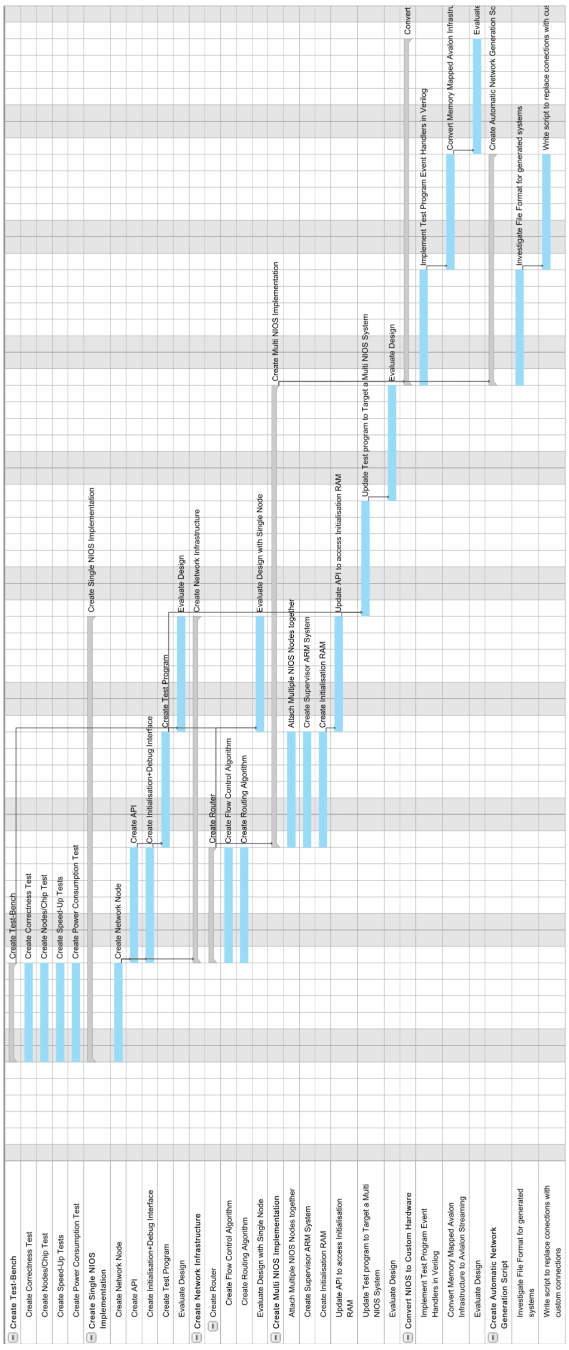

Figure 5.1: Gantt Chart of Milestones and Dependencies

5.2 Required Milestones

These are milestones that must be completed for the project deliverables to be met and the project completed.

5.2.1 Create Test-bench

Description

The first milestone that must be completed is the creation of a test-bench. This will allow the verification of the correctness of the different designs that will be created. This test-bench will run through a specific pattern of tests and provide a short report of the different quantitative criteria that are explained in section 6.1.1. By printing out this report, it will be easier to compare different designs based on the evaluation criteria

By creating a test-bench to provide quantitative evaluation of my designs, the process will be automated, allowing for a faster evaluation cycle as well as allowing me to perform multiple tasks in parallel instead of requiring a manual setup and execution of every test.

This test-bench will be in the form of a small script or program which executes individually created tests in sequence. The results of these tests may need to be formatted depending on the exact output given and then saved to a file to be used later as a benchmark value for future tests.

Risks

A major risk with this milestone is the over-complication the test-bench and thus development of it taking up time that is better spent on developing other milestones. As such, vigilance is required to make sure that the minimum time that is needed will spent on this milestone and the over-complication issue is avoided.

In addition, some of the tests are more important than others, especially the correctness test. As such, those tests that are less important or easier to test by hand than by the test-bench (such as number of *network nodes* on each *FPGA*) are of lower priority and so will have to be removed from the test-bench should it be getting to complicated or taking too long.

5.2.2 Single *NIOS*-based POETS System

Description

The first usable milestone that must be completed is the creation of a single *network node* on the *FPGA* which will run a simplified version of the *CPU* based *POETS* system. Each of the *software nodes* will be run on the same *thread* with all networking between the *software nodes* being handled by the software itself.

