DEGREE PROJECT IN ELECTRONICS AND COMPUTER ENGINEERING
180 CREDITS, FIRST CYCLE
STOCKHOLM, SWEDEN 2015

An Evaluation of the Predictable System-on-a-Chip Automated System Design Flow

MIKAEL KARLSSON

KTH ROYAL INSTITUTE OF TECHNOLOGY

SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

KTH Information and
Communication Technology

KTH Royal Institute of Technology

ICT School

Bachelor project in Electronics and Computer Engineering

An Evaluation of the Predictable System-on-a-Chip Automated System Design Flow

Author: Mikael Karlsson
Supervisors: Kathrin Rosvall
Associate Professor Ingo Sander

Examiner: Associate Professor Ingo Sander, KTH, Sweden

Abstract

In spite of hard real-time embedded systems often being seemingly simple, modern embedded system designs often incorporate features such as multiple processors and complex inter-processor communication. In situations where safety is critical, such as in for instance many automotive applications, great demand is put on developers to prove correctness. The ForSyDe research project aims to remedy this problem by providing a design philosophy based on the theory of models of computation which aims to formally ensure predictability and correctness by design. A system designed with the ForSyDe design methodology consists of a well defined system model which can be refined by design transformations until it is mappable onto an application specific predictable hardware template. This thesis evaluates one such hardware template called the predictable System-on-a-Programmable-Chip. This hardware template was developed during the work on a masters thesis by Marcus Mikulcak [7] in 2013.

The evaluation was done by creating many simple dual processor systems using the automated design flow provided by PSOPC. Using these systems, the time to communicate data between the processors was measured and compared against the claims made in [7].

The results presented in this thesis suggests that the current implementation of the PSOPC platform is not yet mature enough for production use. Data collected from many different configurations show that many of the generated systems exhibit unacceptable anomalies. Some systems would not start at all, and some systems could not communicate the data properly.

Although this thesis does not propose solutions to the problems found herein, it serves to show that further work on the PSOPC platform is necessary before it can be incorporated into the larger context of the ForSyDe platform. However, it is the author's genuine hope that the reader will gain appreciation for PSOPC as an idea, and that this work can instil interest into further working on perfecting it, so that it can serve as a part of ForSyDe in the future.

Referat

Även om hårda realtidssystem ofta verkar enkla så finner man i moderna inbyggda system numera ofta avancerade koncept såsom multipla processorer med komplicerad processor-till-processor-kommunikation. I situationer där säkerhet är ett kritiskt krav, som t.ex. i många applikationer inom bilindustrin, så föreligger enorma krav på de som utvecklar dessa system att kunna bevisa att systemen fungerar i enlighet med specifikationerna. Forskningsprojektet ForSyDe försöker lösa dessa problem genom att tillhandahålla en designfilosofi baserad på teorin om så kallade models of computation som via formella bevis kan garantera förutsägbarhet och korrekthet. Ett system designat med ForSyDes designmetodologi består av en väldefinierad modell av systemet som transformeras, tills dess den kan mappas mot en applikationsspecifik förutsägbar hårdvarumall. Detta examensarbete ämnar att utvärdera en sådan hårdvarumall som kallas predictable System-on-a-Programmable-Chip, eller PSOPC. Denna hårdvarumall utvecklades under arbetet med en masteruppsats av Markus Mikulcak [7] under året 2013.

Utvärderingen bestod av skapandet av ett enkla tvåprocessorsystem med hjälp av PSOPCs automatiska designflöde. På dessa mättes sedan tiden för att kommunicera data mellan processorerna. Dessa kommunikationstider jämfördes sedan med de påståenden som görs i [7].

Resultaten som presenteras i detta examensarbete föreslår att nuvarande implementation av PSOPC-plattformen inte ännu uppnått tillräcklig mognad för att kunna användas i verkliga tillämpningar. De data som insamlats från många olika systemkonfigurationer visar att många av de genererade systemen uppvisar oacceptabla avvikelser. Några system startade inte ens och några klarade inte av att kommunicera data på ett korrekt sätt.

Även om detta arbete inte föreslår några lösningar på de problem som presenteras häri så visar det på behovet av mer arbete med PSOPC-plattformen innan den kan bli en del av hela ForSyDe. Men, det är författarens genuina förhoppning att läsaren förstår de positiva aspekterna av PSOPC som idé, och att detta arbetet kan ingjuta intresse för att arbeta vidare med plattformen, så att den i framtiden kan bli en integral del i ForSyDe.

Acknowledgment

I am deeply thankful to my fiancé and my son. Not everyone is blessed with a family capable of such patience as mine have had throughout my late-in-life studies.

Thank you!

Stockholm, December 2015

Mikael Karlsson

Contents

List of Figures

List of Tables

Abbreviations

BCCC ..... Best Case Communication Cycles
BSP ..... Board Support Package
CPU ..... Central Processing Unit
DSE ..... Design Space Exploration
DSP ..... Digital Signal Processing
FIFO ..... First In First Out buffer
FPGA ..... Field Programmable Gate Arrays
I/O ..... Input/Output
JTAG ..... Joint Test Action Group
MoC ..... Model of Computation
PSOPC ..... Predictable SOPC
SDF ..... Synchronous Data Flow
SDRAM ..... Synchronous Dynamic Random Access Memory
SoC ..... System on Chip
SOPC ..... System On a Programmable Chip
TDM ..... Time Division Multiplexing
UART ..... Universal Asynchronous Receiver Transmitter
WCCC ..... Worst Case Communication Cycles
WCET ..... Worst Case Execution Time
XML ..... eXtensible Markup Language

Nomenclature

t token size in bytes

b buffer length in tokens

s TDM slot length in cycles

Chapter 1
1 Introduction

1.1 The Computer as an Infallible Machine

When Charles Babbage in 1823 received government funding to build his difference engine, it was his intention to remedy the fact that human computers made errors [2]. His desire for a machine that could never produce an error stemmed from the fact that the scientific society of that time made extensive use of hand crafted tables. These long lists of numbers would often contain errors, and a scientist or an engineer would sometimes propagate these errors into their calculations or onto their designs. In a situation where no errors could be tolerated they would have had to recalculate these values themselves rather than blindly trust the printed tables.
So Babbage planted the first seed towards the infallible machine we have now come to trust ever so blindly.

This machine would eventually evolve into the mature entity we now know as the computer. As computer engineers we are taught that a computer has an error rate of almost infinitesimally small proportions, i.e. the computer is in practice every bit as infallible as Babbage would have hoped for. Still we face the fact that computer programs make errors. Clearly having a machine do all calculations were not sufficient. Humans are still in charge of telling the computer what to calculate.
The programs we use to instruct the computer what to do has grown into huge convoluted beasts that we often don't comprehend. We are in the same situation as Babbage found himself in. We cannot any longer use the computer and feel confident enough to claim to know that the computer will do what we think it will do in all situations.

1.2 Humans Make the Errors

In the field of real-time systems, and in particular the so called hard real-time systems, it is essential that we can predict not only the outcome of a certain computation, but also at which point in time the result of the computation is available. We expect

these systems to respect certain deadlines. A missed deadline can in some contexts have fatal consequences. Once again we as humans are the ones who introduce errors. These errors are obscured deep within layers of code, and manifests themselves, many times, on occasions long after the programmer finished his work.

We realize that we can formulate rules that, if obeyed, allow us to claim to know what the program will and won't do, with the same level of confidence as we trust the rules of algebra.

The numerical tables of Babbage's time were calculated according to well understood and well formulated laws, the laws of mathematics, and just as Babbage realized that humans are apt to making errors and should be replaced by machines, we too should now replace ourselves as programmers with machines for the very same reason.

This movement towards higher abstraction is a movement that has existed throughout the history of modern computers, but often with an emphasis on giving programmers more expressiveness rather than to ensure correctness.

We have devised extensive theory with well formulated rules that guarantee programmatic correctness. However, we have proven beyond doubt that we often fail to adhere to these rules when we try to apply them manually, whether it is unintentional or because we are lazy. It then seems only reasonable to let computers infallibly perform the application of these rules. We should merely present the computer with the desired behavior, and let the computer provide the correct implementation.

1.3 It is Hard to Change

The engineering community is a slow moving colossus. In Babbage's days, his machine threatened to put the human computers out of work. In modern day, a shift towards new and better practices depends on industries willingness to invest in new knowledge. The main problem with shifting towards a more formal paradigm in programming is that one perceivably has to give up some of the expressiveness of our current de facto programming languages. Some programming constructs are just inherently "unsafe", at least in safety critical applications. We have to be more stringent in our work and we need to re-learn certain aspects of what we think we already know. To re-educate entire workforces requires cost and time intensive investments. Making huge investments is something that the industry is generally reluctant to do, especially if the benefit of the investment is blurry at best. Corporations have made long lasting commitments into what tools and technologies to use, and it is not easy for them to just turn and start moving in a new direction.

On the other hand, making the transition easy by making available tools and workflows to go with new research findings is not something academia is particularly

proficient in. Researchers generally formulate their theories, only to go on to rant about what a glorious world we would have, would we only implement hers or his ideas.

1.4 The Research Community Provides the Solutions

In year 1996 research commenced on what would eventually be called *Formal System Design* [9; 10], or ForSyDe for short. ForSyDe is a methodology that lets the engineer formulate the behavior of the system on a high abstraction level, in such a way that it's function is formally comprehensible, and the behavior is easier to prove through analysis. The current state of the art to "prove" correctness of a program is by simulating it. In very few practical situations, however, is it possible to define a simulation in such a way that one can say that the simulation exhaustively proves that the program will do what it is supposed to do, and further more, that it will *never ever* do anything that it is not intended to do. Programs developed in ForSyDe can be shown to behave according to the defined behavior, and hence the need to simulate is diminished.

The ForSyDe research project has grown and now encompasses not only the methods to provide the formalism, but also an entire proposed tool set and workflow all the way from an abstract model to a complete hardware/software system. This thesis aims to evaluate portions of the proposed workflow at its current state of development. It is in large parts a continuation of previous work done by Marcus Mikulcak which is presented in his master's thesis from 2013 [7]

1.5 Summary and Contribution

Chapter 2 presents the background to this thesis. It will briefly address the topic of developing hard real-time embedded systems, and the challenges related to this. It also talks about the ForSyDe research project conducted at KTH. Finally it introduces the PSOPC hardware template and automated system generation workflow on which this thesis is based. Chapter 3 gives a hands on tutorial on how to create systems with PSOPC. Chapter 4 describes the contributions of this thesis and presents the collected data and subsequent analysis. Chapter 5 presents the authors conclusions as well as suggestions on future work.

Chapter 2
2 Background

This chapter aims to provide an understanding of what work has previously been done leading up to this thesis. The first section presents arguments for formal design in general, followed by a presentation in broad terms of the ForSyDe research project. The third section presents the design space exploration phase of the proposed workflow, while the fourth and fifth presents the PSOPC platform and the purpose of this paper.

2.1 Embedded Systems

Embedded systems are all around us. Today, almost all consumer electronics contain embedded microcontrollers in one form or another. Whether an embedded system works or not is not always critical. Sometimes it becomes a mere nuisance when the television does not respond to your frantic pressing on the remote. But sometimes it is of utmost importance that the embedded system never fails. The task of designing as complex systems as we nowadays encounter, and at the same time leaving guarantees on the operation, is not trivial. This section tries to explain why it is not so easy using de facto programming languages and current best practises on modern computer systems.