To complete this milestone, the *POETS* system must be simplified to remove the requirements for multiple *threads* as these are not supported by the *Hardware Abstraction Layer (HAL)***wiki:hal** and would require a full *Operating System (OS)* to be installed, reducing the performance and increasing the memory required to store the program.

In addition, for this milestone to be completed, a single *network node* will have to be designed and implemented as shown in fig. 5.2, together with the *API* to allow the custom

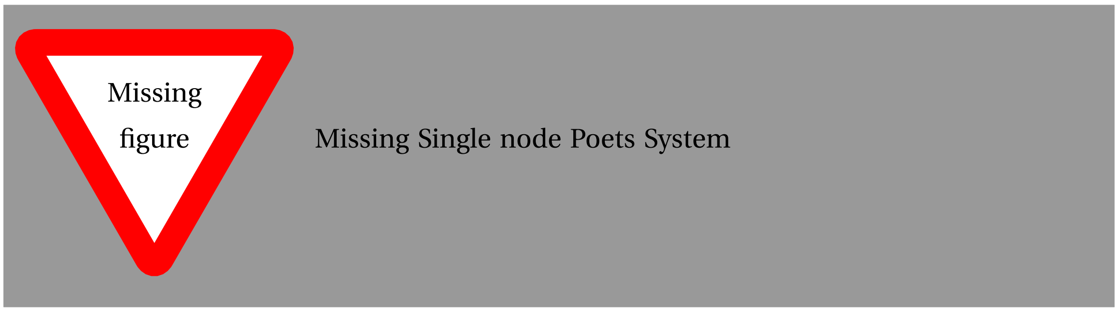

Figure 5.2: Design of a single-node *POETS* system

POETS event handlers to interact with memory and provide the necessary abstraction to allow for easier programming of the *application*.

Risks

This milestone relies a working *CPU* based version of the *POETS* software as well as a Terasic DE1-SoC board on which it can be tested. If the *POETS* software is not available, then this milestone can still be completed, although it may not match the *POETS* specification as closely as it would otherwise have.

If the DE1-SoC development board is missing, then parts of the hardware and software can still be developed through the use of simulation but cannot be combined until the board is available. This will mean that development the next milestones will have to begin to keep on schedule but cannot be integrated until everything is tested.

5.2.3 Network on Chip Infrastructure

Description

To create a multi *network node* based system, an important requirement is the networking infrastructure between all of the nodes within a single *FPGA*. This infrastructure will transport *messages* between the *software nodes* in the form of network *messages* and handle the routing of these *messages* as well as flow control.

This infrastructure has two components that have to be developed to make sure that the result fulfils the criteria given in section 4.2. The first component that must be developed and implemented is the router for standard point-to-point routing between *network nodes*. In this case, it may be useful to implement a form of default routing like the *SpiNNaker* uses as shown in fig. 3.3. At this, point the performance of different routing algorithms will have to be evaluated to determine which one is the best for this system.

The second component for flow control may be required, depending on the measured throughput of the previously mentioned routing algorithm. If the throughput is too low, then I will have to implement a module into the router that attempts to ease the congestion along a particular *channel*. In essence, this component will have the same purpose as the emergency router in the *GALS* system **art:1_plana_furber_temple_khan_shi_wu_yang_2007** However, it will have to designed slightly differently to fulfil the requirement of no *messages* being dropped. On the other hand, I do not need to account too much about hardware failures as this network will be built onto a single *FPGA*.

Risks

Within this milestone, the development has all been done before and as such is known to be possible. The major risk in doing it though is the risk of missing the internal deadline for this milestone. If it appears to become the case, the first response would be to remove the requirement for flow control from within the network. While it is useful for a fast implementation of the "Multi NIOS-based POETS System" (section 5.2.4) milestone, the system will work without it due to the guarantee of no *messages* being list.