2.1.1 Hard Real-Time Systems

In critical applications a system needs to be designed in such a way that the operation can always be guaranteed. This can be exemplified with the example of an airbag controller in a car. We depend on that the operation of such a controller is always correct and never executes in an untimely manner. It needs to react to a certain stimuli at precisely the correct time, otherwise the results are catastrophic. The airbag controller is an example of what is generally called a *hard real-time system*.

The single most important requirement of processes in a hard real-time system is that a process always finishes before its *deadline*, i.e. a point in time after which the result is no longer valid. Missing this deadline usually means failure, in some cases failure of catastrophic measures as in the above mentioned example.

To know whether or not a process will meet its deadline, one has to measure or calculate its *worst case execution time*, or WCET. We can try to estimate it by simulation, but in case it is possible to formally analyze it, analysis is preferred over simulation.

2.1.2 General Purpose Computers

Modern general purpose *central processing units* or CPUs, as well as general purpose operating systems, are not well suited for these kinds of applications for a number of reasons. The systems we come in daily contact with, the operating system and CPU used in our laptops or in our phones are designed to be as fast as possible in the general use case. This means that throughout the history of computer research, the focus has been on inventing tricks that speed up the most common use case scenarios, at the expense of predictability. Modern CPUs have many features that make it impossible to determine how much time a piece of code will take to execute. Features such as *branch prediction* that lets the CPU guess the outcome of an *if..else* test are commonplace in current generation CPUs. Often the guess is correct and the CPU will have spent less time performing the test and the actions following the test, but sometimes the guess is wrong, imposing a penalty for the mistaken assumption.

Modern CPUs also typically contain *cache memories* which are fast memories, usually placed on the die of the CPU itself. Caches are meant to hold data that is highly likely to be needed in the future, so that one can avoid to fetch data from slower storage mediums, such as SDRAM and hard disk drives. In the general case the needed data might be present in the cache, which means that it will be available fast. In some cases it might not be available in the cache, in which case the data acquisition needs to happen from a medium often magnitudes slower. If the data will be cached or not is not always predictable and one has to assume the worst case scenario which often lead to unnecessary overestimation of the access times.

Similarly, modern operating systems are generally optimized for perceived responsiveness and speed in the most common scenarios, at the expense of lack of speed in more uncommon scenarios. They employ various buffering schemes to mimic the function of cache memories, and in case of multithreaded process execution, they rarely employ fully predictable scheduling schemes.

As there are no means to deterministically analyse these kinds of behaviours, we cannot use these kinds of CPUs and operating systems to control our airbag controller

as we can not be sure it will always react with the desired behaviour in any situation. It depends on factors we cannot control or predict.

As will be described in Section 2.4, part of the solution to this is to use hardware over which we have control and which exhibits what is called composability and predictability.

2.1.3 General Purpose Programming Languages

Throughout the history of modern computing we have seen a great number of programming languages spring to life. Some languages are specialized, intended for specific use cases. Others are more general, suitable for a wider array of applications. Almost all languages have had some particular type of applications for which it is a good fit. Some languages execute faster, while others might have a more expressive syntax. During the evolution of programming, some distinct classes of programming languages have emerged. The popularity of a language does not automatically correlate to suitability for all types of applications though, as this section tries to explain.

2.1.3.1 Imperative Languages

The by far most common class of programming languages right now are the so called *imperative languages*. These include languages such as C, FORTRAN, Java etc. The common denominator of these languages is that they all define how a program should perform computations. The programmer lists the steps in an algorithm almost as a cooking recipe, and the computer executes the described steps. This is very close to how the computer actually operates, as it takes one instruction at a time and does whatever that particular instruction is meant to do. During the execution of a program, state is being maintained and altered, and the program might be programmed to do different things depending on the current state. This leads to apparent problems with the ability to formally analyze a program written in this fashion, since, to be able to have claimed to fully understand what a program does, not only does one have to consider all possible inputs to a program, but also all possible states in conjunction with all possible inputs. The problem of completely analyzing a program is not always possible.

Since algorithms described imperatively are often inherently sequential as in “first do this, then this and then this etc.” it is often hard or even impossible to parallelize the execution. The execution of the program depends on one set of states, and if concurrent threads of execution alters that one set of states simultaneously one often find that this leads to problems.

2.1.3.2 Declarative Languages

In an effort to alleviate the situation many other languages and classes of languages have been invented. The main class is often referred to as *declarative languages*. The difference between an imperative language as opposed to a declarative one, where in the former one specifies "how" something is done, in the latter one instead specify "what" should be done.

A special class of declarative languages are the *functional* languages, such as Scheme, Clojure and Haskell, to name but a few. The rationale behind these being that, as with a mathematical function, a functions result is only depending on its input. It does not depend on any hidden states. A certain combination of inputs will always render the same output at every instant in time independent of what has happened before. From an analyzability standpoint, this is far better than with the imperative languages. Also, since the programmer specifies what is to be done, rather than how, it is up to the compiler to implement the program. As such, the compiler is free to make whatever alterations it see fit, as long as it preserves the function. Many compilers are for instance quite able to parallelize programs. Since in purely functional programs, execution does not rely on states, one can often split the execution of many sequential statements into parallel threads and perform many computations concurrently.

2.2 ForSyDe

ForSyDe tries to solve three problems. First it forces the system designer to think structured. The idea is to design programs in such a way that they are already correct to begin with. This is achieved by forcing the designer to abstractly model the program first. In traditional imperative programming, programmers tend to go from a written (or often only a vague idea not even put on paper) straight into coding algorithms on a low abstraction level. This often have the unfortunate effect that the developer looses focus on the big picture. In larger development teams it is even more important to properly model the system on a high abstraction level, so that everyone can agree on what the system is supposed to do. Only after a complete model has been agreed upon should any low level coding commence. But even in large corporations with experienced teams, developers rush into the creative coding and forget about the model. When it becomes time to connect all the systems parts, they quite often don't fit together because the proper interfaces between modules were not modeled on beforehand.

Would a model have been established in the first place, it would have been easy to compare the implementation to the model and continuously assess the integration of all the parts. Since ForSyDe forces the developer to first create a model, the implementation will much more likely behave as expected. Furthermore, a model

in ForSyDe is directly executable, which means that one can conduct simulations already in the modelling stage of development. This is a huge benefit compared to other paradigms, where it is often not until the later stages in production where any actual system behaviour can be tested and assessed.

Secondly, ForSyDe maps the system model onto one of several software/hardware implementation according to a set of constraints defined by the system designer. These constraints can include limits on memory usage, hardware cost, power demands etc. This lets the developer, not only automatically get an implementation, if such exists, that complies with the requirements, but also try different tradeoffs by altering the constraint set.

Finally ForSyDe generates the complete hardware/software system according to the output of the design space exploration. The purpose of ForSyDe in short is to let the designer create systems whose function is formally proven to be correct as well as optimal with respects to constraints.

2.2.1 Synchronous Data Flow

ForSyDe programs are expressed as a graph of nodes, each of which are one of many supported models of computation, or MoCs . This thesis will focus on *Synchronous Data Flow*, or SDF [6] since this is the only MoC currently supported by PSOPC, the platform this thesis aims to evaluate. Without going to deep into what SDF means, it is a model of computation that has the following important properties. The actual time passed when executing an SDF, as a consequence of computing and transferring of data, is not taken into account. The only notion of time is in what order computations occur. An SDF is usually modelled as a directed graph as shown in figure 2.1. In an SDF, nodes are called actors and are the computational parts of the program. Arcs between actors are communication channels between different computational blocks, often modelled as FIFO queues. An actor is said to "fire", meaning that a computation may occurs only as soon as there is sufficient input available. The data that is passed between actors are called *tokens*, and can be any kind of data that can be atomically defined [1]. The number of input tokens is fixed and known at compile time. As soon as there are as many tokens as an actor needs to be able to do its computation, it becomes eligible for scheduling. At any point in time after this the actor may fire. When the actor fires it "consumes" all inbound tokens, and will eventually produce a fixed and predetermined number of output tokens. In SDF an actor always consumes a fixed number of input tokens and produces a fixed number of output tokens per each "firing".

[1] A token can only come from one actor, traveling to one actor. There is no restriction on what the data actually is. It could be a simple integer, or a satellite image, or even a list of images. But it cannot be a compilation of data from different source actors, e.g. an x coordinate from one actor and a y coordinate from another.

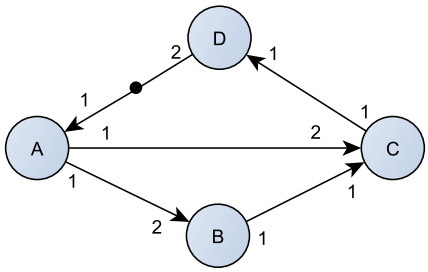

Figure 2.1: An example of an SDF graph. Actor C can only fire once A has fired twice and B once. The dot on the arc between actor D and A denotes a delay. This delay is needed to get the system started, since some actor needs to be the one to fire first and consequently needs some initial input. Usually this data is some null value. A valid schedule for this system would be A, A, B, C, D, and then this schedule repeats. The delay in this case needs to hold two initial tokens. A real system would have inputs and outputs as well, but are omitted here for clarity.

The benefit of modelling using SDF is that it is easy to formally prove the function of the program. Once the SDF graph is defined, i.e. the number of input and output tokens for each node has been determined, one can analytically determine a static schedule for the execution order. This means that it can be mathematically proven that the program is schedulable, i.e. that it will not halt unexpectedly, a so called deadlock, and that the size need for the communication buffers will be bounded and not grow past a maximum size. A static schedule is often more straight forward to implement, compared to dynamic schedules.

These graphs can then be transformed into other graphs with the function preserved [11], such as breaking one node into multiple parallel nodes so that execution of parts of the program can be done concurrently.

2.2.2 Haskell

The first implementation of ForSyDe uses Haskell as its programming language which, being a purely functional language, detaches the formulation of the program from the implementation. Once the program is expressed, since its constituent parts are side effect free, the program can be transformed without altering the function of the program according to a set of transformation rules. These transformation rules can be shown to produce formally equivalent results [9].

2.2.3 SystemC

In an effort to gain interest from the industry a SystemC implementation of the ForSyDe system was developed. Being derived from the C programming language,

it is an easier entry into ForSyDe for programmers with a background in C and its relatives. However, since C is not a purely functional language, some language constructs has to be avoided in order to comply with the stringent rules of ForSyDe.

This thesis has not been developed with neither Haskell nor SystemC, so the reader will only need basic knowledge of C to understand the code presented later.

2.3 Design Space Exploration

Design space exploration or DSE, is the act of searching through the set of possible implementations for a best fit. ForSyDe contains a tool that tries to map the system behavior defined as an SDF graph onto one of many predictable hardware platforms [8]. The mapping tries to find an optimal implementation and schedule based on the graph on one hand as well as a set of constraints on the other. The designer may constrain the memory usage, the power requirements, the processing throughput etc. [8]. The output of the mapping step is a hardware implementation, a schedule and performance data.

The inherent problem of DSE is that even trivially sized systems has enough parameters to make the search space vast, and for large systems it may very well be so that an exhaustive search is not feasible within practical limits. The ForSyDe DSE tool's solution to this problem is a tradeoff between solution optimality and search time [8]. The set of mapping tool is modular to accommodate for the addition of more types of constraints in the future.

2.4 The Predictable System-on-a-Programmable-Chip platform

Many hardware platforms for embedded systems nowadays sport many processing cores. Especially in the field of *Field Programmable Gate Arrays* or FPGAs, one can devise systems with an arbitrary number of processors, only limited by the number of logic blocks in the particular FPGA. The PSOPC platform fits into the end of the ForSyDe tool chain. It is responsible for taking the output of the DSE stage as input, and produce the actual hardware synthesis for an FPGA. The PSOPC system is based on Altera's line of FPGAs, and in particular the SOPC-builder software for aiding in the construction of *System on Chip* or SoC.