Unfortunate

5.2.4 Multi NIOS-based POETS System

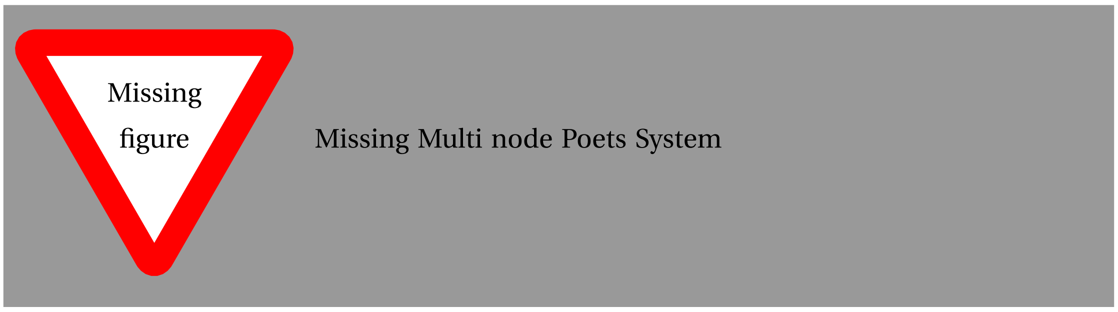

Figure 5.3: Design of a multi-node *POETS* system

Description

Once I have finished the previous milestones, I am ready to connect multiple *NIOS* based *network nodes* into a single *POETS* system within a single *FPGA*. This milestone will provide the first working version of a *POETS* system on an *FPGA* with multiple *software nodes* running at the same time and passing *messages* between each other.

While much of the infrastructure to create this milestone will have been done in the previous milestones, at this stage, the software will have to similarly be updated so that it can run on multiple *software nodes* at the same time in an asynchronous fashion.

In addition, due to there being multiple *software nodes* running at the same time, it will be difficult to oversee all of them and initialise them by hand. As such, a comprehensive initialisation, debugging and data extraction infrastructure will need to be created at this point to allow for the system to automatically be configured to run the application with as little intervention on my part as possible. This system will also allow me to oversee and debug the application much more easily.

To do this, each *software node* will require an interface to expose the internal memory as well as routing configuration to some sort of overseer (most likely the ARM based *CPU* on the *FPGA*, although a *soft microprocessor* would also work) which allows for the extraction of debugging information and also initialisation of the *software node*'s memory.

Risks

5.3 Optional Milestones

These are milestones that should be completed by the end of the project but that can be removed if the project falls behind schedule/they are too difficult to complete.

5.3.1 Replace NIOS with Custom Hardware

Description
Once I have created the basic network for *POETS* to run on using *NIOS* processors, the next task that I will have to complete is to replace the *NIOS* processors with custom Verilog hardware. This will ideally reduce the complexity of the *network nodes* as the *NIOS* itself uses up around 16000 logic element. In addition, this change will allow me to transition from memory-mapped model to a streaming model, thereby potentially improving the throughput of each node.

To achieve this, I will have to write custom Verilog modules that perform the same actions as the curretn *POETS* receive and send event handler have as well as the memory storage for that *software node*'s state. HOwever, the routing system will not have to be modified as the interface between the processing *software node* and the router already had the streaming interface.

Risks

5.3.2 Automatic Node Creation Script

Description
Another optional milestone that I wish to complete is a script to automatically create and connect the system based on a few input parameters. At small sizes, the system can be built up and connected by hand but after a while, this method becomes unwieldy, especially for the network connections.

As Quartus saves the project in a variant of XML data, it should not be a difficult task to create a parser for this file and insert new *network nodes* to it with the correct network connections. By doing this, I will be able to significantly cut dfoen on the time that it will take to generate different sized networks and be able to automate a significant portion of the repetitive work that I will end up doing.

As this will cut down on the work that I will be doing, it may be beneficial to complete this milestone as soon as I have the first working version of the multi-node system in place, but it is not required to complete the project and so will remain an optional milestone.

Risks

5.4 Extension Milestones

These milestones are not expected to be completed unless everything else goes better than expected and the project is already fulfilling all objectives.

5.4.1 High Level Synthesis for Custom Hardware

An automation task that would be beneficial for this project is to automate the creation of the event handlers from the C code that it is written in to the verilog that is required for the replacement of *NIOS* with custom hardware milestone (section 5.3.1). To reduce the amount of work that I will have to do when converting more complicated programs into verilog.