Altera provides a *soft* processor core called NiosII. NiosII comes in three variants where NiosII/e (e for economy) is most suitable for predictable systems for rea-

sons such as lack of advanced instruction pipeline, branch prediction and cache memory [5].

SOPC is primarily a graphical environment which aids the developer in generating whole systems with all the needed components and peripherals, such as processor, memory controllers, I/O, as well as the connection between all components. The interconnect structure provided by Altera, called *Avalon Switch Fabric*, is however not optimal for our purpose. It is not a bus in the traditional sense, where only one master device can communicate with one slave device at any time, but shares some features of more advanced network topologies. Due to what is called the *slave-side arbiter* and a flexible routing scheme, multiple masters can simultaneously communicate with multiple slaves provided that not more than one master tries to communicate with the same slave. As Figure 2.2 and 2.3 on page 15 shows, when two or more masters request to communicate with the same slave device, the arbiter selects which master is granted access, and blocks the other masters. To let all masters have time to communicate, the arbiter employs a *fairness-based round-robin* scheme [4; 7], where each master is guaranteed a selectable number of transfers. However, if one master is not using all allotted number of transfers it forfeits them to the other masters, which has their number of transfers immediately replenished. This means that if one master α is allotted 4 transfers and another master β 4 transfers, it is not certain that master α actually gets half of the total amount of communication time. If master α only accesses the slave for one transfer and then goes away to do other business, but then right away decides it wants to make another transfer, it has to wait until all the other masters has had their turn, effectively leaving master α with only 1/5th of the total time. If two masters are connected to the same slave device, their timing behavior is dependent on each other and the system is said to be not composable.

2.4.1 Time Division Multiplexing arbiter

As explained in the previous section, components connected through the standard interconnect are not composable. The timing changes depending on other components and their actions. Composability is a criterion in ForSyDe, and this effectively renders the unaltered Avalon Switch Fabric useless for our purposes. In 2013 Marcus Mikulcak published his master thesis [7] in which he proposes a modified arbiter for Avalon slaves based on time division multiplexing. As illustrated in Figure 2.4 each master is allotted a fixed number of time slots in each round, whether the master wants to access the slave or not. This minor change makes the access times predictable at the cost of potentially wasted bandwidth, since unused time slots cannot be used by other masters. The TDM arbiter modification is retrofitted to the system that SOPC-builder generates.

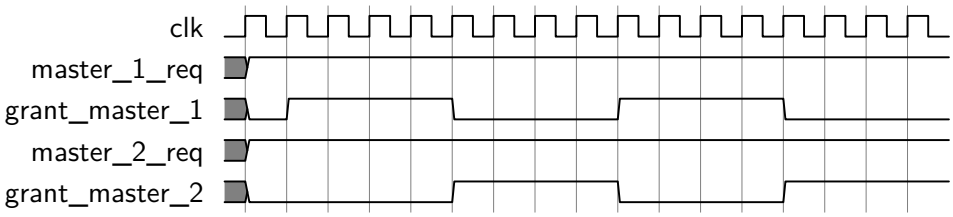

Figure 2.2: Illustration of the Avalon arbitration scheme. In this example each master can make four consecutive transfers, and when it has used them up, access is granted to other masters. Signals and timing are illustrative only, and do not necessarily correspond to real signals.

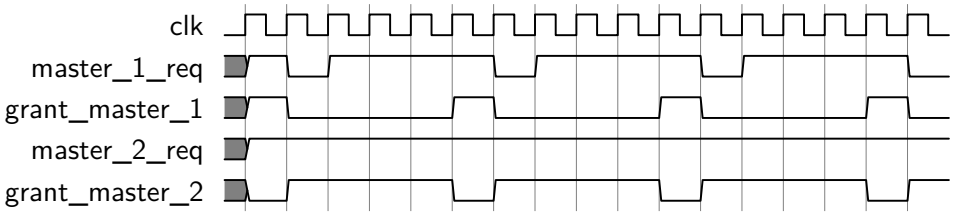

Figure 2.3: If one master only wants to transfer once, it forfeits the rest of his transfers for this round and has to wait until all other masters have used up or forfeited their transfers before being granted more transfers. In this example master_1 can only send for 20% of the time instead of 50% as in the example in figure 2.2. The system is not composable.

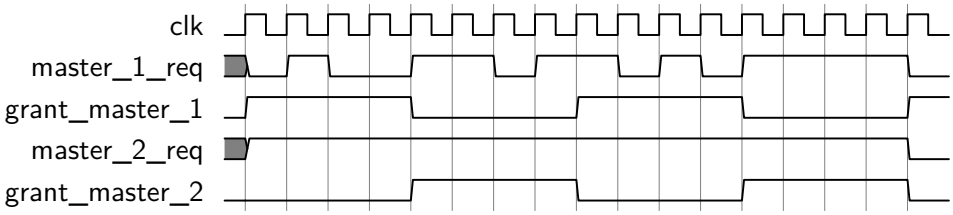

Figure 2.4: Illustration of how the TDM-arbiter makes the masters independent of each other. Regardless of when and how often a master requests to transfer, each master is always guaranteed its number of transfers, in this case 4 transfers per 8 transfer cycles. Again, these signals are for illustrative purposes and do not correspond to actual hardware signals.

2.5 Motivation

The motivation for this thesis is that as the ForSyDe research project has moved along and grown, the whole tool chain is no more only separate bits or pieces, but starts to resemble a chain. As such the whole chain has not yet been thoroughly tested. Not all links in the chain are finished though, parts that are not yet done include the automatic code extraction, but sufficiently large portions of the system is in a mature enough state to begin to be tested.

This thesis aims to help in validating and assessing one vital part of the complete ForSyDe ecosystem, namely the PSOPC system.

More precisely, this thesis aims to provide a simple application, from which to derive what parameters control inter-actor communication time in a PSOPC system. These measured metrics will provide a benchmark for the DSE tool, and will allow for seeing whether the projected performance data is in line with observed data.

Chapter 3

3 Describing Systems in PSOPC

This chapter describes the function of the PSOPC scripts. Instructions on how to obtain a copy of PSOPC was at the time of publication available at [1]. The author of this thesis has used a Linux virtual machine on VMware Player, running Ubuntu 14.04 LTS and Quartus version 12.1 with the *University Program* modules installed. The FPGA used is the Altera Cyclone IV (EP4CE115F29C7N) on the DE2-115 evalution board.

3.1 Overview of PSOPC

The PSOPC system was created to automate the stages after the DSE phase. In its current condition the PSOPC system creates the physical layer of the complete system model, meaning that it generates all entities marked with bold lines in Figure 3.2. In the future, PSOPC is meant to also create the parts within each processor, marked with dashed lines, but this is currently not implemented [7].

It is assumed that, before running these tools, the system designer has already defined the system model by use of one or more of the supported MoCs, and it is furthermore assumed that this model is an SDF process network. ForSyDe supports more MoCs, but [7] focuses mainly on providing support for SDF.

The DSE mapping will deliver a hardware description in an abstract form, similar to the example in listing 3.1, and PSOPC will transform that abstract description in two simple steps with two easy to use scripts.

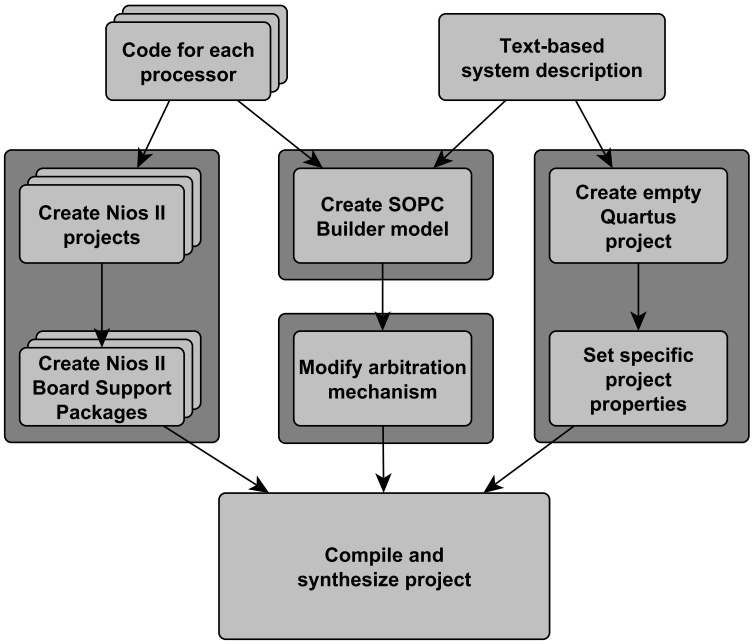

Figure 3.1: Overview of the PSOPC system. The PSOPC system takes as input a textual description of a system and creates a SOPC builder project. The arbiters created by SOPC Builder are then modified into TDM arbiters. PSOPC also generates a Quartus project and compiles the actual hardware image. This image is then transferred onto the FPGA and all processors are started. Finally PSOPC creates a software skeleton in C for each processor. (Image adapted from [7])

```
<dse_output>
  <!-- process 0, mapped to CPU 0, one SDF channel to process 1 -->
  <process id="0" cpu="0">
    <channel>
      <!-- channel from process 0 to process 1 -->
      <to_process>1</to_process>
      <!-- tokens have a size of 32 bytes -->
      <token_size>40</token_size>
      <!--the buffer size in number of tokens on the receiving side-->
      <buffer_size>4</buffer_size>
    </channel>
    <!-- the memory requirement of the process on the processor -->
    <local_memory_consumption>32768</local_memory_consumption>
  </process>

  <!-- process 1, mapped to CPU 1, one SDF channel to process 0-->
  <process id="1" cpu="1">
  <!-- channel from process 1 to process 0 -->
      <to_process>0</to_process>
      <!-- tokens have a size of 60 bytes -->
      <token_size>60</token_size>
      <!--the buffer size in number of tokens on the receiving side-->
      <buffer_size>4</buffer_size>
    <!-- the memory requirement of the process on the processor -->
    <local_memory_consumption>32768</local_memory_consumption>
  </process>
</dse_output>
```

Listing 3.1: dse_out.xml *A simple example of a DSE output in XML format.*

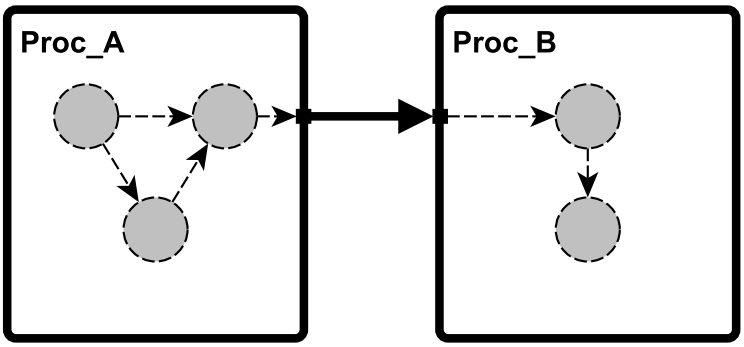

Figure 3.2: PSOPC generates the boldly marked entities, while currently the dashed parts are left to the system designer to implement manually.

3.1.1 Directory Structure

It is suggested that one keeps a certain directory structure while working with the PSOPC environment. This thesis proposes the following structure:

```
PSOPC
├── built_systems
├── documentation
├── projects
└── source_code
```

where *documentation* contains doxygen generated documentation which one may optionally generate[1]. The *source_code* directory contains all the PSOPC system files, *projects* should contain the projects in the form of the input files to the system generation, while *built_systems* will contain the generated output.

Remark! For PSOPC to put the generated system files into the *built_systems* folder, one has to modify the *create_quartus_project.py* script. The version of the script used in this thesis contains hardcoded paths which needs to be altered to match the directory structure of each particular installation.

[1] See PSOPC web page [1] for detailed information on how to generate the documentation

3.1.1.1 The *source_code* Directory

The structure of this directory looks like the following:

```
source_code
├── dse_output_processing
│   └── dse_to_architecture.py
└── system_creation
    ├── create_quartus_project.py
    └── lib
```

The file *dse_output_processing/dse_to_architecture.py* is the script that transforms the XML file from the DSE mapping into an intermediate hardware description that we need as input to the system generator script. The directory *system_creation/lib* contains all supporting files and scripts that make up PSOPC, and the file *system_creation/create_quartus_project.py* is the script one runs to generate the system.

3.1.1.2 The *projects* Folder

A new project directory should initially only contain the DSE output in the XML form. This is here illustrated with an example project:

```
projects
└── example_project
    └── dse_out.xml
```

After the *dse_to_architecture.py* script has been run with the XML file shown in listing 3.1 the directory will look like:

```
projects
└── example_project
    ├── cpu_0
    │   ├── buffers.h
    │   └── cpu_0.c
    ├── cpu_1
    │   ├── buffers.h
    │   └── cpu_1.c
    ├── dse_out.xml
    └── system_description.xml
```

This project is now ready to be built by the *create_quartus_project.py* script.

3.1.1.3 The *built_systems* directory

For every project that the *create_quartus_project.py* script is successfully run for, a large number of files and directories gets created. However, in particular, one directory is more relevant than anything at this point. Inside *built_systems/example_project/* a directory named *software*, containing two directories for each processor instantiated in the system, can be found. For the example project discussed here, this would be:

```
built_systems
└── example_project
    └── software
        ├── example_project_cpu_0
        ├── example_project_cpu_0_bsp
        ├── example_project_cpu_1
        └── example_project_cpu_1_bsp
```

The directories with names ending with *bsp* contain the *board support package* or BSP for each processor. The other directories contain only a single *Makefile*. The purpose of this file will be discussed later.

3.1.2 Workflow

In short, the steps to take to produce a running system are the following, illustrated with the *example_project*:

1. **Preparations:** Attach the FPGA board and start a Nios II command shell with:

   ```
   $ nios2_command_shell.sh
   ```

2. **Create system description from DSE output:** The following commands generates the file *system_description.xml* as well as source code skeletons for each CPU.

   ```
   $ cd PSOPC/source_code/dse_output_processing/
   $ python dse_to_architecture.py --input_file ../../projects/
     example_project/dse_out.xml
   ```

3. **Edit the system description file:** After *system_description.xml* has been created, this file has to be manually modified. It is located in the directory *PSOPC/projects/example_project/*. The TDM arbiter slot length needs to be set to desired value. Also, all slot assignments for the memories are unset by default, meaning that no master will ever be granted access to them. This has to be changed to reflect the slot division wanted for your particular system.

   Considering a system with one communication channel from CPU 0 to CPU 1, and using a TDM slot length of 10000 cycles, one would need to modify the following line:

```
<!--Communication memories-->
<module kind="altera_avalon_onchip_memory2" name="
  shared_memory_process_0_process_1" size="512" tdma_slot_length
  ="5000" cpu_0_data_master_slots="---"
  dma_cpu_0_write_master_slots="---" cpu_1_data_master_slots
  ="---"/>
```

   into this:

```
<!--Communication memories-->
<module kind="altera_avalon_onchip_memory2" name="
  shared_memory_process_0_process_1" size="512" tdma_slot_length
  ="10000" cpu_0_data_master_slots="X--"
  dma_cpu_0_write_master_slots="-X-" cpu_1_data_master_slots="--X
  "/>
```

4. **Build the system:**

```
$ cd PSOPC/source_code/system_creation/
$ python create_quartus_project.py --project_name
  example_project --system_description ../../projects/
  example_project/system_description.xml --sopc --source_code
  ../../projects/example_project/
```

   This will prepare and invoke all the Altera tools used to actually build the system. This script can take considerable time to finish, but should provide lots of textual feedback as to what is going on. A more detailed description of what the script does can be found in [7]. The last thing this script does is to download the software onto your attached FPGA.

5. **Start terminals:** To be able to monitor the output from the programs you will run on the processors, we need to start terminals that show the output of each processors JTAG UART. Each processor has a unique instance ID associated with its JTAG. For cpu_0 this should be number 0, and for cpu_1 it should be the number 1. In our example you would do the following:

```
$ xterm -name ''CPU_0'' ''/path_to_altera_tools/altera/12.1/
  nios2eds/bin/nios2-terminal --instance=0'' &
$ xterm -name ''CPU_1'' ''/path_to_altera_tools/altera/12.1/
  nios2eds/bin/nios2-terminal --instance=1'' &
```

Notice the trailing *ampersand* &, which lets you continue working in the shell after you invoke a command that does not exit.

6. **Write the C source code:** PSOPC has already created created skeleton source code files for all processors. Each processors source code is found in a subdirectory of the project directory. At this point cpu_n.c contains:

```
/** @file
 * @brief The main source file for CPU n.
 */
#include "buffers.h"

int main()
{
    initialize_buffers();

    //declare communication variables here

    while (1)
    {
        //receive_token(sender, receiver, pointer_to_data);
        //call process functions here
        //send_token(sender, receiver, pointer_to_data);
    }
}
```

To have processor 0 continuously send data to processor 1, you would just make a call to send_token(0, 1, pointer_to_data_to_send) at the end of the while loop in cpu_0.c, and a call to receive_token(0, 1, pointer_to_where_to_store_data) at the beginning of the while loop in cpu_1.c. The call to receive_token() blocks until there is data to receive, and the call to send_token is blocking if the channel buffer is full.

7. **Compile and download the C source code:** For each processor we now compile and download the source code:

```
$ cd built_systems/example_project/software/
  example_project_system_cpu_0
$ make
$ make download-elf
$ cd ../example_project_system_cpu_1
$ make
```

```
$ make download-elf
```

If you at this point encounter any bugs, you just reiterate step 5 and 6 until you are satisfied.

3.2 System Model

This section presents how to transform your system model into the proper input format for the workflow.

3.2.1 DSE Output Format

In his thesis [7], Mikulcak defines a syntax that shows one possible way of interacting with the PSOPC system. It is a simple XML format with only a handful of tags defined. You define the processes and the communication channels between them. Listing 3.1 on page 18 shows one example of a system.

The root element is called *dse_output* and it contains all of the *process* elements. For each actor in your SDF graph you define one *process*. Every *process* has two attributes, a unique number called *id*, and a number called *cpu* indicating which cpu the actor belongs to.

Each *process* element contains one mandatory child element called *local_memory_consumption* whose content is the number of bytes required by the process for stack and heap data.

Each process also contains zero or more *channel* elements, which correspond to the outgoing arcs of the SDF graph. A *channel* element has three child elements, *to_process*, *token_size* and *buffer_size*. The content of the *to_process* element is an integer number corresponding to the unique *id* of the receiving process. The content of the *token_size* element is the integer number of bytes required to hold one token. The content of the *buffer_size* element is an integer corresponding to how big the recieving buffer needs to be according to the schedule set by the DSE phase.

For each CPU mentioned in the *process* elements, the system will create a *processing element* consisting of one Nios II/e with a *performance counter* useful when profiling your code, one *JTAG UART* interface for communicating with a PC host machine, a block of local memory to facilitate fast and predictable access to local data and instructions and, maybe most importantly, one DMA controller responsible for sending and recieving tokens. The DMA is an important integral part of the PSOPC system, since it serves to decouple the execution of actors from the communication between actors.

For every *channel* element with token sizes larger than 4 bytes, the system will create one shared on-chip memory block along with a modified TDM arbiter. For channels with a token size of 4 or less bytes, PSOPC will instantiate a hardware FIFO buffer instead. The worst case communication cycle bound will decrease, but the use of hardware FIFOs over software FIFOs is transparent to the programmer. Both use the same C functions to send and receive tokens.

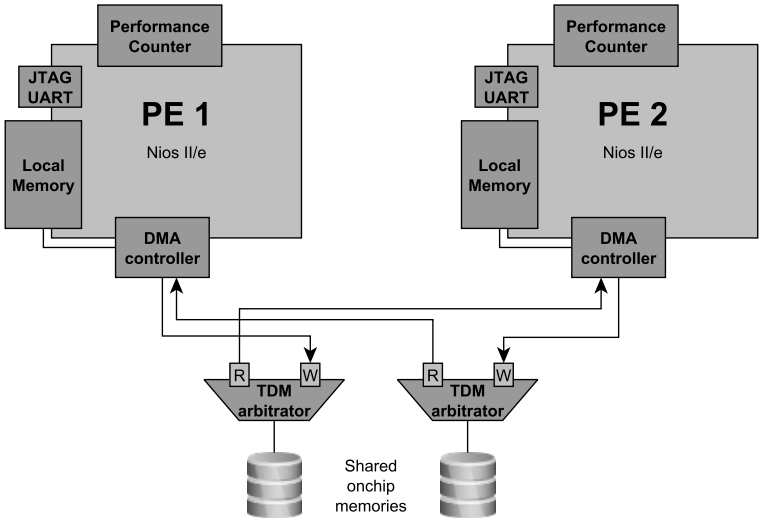

Figure 3.3: For each processing element, a full Nios II/e processor core is instantiated along with Performance Counter, JTAG UART, local memory and a DMA controller. For each communication channel the system instantiates a dedicated block of shared on-chip memory with modified TDM arbiters. The size of these memories are instantiated to be exactly $t_i \times b_i$ bytes, where t_i is the number of bytes per token for channel i and b_i is the length of the buffer for channel i. Image adapted from [7]

Figure 3.3 shows the architecture which would be created from the example in listing 3.1.

3.2.2 PSOPC Input Format

It is convenient to only use the format presented in the previous section. However, some degree of control is lost if one only works on this level of abstraction. For instance, in the previously presented format you have no control over the TDM arbiter settings, such as slot length and slot allocations. There may also be more modules that one may want to connect to the system. The currently supported modules are:

- Nios II/e processors
- On-chip memory blocks
- SRAM controller
- JTAG UARTs
- Performance counters
- PIO modules

- DMA controllers
- FIFO memories

As the platform matures there will probably be more modules to choose from, such as DSPblocks and video signal generators etc.

To be able to fully customize the output of the PSOPC system generator, one would work with the *system_description.xml* file located in the project directory. The syntax is closely related to how SOPC builder natively describes systems. Consult [7] for further reference on this format.