Doing this will save me some time although it may become very complicated, especially with all of the data structures that *POETS* includes as well as the interface to the actual hardware that will have to be considered.

Description

Risks

6 Evaluation Plan

6.1 Evaluation Criteria

6.1.1 Quantitative

These criteria are ones that I will use to evaluate the

finish

To compare these results, I will require a benchmark result which I can use a s baseline to determine how the performance of the tested design has changed. For this I will use a *CPU* based version of the *POETS* system as this is for all intents and purposes correct and is also the version of *POETS* that I started the project with. However, this version can have wildly different performance characteristics depending on the hardware that it is running on.

To account for this, all of the benchmarks should come from a single hardware configuration to allow for correct comparisons during the evaluation stage. While I could use my personal computer for this test, *POETS* is designed to take advantage of a larger numbers of parallel *CPUs*. As such I will attempt to create a benchmark of the *POETS CPU* application on the hardware that it was originally designed upon at Cambridge University for the BIMPA project**pub:naylor2014rapid** and fall back to my personal computer should this not prove a feasible solution. However, at that point, the results that I will gather for quantitative evaluation will no longer accurately reflect the parallelism that the *POETS* system provides.

Maybe explain more

In either case, it will be beneficial to store the baseline results to prevent changes in result data between individual tests and also considerable speed up the time taken to test any evaluation criterion due to the time taken to recompute the baseline having been removed.

add flow chart of testing procedure here

Correctness

The most important criteria for which I will have to evaluate each solution to my project is correctness. For this, I will have to compare my implemented solution against a reference solution which will be the *CPU* version of *POETS*. If the results that are received from both versions are identical for a given test program, then I will know that the solution is correct and can then go on to evaluate the other criteria.

This criteria will ideally be done through a *bit-exact* matching of the reference output and the output of the tested system at each step. As the *POETS* architecture only releases the result once all *software nodes* have reached the same time step/calculation step, these results should be identical regardless of teh system that they are running on.

During testing with a simulation program running on a *CPU* and later on a *GPU*, I noticed that the results did not always match up in a *bit-exact* pattern. These differences could not easily be predicted beforehand although they tended to be low in number. They also tended to appear more frequently on computational heavy tasks and those where floating point numbers were being used.

There are two main methods to avoid this sort of error during the testing of this project that do not restrict the types of tests that can be run. The first is to manually go over the results every time these errors appear and determine if they are significant or not. This method will obviously be slow to do and also complicated for large scale tests with many results.

The second method that I can apply is to use a simple statistical model to automatically decide whether the errors that were found are true errors or simply due to differences in the hardware that is running the test. For this I could use measures such as the *bit-exact* difference between the two values as well as the total number of errors that have occurred. If both values are less than some to be determined threshold, then the errors can be safely ignored.

Output type?

Nodes On Chip

The main advantage that an *FPGA* has over traditional *CPUs* is the ability to have many components in parallel. As such, the number of *network nodes* that can be fit on a single *FPGA* is an important evaluation criterion. This will allow me to compare different implementations as well as versions of the same implementation to evaluate what provides the best usage of the total area of the *FPGA*.

Feels like missing explanation

For this criterion, the measure will be number of discrete *network nodes* that can be placed onto the Cyclone V *FPGA* that I am working with, together will all of the associated infrastructure that they require. For this criterion, there is the danger of the tooling optimising the placement of the different designs by different amounts if left to their own devices with the highest possible timing requirements. As such, for this criterion, the timing requirements will be considerably relaxed to allow the tooling to place the design

In addition, for this criterion, there is the danger that the testing of it can take considerable time as the design will have to be rebuilt with an ever increasing number of *network nodes* contained within it. As such, I will have to calculate the maximum number of theoretical *network nodes* contained within the system by dividing the total number of *logic elements* by the number of *logic elements* used within a single *network node* as shown in eq. (6.1).

$$N_{max} = \frac{LE_{total}}{LE_{node}} \tag{6.1}$$

talk about removing some as this is the theoretical max

As this criterion is not applicable to any *CPU* implementation of the *POETS* system, this criterion will be used in comparison between different *FPGA* implementations only and so will not have a baseline value which I will compare against.