3.3 Graphical Representation of the System

The PSOPC system generator also automatically generates a diagram showing the system. An example is shown in Figure 3.4. These diagrams is in a format called *dot*. These files can be rendered into an image by use of one of several programs in the Graphviz package. The syntax on a Linux machine to render the diagram as a png file would, provided that you have installed Graphviz properly, be:

```
$ cd built_systems/example_project/
$ dot -Tpng example_project.dot -o example_project.png
```

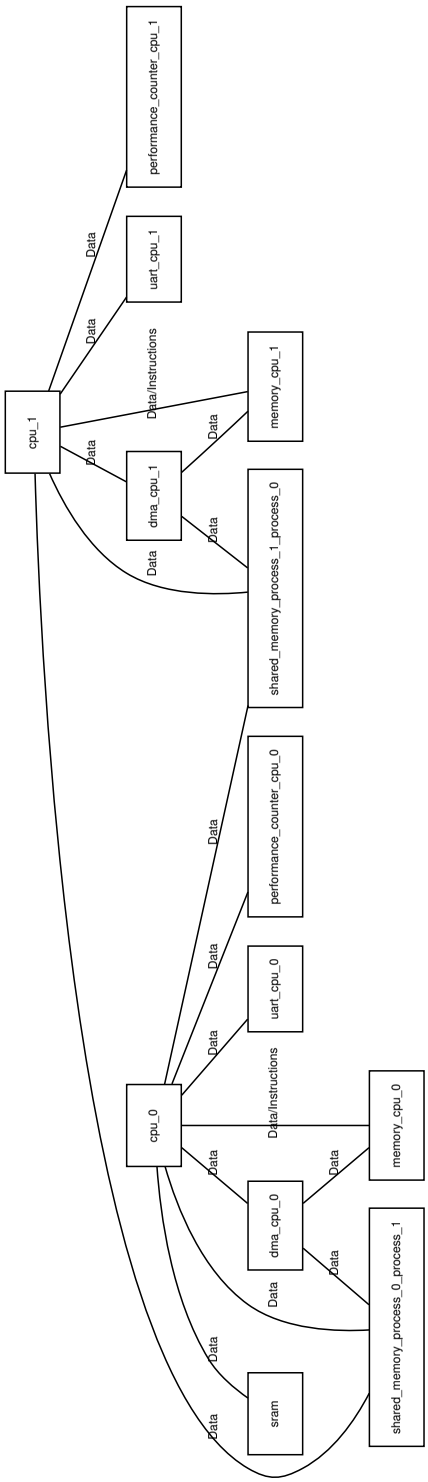

Figure 3.4: This graph represents the test systems created for this thesis. PSOPC automatically generates a graph depicting the system and its modules. The graph is readily available in the Graphviz dot-format after the running of the PSOPC create_quartus_project.py script. Use one of many tools in the Graphviz package to convert into an image.

Chapter 4
4 Contribution

This chapter describes the application created for this thesis, the measurement data collected as well as the analysis of this data.

4.1 Method

In order to be able to assess the PSOPC system, a simple system was created. This system is comprised of two PSOPC processes mapped onto two different processors. These processes communicate by use of the PSOPC token passing mechanism. The two processors runs a simple SDF-applications, whose only purpose is to measure and report the time it takes to send tokens from CPU 0 to CPU 1 and back again. The application also checks data integrity by comparing the data being sent with the data being received.

4.1.1 Test System Generator Script

To facilitate fast and easy testing of many different system configurations a script was created. This script called *tpsopc.sh* ("Test PSOPC") takes three parameters. The parameter **-t** controls the size in bytes of each token being sent, the parameter **-b** controls how many tokens the FIFO buffer shall be able to hold, and the parameter **-s** controls the length in cycles of a TDM slot. The script then automates the whole test run with the exception of one human intervention due to a bug in PSOPC or possibly SOPC Builder. According to Mikulcak [7] each memory module must have a base address which is divisible by its address span, i.e. a memory of size 2048 bytes needs to have a base address that is divisible by 2048. The algorithm in use is not always able to produce valid results. When this happens, the system designer has to manually assign base addresses. This error appeared for all systems created during the work of this thesis, but was always easily remedied by making use of the built in feature of the graphical user interface in SOPC builder to automatically generate

base addresses. After these steps PSOPC handles the creation and instantiation of the system automatically.

When the system has been synthesized, downloaded onto the FPGA and booted, the test system generator script proceeds to open up an UART connection to CPU 0, redirecting its output to a log file. After that the script compiles and downloads the two C programs onto the two Nios II processors, after which the programs are left running to completion. The script then waits for the UART connection to terminate, which signals that the programs has reached the end. It then extracts the data from the log file and formats it for easy input in GNU Octave. The script finishes by playing a telephone signal to let the operator know it has finished and is ready for another test run. The whole process takes about five minutes on the particular computer and virtual machine in use for this thesis.

4.1.2 Test Template - Hardware

```
1  <dse_output>
2          <process id="0" cpu="0">
3                  <channel>
4                          <to_process>1</to_process>
5                          <token_size>REPLACE_1</token_size>
6                          <buffer_size>REPLACE_2</buffer_size>
7                  </channel>
8                  <local_memory_consumption>65536</
       local_memory_consumption>
9          </process>
10
11         <process id="1" cpu="1">
12                 <channel>
13                         <to_process>0</to_process>
14                         <token_size>REPLACE_1</token_size>
15                         <buffer_size>REPLACE_2</buffer_size>
16                 </channel>
17          </process>
18 </dse_output>
```

Listing 4.1: Template dse_out.xml *This template is used by the test generator script to generate communication channels of arbitrary dimensions. The strings "REPLACE_1" and "REPLACE_2" are substituted for the input parameters to the script.*

As suggested by above listing (4.1), the test generator script generates a simple system, consisting of only two processors and two communication channels. The dimensions of the channels is governed by the parameters supplied to the script upon invocation. The script then execute the same steps as described in Section 3.1.2 and performs the same modifications of the *system_description.xml* file as described in Section 3.1.2.3, by use of the *sed* command and regex pattern matching.

The TDM arbiter slot table for the two channels will be identical in function since the same exact data is passed through both channels. There are three masters connected to each shared memory block; the sending CPU, the sending DMA and the receiving CPU. Consequently, the table have three entries. One TDM round is then $3 \times s$, where s is the number of cycles specified with the **-s** parameter to the test generator script. The transmission of one token involves all three masters, and each master has to wait at most $2 \times s$ before it is granted access to the memory.

4.1.3 Test Template - C Source

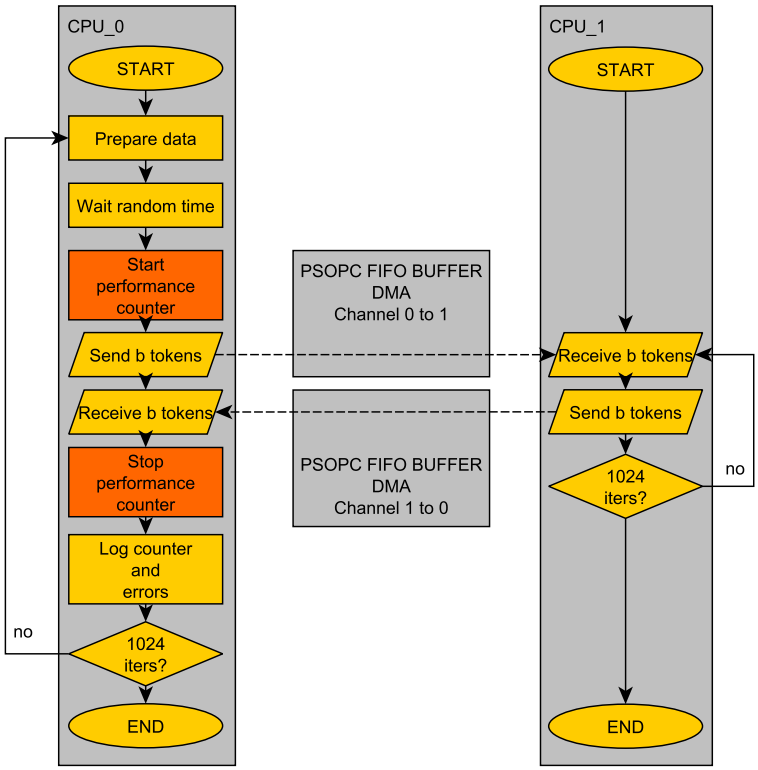

Figure 4.1: Flow chart representation of the system testing program. The program on CPU 0 starts by generating random tokens. It then waits some random amount of time to minimize unintentional timing alignment anomalies. After the wait CPU 0 sends as many tokens as specified with the -b parameter to the script. CPU 1 receives these tokens, and sends them back right away. CPU 0 receives the tokens, compares them byte by byte with what the script sent. If there exists any discrepancies between sent and received bytes, an error is logged, else the total number of cycles spent on sending and receiving bytes are logged. The process is repeated 1024 times.

Figure 4.1 shows the structure of the test program. It performs the same sequence 1024 times. First it fills the data structure representing tokens with random unsigned

bytes. It then waits a random amount of time to decrease the risk of experiencing timing anomalies due to accidentally aligning with the TDM round each subsequent test. CPU 0 then start the performance counter and proceed to send as many tokens as can be fitted in the buffer to CPU 1. CPU 1 then receives the tokens and passes them back to CPU 0 as soon as all tokens have been received. When in turn CPU 0 has received all tokens two things happen, first the time it took to send these tokens is logged and then a check is performed to see whether the received data matches the sent data. When these don't match, an error is logged.

Two templates have been created. The first template uses a for loop to make the desired number of send and receive calls, while the other template inlines as many send and receive calls as needed without using any loops. The merit for the first template is that the code memory footprint does not change depending on the number of tokens to send and receive. Meriting for the second template is that the measured time only reflects the actual send and receive calls and not the execution time overhead of the loop construct.

4.1.4 Measurements

Although other cost metrics such as number of logic elements is important in the design space exploration, only transmission costs are considered in this thesis. To also assess claims made on how large PSOPC systems are would have broadened the scope of the thesis too much. The most important metric in the context of implementing hard real-time systems as SDF graphs is arguably WCCC, or worst case communication cycles. Ideally you would want to have a known upper bound for every system configuration. In [7], Mikulcak claims that the WCCC for a token transmission can be determined by the following formula,

$$1200 + 48t + max(TDM) \tag{4.1}$$

where $max(TDM)$ for all test cases in this thesis equals $2 \times s$ as explained in Section 4.1.2. According to [7] this formula is valid under the assumption that a transmission can always finish faster than the allotted time slots per TDM round for each participating sending master, i.e. it is a requirement that

$$1200 + 48t < n_i \times s \tag{4.2}$$

where n_i is the number of slots allotted to a master i and s is the number of cycles per slot. If a token transfer takes longer than $n_i \times s$, the sending master will be interrupted and will have to wait for the arbiter to grant another round of access. For all systems created during work on this thesis, each master is only granted one slot, so $n = 1$ for all masters i.

4.1.4.1 Definition of a Transmission

The formula in Eq.4.1 describes the worst case communication cycles for t bytes. Mikulcak is not entirely clear on what constitutes a transmission. The definition used in this thesis is that the transmission time is the time as measured from the instant that we issue the send command, to the instant at which all data to be transmitted resides in the destination memory. Since no exact definition of the word transmission is given in the context of measuring and predicting WCCC in [7], the definition given herein is the definition assumed.

Each CPU has its own performance counter connected to it. In principle it would be possible to synchronize the two CPUs to let CPU 0 determine when the measurement starts, and let CPU 1 determine when the measurement ends. This, however, is not a trivial task since the two processors are truly independent. This would require a deep understanding about how the synthesized hardware behaves at gate level. Therefore a different approach was opted for. CPU 0 is responsible for determining both the start of the measurement as well as the end of the measurement, but instead of measuring single transmissions, measurements are taken on pairs of transmissions. The transmission time of a single transmission is then approximated by dividing the measurement by two. This seems a fair estimate since both transmissions are executed back to back with no operations in between. This might however give lower than true WCCC measurements, since CPU 1 always issues its send command immediately after it has finished receiving every time, and could in theory always be aligned with the start of its arbitration grant slot, thereby potentially reducing the measured WCCC for two back to back transmissions to $2 * 1200 + 2 * 48 * t + 1 * max(TDM)$.