Speed-up compared to CPU implementation

One important measure of the performance of my *FPGA* based system is to compare the time taken to execute a given set of problems using the standard *CPU* implementation to the implementation on the *FPGA*. By running this test I can evaluate how many times faster or slower the *FPGA* implementation is compared to *CPU* implementation using the formula shown in eq. (6.2).

$$S = \frac{T_{FPGA}}{T_{CPU}} \tag{6.2}$$

Speed-up compared to other FPGA implementations

To evaluate the different implementations of the *FPGA* system, I will have to compare them against each other. From the test to evaluate the speed-up against the *CPU* implementation, I will already have all of the raw data available, so this criteria will simply require calculating the speed-up using the formula shown in eq. (6.3)

$$S = \frac{T_{FPGA_{new}}}{T_{FPGA_{old}}} \tag{6.3}$$

Power-usage compared to other Field-Programmable Gate Array designs

In many of the applications, where a distributed systems are used, power usage is a big concern**pub:4_xizhou_feng_rong_ge_cameron** as it directly relates to heat produces and electricity costs, which are magnified by the number of components running at the smae time in distributed systems.

As such, one evaluation criteria is the power consumed by the designs during normal operation. For this, I will have to create a processing intensive and long-running application, which will be run on the design and the power usage profiled.

While it would be nice to compare the power usage between the *FPGA* and a *CPU* implementation, due to the difference in hardware, the results will not be in any way meaningful as different hardware configurations will yield different results. As such, I will focus on differences in power usage of the *FPGA* designs as they will all be running on the same hardware.

To calculate the power consumption, I will require a baseline of power consumed by the *FPGA* when it is not running. This static power, will always exist in some form or another as all unused transistors will be consuming power all of the time**5_gardiner_2008** However, as all of these calculations will be running on the same hardware, the power consumption will be roughly similar for all tests or at the very least a function of the transistors used. As such, the measured result can be used to evaluate the different algorithms.

As this method can be quite time consuming and complicated to do on an actual *FPGA*, I will use the Altera PowerPlay**man:powerplay** for faster estimation of power usage of the different designs and then use actual measurements when the full designs are completed. By using the Altera tool, I will also be able to compare the estimated power usage for different sizes of systems as well.

Maximum Clock Frequency

For an *FPGA*, the maximum clock frequency that the design can run at is a good measure of how well the design works on the *FPGA*. With higher clock frequencies meaning that the system is better pipelined. However, much of the performance in this regard comes from how much effort the fitter puts in to try and meet the timing constraints. As such, I will have to take care to have the amount of effort that the tools do for each test to be the same.

One way of doing this is to keep recompiling the system with increasing frequency constraints until it cannot be done. However, this has the downside of taking a very long time to do, especially near the limit of clock frequencies.

As such, a second way would be to compare the estimated maximum clock frequencies that *Quartus* reports during compilation. While this measurement is not as exact as the previous method it will suffice for the detection of large changes in this value but will have difficulties with very similar clock frequencies.

6.1.2 QUALITATIVE

Some of the evaluation criteria cannot be quantified into any meaningful value. While these are important, they are not as important as the quantitative criteria that the designs exhibit. To evaluate these criteria, I will have to base the results on personal experience and only in a comparative sense, e.g. 'A is easier to use than B' or 'B appears to scale to larger systems compared to A'. The main reason I am basing these on personal experience while developing the project is due to there not being enough time available to train others in the use of the system and then judge their responses.

Ease of Scaling

The first of qualitative criterion that I will be judging is how the system appears to scale to a larger number of *software nodes*. This does not mean how many nodes it can theoretically or practically support as this will be limited by the address space and network congestion. Instead, I will be judging how much effort it takes to create a network of a certain size and then add extra *software nodes*.

Actions like placing all connections by hand and individually changing addresses will mean a worse ease of scaling than an automated tool that automatically creates a network for a given input size.

Flexibility of Application

The second qualitative criterion is how flexible the system is in terms of custom applications. If the developer who is developing on the system requires a lot of effort to produce an application that the system was not explicitly designed for, then the flexibility of it will be rather poor. On the other hand, if the system can easily be used with a wide variety of application types, then the flexibility will be higher.

7 Conclusion