While Mikulcak in [7] is only concerned with token size and slot length, measurements in this thesis contain a third variable, namely buffer length. To send many tokens, some sort of repetition scheme has to be implemented. To achieve that, we could either use a loop, using *for* or *while* constructs, or we can place as many calls one after another until we have sent all tokens. The former adds execution time overhead due to the C runtime having to perform conditional tests on each loop, but presents a plausible real life implementation. The latter does not add execution time overhead, but memory footprint grows proportionally to buffer length. Tests have been conducted using the second method, with the motivation that since the quantity to be measured is execution time, any auxiliary feature that alters the execution time is unwanted. Using the inline method described above, the only measurement error should come from the overhead of starting and stopping the counters, which, according to [3] amounts to 2 to 3 machine instructions per macro invocation.

Another discrepancy between the measurements taken in this thesis and the ones taken in [7] is that Mikulcak did not use the DMAs to transfer data, whereas the measurements presented here were all taken using the DMA to transfer the data.

According to [7], to be able to *"measure the exact time to transfer a token using the implemented send_token and receive_token functions, the DMA controllers are disabled [...] thereby forcing the utilization of memcpy and therefore the processor itself to send and receive tokens"*.

4.2 Observations and Analysis of Data

The analysis is conducted on a limited sample set. This is due to the fact that many of the synthesized systems did not work properly. They exhibited different anomalies including failing to start, dropping tokens, and other effects. These anomalies are discussed more in depth in 4.2.1 To avoid contaminating the sampled times with potentially faulty measurements, all sample sets coming from systems that displayed anomalous behavior was rejected from the data selection.

The initial hypothesis was that the number of cycles required to send data would be a function of three variables, t, number of bytes per token, b, number of tokens per transmission and s, number of cycles per TDM arbiter slot.

4.2.1 Stability Issues

During the data collection phase, it was apparent that some PSOPC generated systems suffered from stability issues. Not all configurations worked. Table 4.1 demonstrates two particularly troublesome group of tests that did not perform satisfactorily. No visible pattern emerged during the work of this thesis that could point to the cause of these problems. The respective tokens that did not make it through did not share any immediately apparent common patterns or discernible features compared to tokens that were transmitted properly. Apart from the configurations listed in Table 4.1, many more configurations were tested. The greater majority of those would work, and in case they did not work satisfactorily, it would be due to the system dropping one or two tokens during a full test. Even though this might seem a small number, it is not acceptable to have any lost tokens at all. Another group of system configuration that did not work during the work on this thesis were all systems where token sizes were 4 bytes or less. In these cases PSOPC instantiates a hardware FIFO instead of a software FIFO. None of the tokens communicated over hardware FIFO was received correctly.

4.2.2 Data Selection

To sample three variables, and to sample with both adequate resolution and range, is not trivial. If we would want n steps in resolution per variable we would have to perform n^3 tests. With one test taking approximately 5 minutes, even with such a

	$s = 1000$					$s = 5000$				
$b\backslash t$	8	16	32	64	128	8	16	32	64	128
1	O	O	O	O	O	O	O	O	O	O
2	!	O	O	O	O	X	!	!	!	O
4	O	!	X	X	O	O	O	!	!	O
8	!	!	X		!	O	O	!	!	O
16	!	!		X	!	O	O	O	X	∞
32	O	O				O	O	X	∞	-

Table 4.1: This table shows particularily troublesome system sets, namely those which had TDM arbiter slot times of 1000 and 5000 cycles respectively. An "O" denotes succesful test, "!" denotes that some tokens were lost, "X" denotes that most or all tokens were lost, "∞" denotes that main started over repeatedly, with only the first printf() statement visible. "-" denotes that nothing happened, the program never started. Even though the systems with $s = 1000$ and the systems with $\{s = 5000, t = 128\}$ do not fulfill the requirement in Eq.4.2, the systems themselves should work, albeit without respecting WCCC requirements. This fact does not explain why some arbitrary systems would not start at all, or lose tokens.

low n as $n = 10$, that is 10 full working days worth of data collection. Since a lack of understanding beforehand in terms of what values for the three variables t, s, b were plausible depictions of real systems, I opted for broad range of samples, in favor of a more limited range of values with higher resolution. What seemed most reasonable was to select the values exponentially, as powers of 2. After having to start over with the data collection because a revision of the test program, and due to the fact that many data sets had to be rejected due to erroneous results, the collected data is rather limited, with approximately 60 000 usable datapoints.

4.2.3 The Effect of Randomizing the Test

The first thing to analyze was if the concern regarding TDM-round aliasing was correct. The hypothesis was that if one naively runs the same test loop over and over again, the length of the loop and the length of the TDM-round could be of such unfortunate proportions that it would falsely alter the distribution of possibly observable WCCCs.

As Figure 4.2 on page 40 confirms, the effect of aliasing is very pronounced once you take away the random wait each loop. The distribution within the randomized sample sets exhibit sharp limits, and provide a much better measure for WCCC than the non-randomized sets.

System configuration	Predicted $(1200 + 48 \times t + 2 \times s)$	Measured
$t = 16, b = 1, s = 4000$	9 968	32 512
$t = 16, b = 1, s = 8000$	17 968	38 488
$t = 16, b = 1, s = 16000$	33 968	50 506
$t = 16, b = 1, s = 32000$	65 968	98 438

Table 4.2: This table shows that the figures given by the formula given in [7] are consistently too low.

4.2.4 Deviating Observations

Some of the sample sets exhibit nonlinearities that cannot be explained by Mikulcaks formula for WCCC. He proposes the linear relationship that $WCCC = 1200+48\times t+ max(TDM)$. Mikulcak has a stochastic term in his formula for calculating WCCC, namely TDM, which is the time a master has to wait for his turn to communicate. As shown in Figure 4.2 (page 40), the distribution of this term is dependent on the calling program. The function max(TDM) however is directly calculable as being the maximum number of allocated slots that can go before the sending master, times the length of one slot. In our system, with only three masters, each having one slot, max(TDM) equals $2 \times s$, which for the uppermost case in Figure 4.4 (page 42) equals 8000 cycles. Figure 4.4 (page 42) show four histograms for the tests of four systems with different TDM slot lengths. In the topmost test case, there is a gap between the group to the left and the group to the right. The gap between the lowest number of communication cycles of the rightmost group, and the highest number of communication cycles of the leftmost group is 18018 cycles. This number cannot be explained by the $max(TDM)$ term. The same kind of gap can be observed in Figure 4.3 (page 41). However, the cases depicted in Figure 4.3 violates the requirement stated in Eq.4.2. The banding effect seen in both Figures4.3 and 4.4 could possibly stem from the fact that a transmissions would not finish within one TDM round, but since $\{t = 16, b = 1, s = 4000\}$ exhibits banding without violating Eq.4.2 that would in such case suggest that the criterion as stated in Eq.4.2 is not strict enough. Another possibility is that this effect is not pertaining to violating Eq.4.2, but something else entirely.

Furthermore, Mikulcak's formula predicts that WCCC for a system $\{t = 16, b = 1, s = 4000\}$ should be 9968 cycles, whereas the histogram clearly shows that both groups lie well beyond this prediction. This is also true for the other configurations depicted in Figure 4.4 on page 42, whose measured WCCC is also well beyond what Eq. 4.1 suggests as shown in Table 4.2.

Looking at Figure 4.5 on page 43 and yet another set of tests, it becomes ever more apparent that Mikulcak's formula for WCCC prediction does not apply to the presumptions in this thesis. The values predicted by the formula are only giving

values between 66 % and 70 % of the measured values. This is not in line with Mikulcak's results in [7] where the predicted WCCC is always pessimistical and between 100.02% and 101.4%. He presents two different sets of measurements, one set where the sending master is granted access immediately, and another where the master has just missed its window and has to wait the full $max(TDM)$ extra cycles. How this behavior was achieved was not clearly explained, and not something that was tried during the tests in this thesis. Rather the waiting times before being granted access is in this thesis considered to be any possible value between 0 cycles and $max(TDM)$ cycles. This behavior is also visible in Figure 4.2 where the distribution of attained measurements is randomly spread out over the a range with sharp limits due to send calls not aligning with TDM grant slots.

4.2.5 The Buffer Length

An initial hypothesis was that not only token size and slot length was a variable in determining communication time, but that also the length of the buffer might play a role. Looking at Figure 4.6 on page 44 we notice that the histogram gets shifted to the right, without altering the distribution and the length of the interval of possible communication times. This means that all transfers were finished within the time slot assigned. This is confirmed by Mikulcak's formula, which if we disregard the $2 \times s$ term, we can calculate that the transfer time, counting from the instant the master is granted access, will be $4 * 1200 + 4 * 48 * 32 = 10944$ cycles, which is well within the alloted 16000.

4.2.6 Another Approximation

When analyzing the data gathered through the work on this thesis, one thing stands out more clear than anything. The difference between measured WCCC and BCCC is almost always approximately $\frac{3s}{2}$. One approach to finding a formula would be to solve some set of linear equations. If WCCC is a linear function $w_{ccc}(t, b, s)$, then, by solving for example the following equation (where $g(t, b, s)$ is a function giving a vector of measurement points),

$$\begin{aligned} Ax &= B, \\ \begin{bmatrix} 8 & 16000 & 1 \\ 16 & 16000 & 1 \\ 32 & 16000 & 1 \\ 64 & 16000 & 1 \\ 128 & 16000 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} &= \begin{bmatrix} min(g(8,1,16000)) \\ min(g(16,1,16000)) \\ min(g(32,1,16000)) \\ min(g(64,1,16000)) \\ min(g(128,1,16000)) \end{bmatrix} \end{aligned} \tag{4.3}$$

we obtain the following expression for WCCC:

$$w_{ccc}(t,b,s) = b \times \left(\begin{bmatrix} t & s & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \right) + \frac{3s}{2} \tag{4.4}$$

Solving for this gave,

$$\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 23.023 \\ 1.6349 \\ 0.00010218 \end{bmatrix} \tag{4.5}$$

We see that the constant term is negligible, so our approximate expression becomes

$$w_{ccc}(t,b,s) \approx b \times \Big(23.023 * t + 1.6349 * s \Big) + \frac{3s}{2} \tag{4.6}$$

4.2.6.1 How Does New Approximation Perform?

Looking at Figure 4.7 on page 45, 4.8 on page 46 and 4.9 on page 47, we see that the data do not indicate that WCCC is a linear function at all. The samples show piecewice linearity, but sample points are too few to rule out polynomial solutions. The system is over constrained and a unique solution was not found. However when $16000 \lessapprox s \lessapprox 32000$, an approximation of WCCC was found to be Eq. 4.6. This approximation however is of no use, since it underestimates the real WCCC. It is a strict criterion that the predicted WCCC is always above real WCCC.

4.3 Conclusion

The sample set was biased towards too low numbers of TDM slot length, where the platform showed considerable instability. It would probably have been better to take more samples with higher values for slot length.

One possible reason to the discrepancy between the results here and the results presented in [7], could stem from the fact that Mikulcak did not use DMA, but handled all communication using the CPU. The author of this thesis argues that any meaningful discussion about systems generated by PSOPC should assume the default state of the generated systems, and the default state is to have the DMA controllers enabled. Furthermore, the author of this thesis argues that the method of measurement proposed in this thesis produces measurements accurate to a real situation, and if using DMA introduces some unacceptable uncertainty, one should maybe reevaluate the current choice of implementation of the send_token and receive_token functions.

An approximation of WCCC was found, but since it underestimates the real figures, it is of no use.

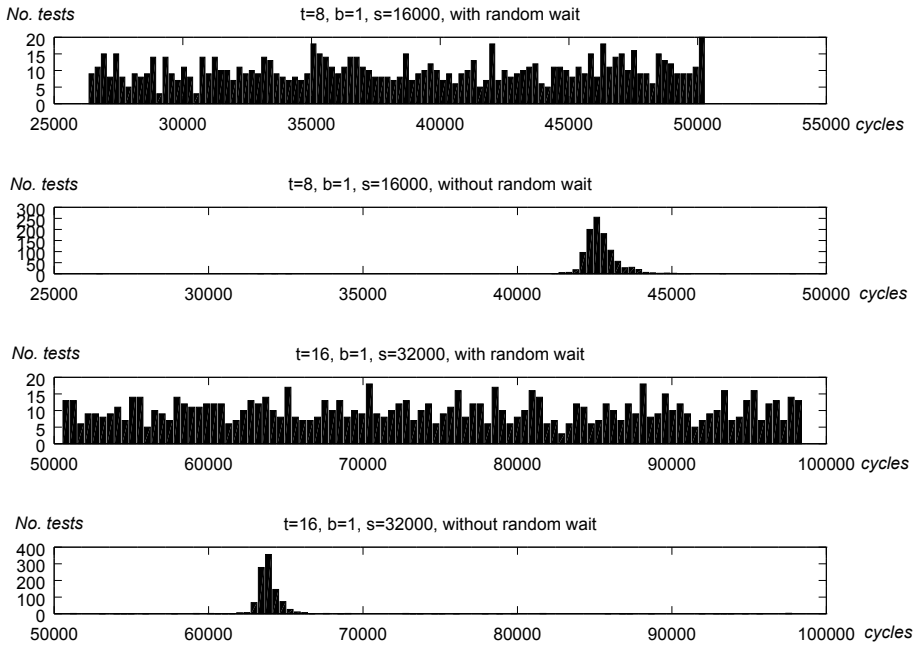

Figure 4.2: This figure clearly shows that the aliasing effect is pronounced if one does not introduce a random waiting time at the beginning of each test loop. The sample sets that were taken with randomization show a uniform distribution with clear upper and lower limits, i.e. WCCC and BCCC. The samples taken without the randomization exhibit clustering around some center, with a slightly skewed bell shaped distribution with comparatively small deviations. These small deviations do not tell anything about where WCCC and BCCC might be situated on the axis.

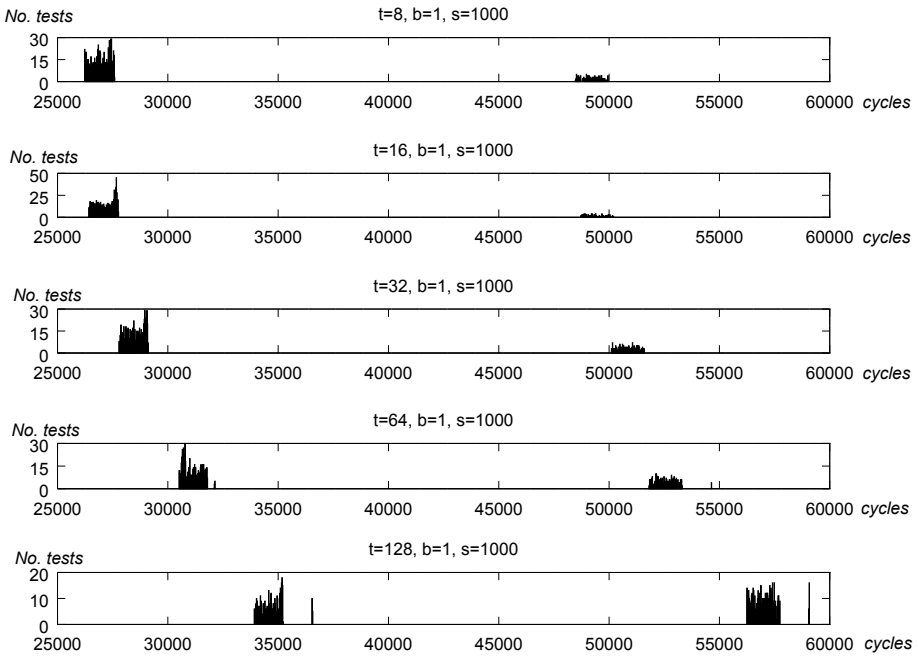

Figure 4.3: The histograms show distributions with varying token sizes. The banding effect appears whenever slot length goes below and including 4000 cycles. This effect might stem from the fact that the time to send a token is greater than the slot length, so the transfer is distributed over multiple TDM-rounds. Why we can observe two distinct groups of transfer times could not be explained during the work with this thesis.

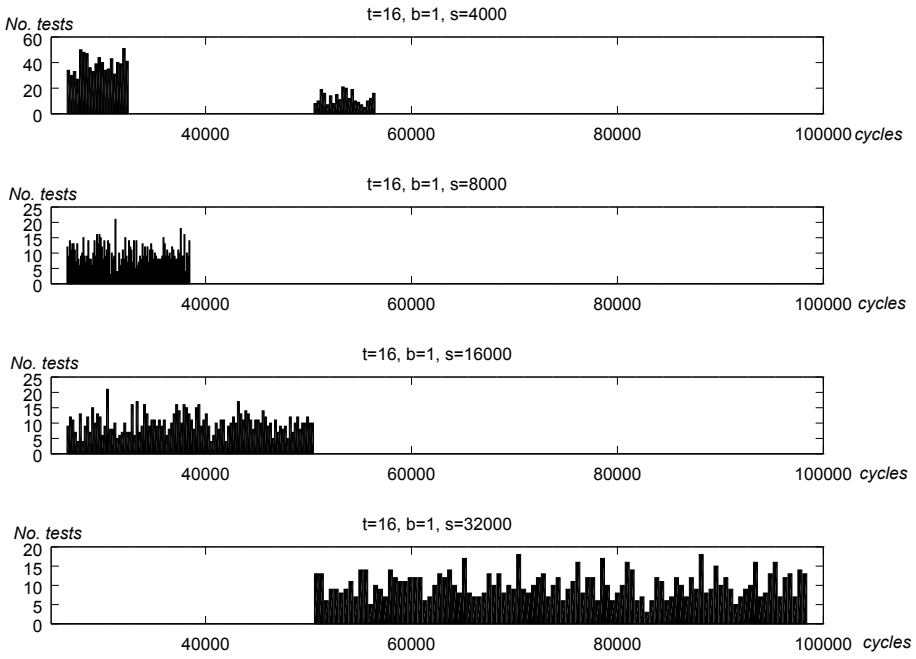

Figure 4.4: This figure shows four histograms for the tests of four systems with different TDM slot lengths. Notice that system $\{t = 16, b = 1, s = 4000\}$ exhibit banding in its distribution. This configuration does not violate the requirement postulated in Eq.4.2, which might indicate that either the requirement or the WCCC formula is not defined correctly or not applicable to these tests, or the banding effect is not directly related to length of a token transfer.

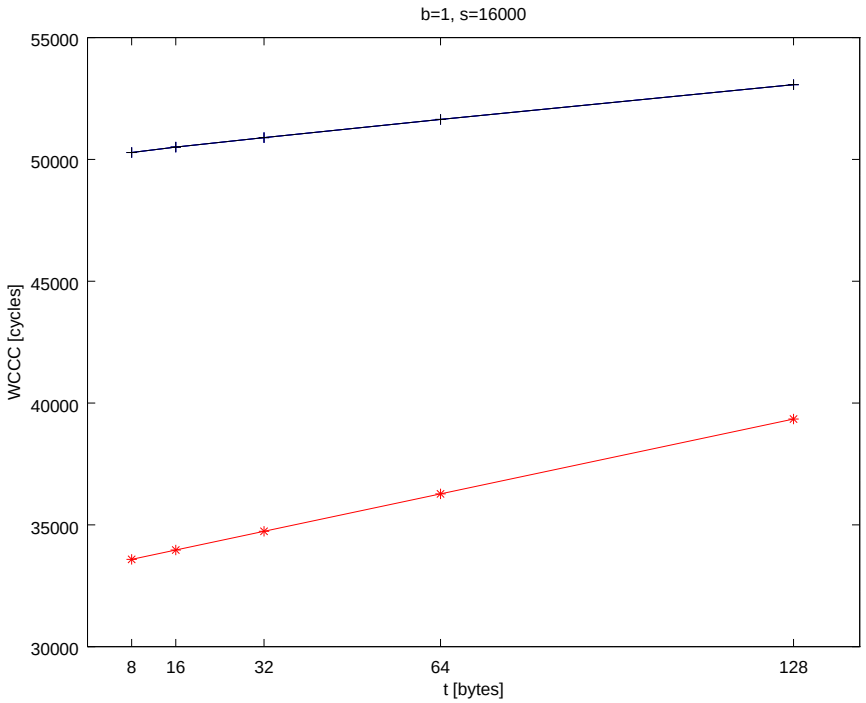

Figure 4.5: Two plots showing WCCC for different token sizes. The lower plot is calculated using the formula in [7], while the higher is sampled data.This plot shows that the formula given to predict WCCC does not accurately depict the real systems.

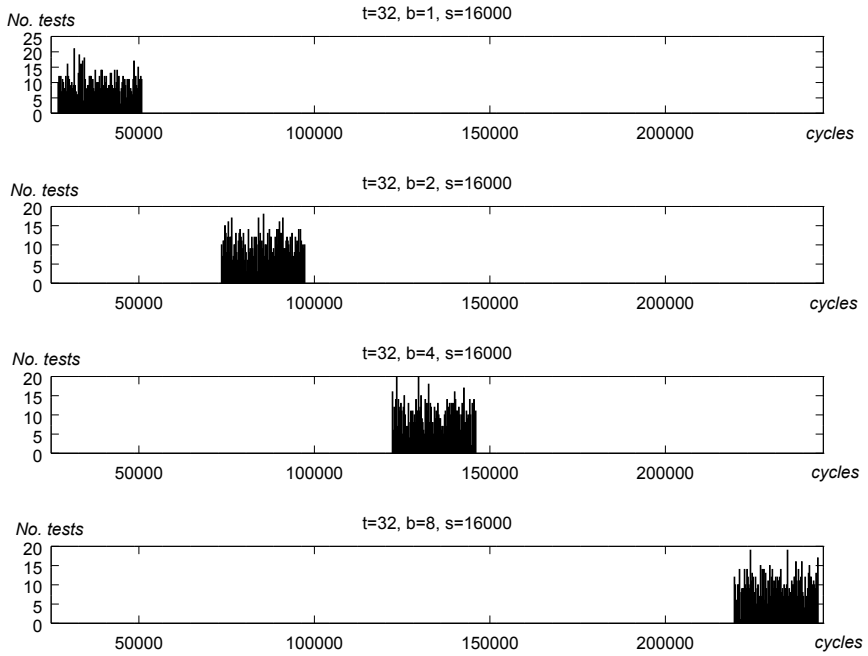

Figure 4.6: Histogram over WCCC vs b. We notice that all data points are translated, but that the distribution and the interval between min and max are intact.

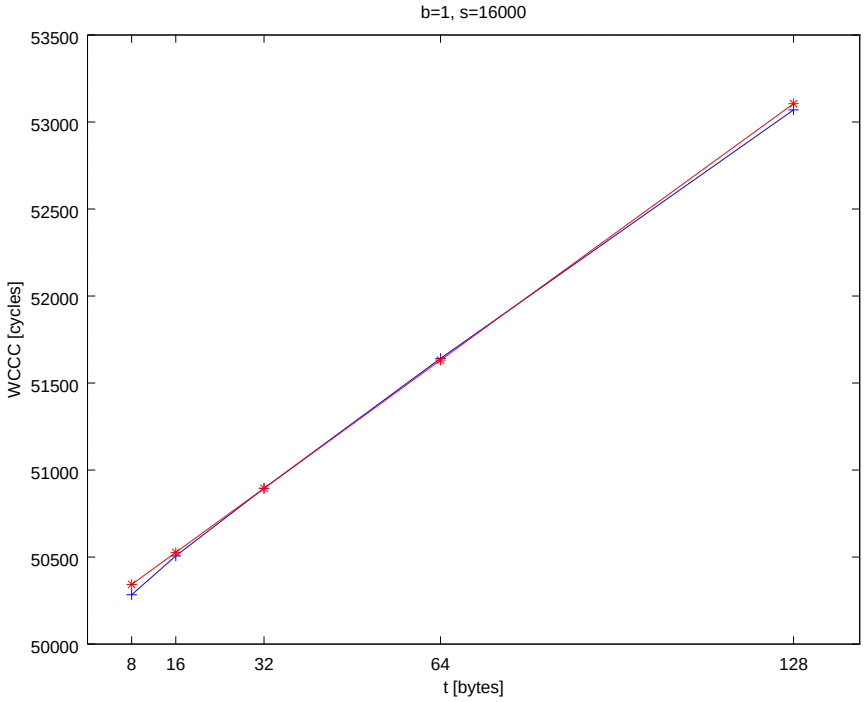

Figure 4.7: This plot shows WCCC for both the revised approximation presented in this thesis, as well as sampled data, as we vary token size. The approximation (star markers) lies close to the measured values (plus markers). The approximation is not strictly pessimistic and is therefore not useful as WCCC.

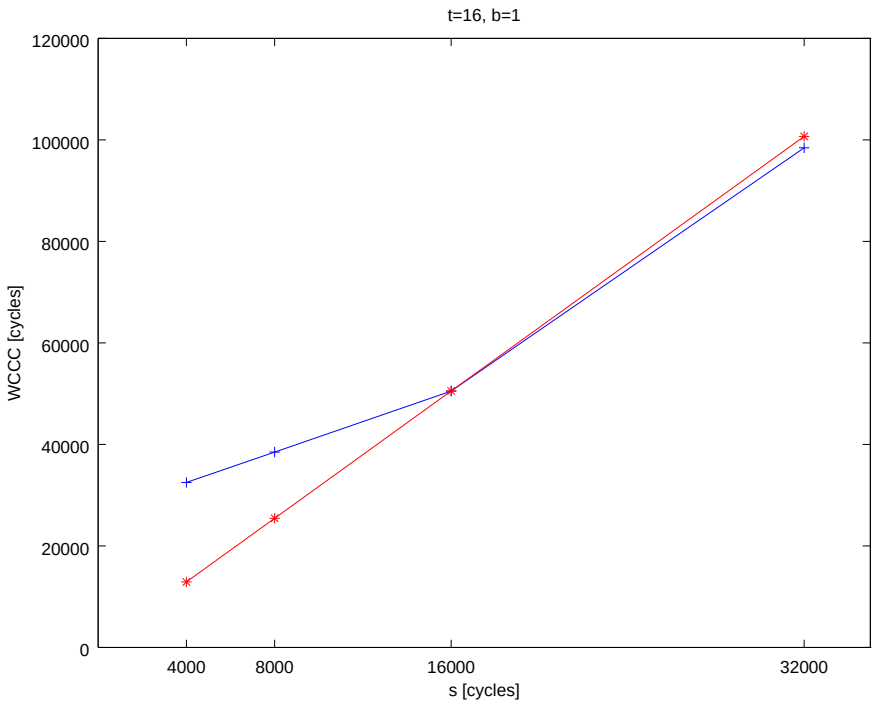

Figure 4.8: This plot shows WCCC for both the revised approximation presented in this thesis, as well as sampled data, as we vary slot length. Here we see that the measured value seems to be piecewise linear. The approximation lies below the measured WCCC, which means that the approximation is not acceptable. Under some circumstances will the real WCCC be higher than what we would expect from the approximation.

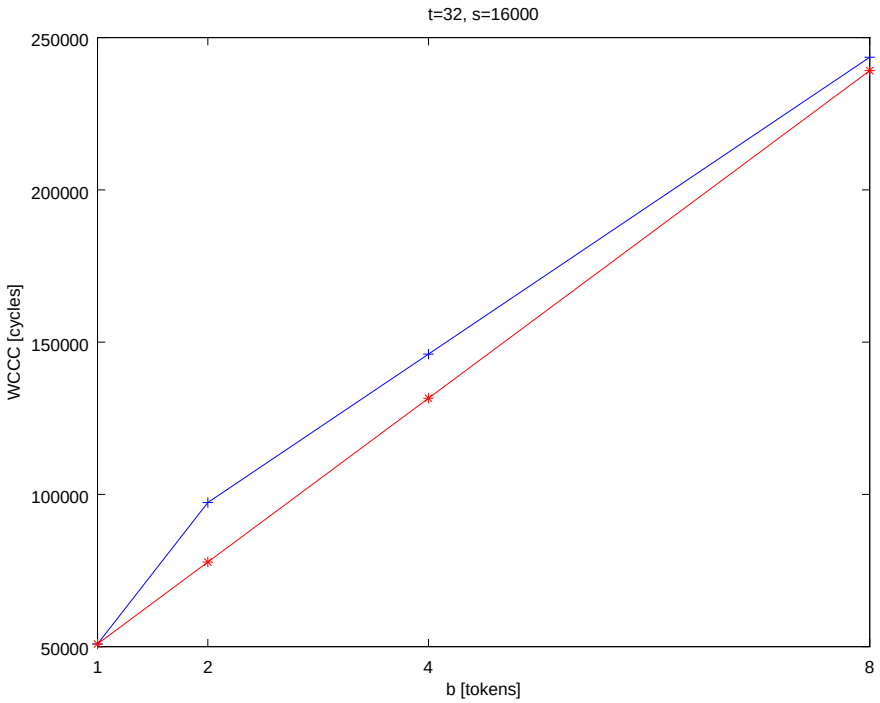

Figure 4.9: In this graph, showing WCCC with varying buffer sizes, the new approximation is far below the true WCCC. It is not a safe estimate, since WCCC always has to be higher than what could ever appear in a real situation.

Chapter 5
5 Conclusions and Future work

The purpose of this thesis was to assess the quality of systems created with PSOPC. The initial idea was to create a multiprocessor system running an application modelled as an SDF graph. This would serve, both as an example of a synchronous data flow application, but also as a benchmark to measure the current DSE tool against. At the start of the work with this thesis, little was known about how well the predicted execution and communication times would compare to real figures measured on real hardware.

5.1 What I Have Not Done

During the course of working with the thesis and trying to implement an example SDF application, lots of problems were encountered. After a lot of work I decided, in dialogue with my supervisors, to put emphasis on trying to measure communication time, rather than to implement a useful application. In the end there was also no time to incorporate the DSE tool's timing projections into this thesis. As a result of this, the collected data and the conclusions derived from it, are left as reference for future work on this subject.

5.2 What I Have Done

I have found that the platform, and systems generated with it, exhibits some anomalies that needs to be investigated further. I have found that the systems with hardware FIFO do not seem to work, and not all configurations with software FIFO will run either, or show problems in regards to being able to maintain the integrity of the data being passed around.

I have further found that the formula for estimating WCCC given in [7] underestimates real measured WCCC under the assumptions of this thesis. Mikulcak's formula (Eq.4.1) is probably only valid with DMA disabled when the processors

handles the transfers themselves. The systems created within this thesis have all had DMA enabled since this is the default behavior for a system created with PSOPC.

In an attempt at finding a better approximation which could be used to describe systems as they generally are created in PSOPC, an approximation was calculated in Section 4.2.6. This approximation is probably only applicable under the conditions discussed here, with two SDF actors communicating with each other. I have further shown that this approximation is not usable as a prediction of WCCC since it is not strictly pessimistic for all configurations, but since it is closer to the measured figures in the studied cases, maybe it could serve as an estimate of transfer speeds, provided that it is tested and verified on more system configurations.

5.3 Future Work

Stability has been an issue during the work on this thesis. It seems that there are aspects of PSOPC that needs to be looked into more thoroughly. But as it stands now, it is a great starting point for further development. The next step should be to try to identify the bugs in PSOPC, such as the packet losses and the fact that some systems did not even run.

It should be pointed out that some aspects of the PSOPC systems was not fully understood by the author of this thesis, particularly with respect to how DMA affects performance. This is in part because [7] offers limited explanation to some of them. It would make sense to redo the experiments in this thesis without the use of DMA to see if the claims in [7] holds under these conditions.

Whenever the bugs have been mitigated and the timing of systems using DMA is better understood the main area of interest should be further automation of the system development process. More specifically, ideally the PSOPC scripts should eventually be automatically invoked after the DSE phase, and the result should be a complete running system. In my view, the goal for a future ForSyDe IDE should be that the only responsibility of the system developer is to construct the overall system model, the code for each process and a set of constraints. Everything else should be automatically generated after that.

For this to become a reality PSOPC has to be extended to be able to handle communication between processes mapped to the same processor, as well as feedback communication from one actor to itself. It also needs to be able to extract and implement both the schedule given by the DSE tool as well as the actual code for each process in the network. Eventually PSOPC should also be extended to support all MoCs that ForSyDe supports.

Bibliography

[1] psopc - Predictable MPSoC Template based on the Altera SOPC Builder (https://forsyde.ict.kth.se/trac/wiki/psopc_platform).

[2] Martin Campbell-Kelly and William Aspray. *COMPUTER - A History of the Information Machine*. BasicBooks, 1996.

[3] Altera Corporation. Profiling Nios II Systems, July 2011.

[4] Altera Corporation. SOPC Builder Design Optimizations, 2011.

[5] Altera Corporation. Nios II Classic Processor Reference Guide, 2014.

[6] Edward A. Lee and David G. Messerschmitt. Synchronous data flow. In *Proceedings of the IEEE*. IEEE, September 1987.

[7] Marcus Mikulcak. Development of a Predictable Hardware Architecture Template and Integration into an Automated System Design Flow (TRITA-ICT-EX-2013:138). Master's thesis, KTH Information and Communication Technology, Sweden, 2013.

[8] Kathrin Rosvall and Ingo Sander. A Constraint-Based Design Space Exploration Framework for Real-Time Applications on MPSoCs. In *DATE '14 Proceedings of the conference on Design, Automation & Test in Europe*. Dresden, Germany, March 2014.

[9] Ingo Sander. *System Modeling and Design Refinement in ForSyDe*. PhD thesis, Royal Institute of Technology, Department of Microelectronics and Information Technology, Sweden, 2003.

[10] Ingo Sander and Axel Jantsch. Development and application of design transformations in ForSyDe. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, volume 23. IEEE, January 2004.

[11] Sundararajan Sriram and Shuvra S. Bhattacharyya. *Embedded Multiprocessors: Scheduling and Synchronization, Second Edition*. Taylor & Francis Group, LLC, 2009.

TRITA-ICT-EX-2015:104