HIGH-LEVEL SYNTHESIS OF ELASTICITY: FROM MODELS TO CIRCUITS

A thesis submitted to the University of Manchester for the degree of Doctor of Philosophy in the Faculty of Engineering and Physical Sciences

2016

Mahdi Jelodari Mamaghani
School of Computer Science
# Contents

Abstract  
Declaration  
Copyright Statement  
Acknowledgements  

## 1 Introduction

1.1 High-level Synthesis  
1.2 Elasticity  
  1.2.1 How Elasticity Contributes  
  1.2.2 What Degree of Elasticity is Enough?  
1.3 State of the Art  
1.4 Aim of this Research  
1.5 Contribution of this Research  
1.6 Thesis Organisation  
1.7 Peer-reviewed Publications  

## 2 Related Work

2.1 High Level Synthesis Flows  
  2.1.1 High Level Control-Driven Synthesis  
  2.1.2 Towards The Synchronous Behaviour  
  2.1.3 High Level Data-Driven Synthesis  
2.2 Popular Synchronous Synthesis Flows
3 Background

3.1 The Teak Synthesis System ........................................... 51
  3.1.1 Input CSP-based Language: Balsa ............................ 51
  3.1.2 Syntax-directed Synthesis of Balsa ............................ 54
  3.1.3 Teak Architectural Properties ................................. 55
  3.1.4 Teak Visualisation Engine ..................................... 58

3.2 Asynchronous vs. Synchronous Elasticity ......................... 60
  3.2.1 Elastic Blocks .................................................. 60
  3.2.2 Elastic Channels ............................................... 61
  3.2.3 Elastic Protocols .............................................. 61
  3.2.4 Back-pressure .................................................. 62
  3.2.5 Non-linear and Conditional Pipelines ....................... 63
  3.2.6 Elastic Components ............................................ 64
  3.2.7 Clustering Elasticity ......................................... 65
  3.2.8 Conclusion ..................................................... 66

4 eTeak: Synchronous Elastic Dataflow Synthesiser .................. 67

4.1 eTeak Synthesis Flow at a Glance ................................. 68

4.2 Synchronous Elastic Components of eTeak ......................... 69

4.3 SELF-Adapted Macromodules of eTeak ............................. 72

4.4 Advantages .......................................................... 74

4.5 Challenges ........................................................... 77

4.6 Using eTeak for Debugging Purposes ............................... 81

4.7 Motivating Example .................................................. 81

4.8 Case Study: SSEM Processor ....................................... 84

4.9 Conclusion ........................................................... 86
## 5 eTeak Dataflow Re-synthesis

5.1 Synchronous Sequential Machines ............................... 88

5.2 Transforming eTeak Dataflow Network to SSM ............... 89

5.2.1 Code Generation: The Algorithm ............................ 91

5.2.2 RTL Implementation of eTeak Dataflows .................... 92

5.2.3 A Case Study: Sparkler Shifter Unit ....................... 95

5.3 RTL for the Shifter Case Study ............................... 97

5.4 Synchronous Machines to Patient Systems ................... 100

5.5 Conclusion ..................................................... 102

## 6 De-elastisation: from Elasticity to Local Rigidity .......... 103

6.1 Introduction .................................................... 104

6.2 Elasticity for Handling Uncertainty ......................... 105

6.3 De-elastisation .................................................. 106

6.3.1 Loops in eTeak Dataflow Networks ....................... 109

6.3.2 The Intuition ................................................ 110

6.3.3 Algorithm .................................................. 112

6.3.4 Potential Improvements ................................. 115

6.4 De-Elastisation vs. Elastic Circuits ......................... 115

6.4.1 SSEM Iterative Processor ................................. 116

6.4.2 Sparkler Pipelined Processor ............................ 116

6.4.3 Bitonic Sorter ............................................ 117

6.4.4 Results and Discussion ................................. 118

6.5 De-elastisation: A System-level Design Flow ............... 121

6.5.1 Patterns for Design Space Exploration .................. 122

6.5.2 De-elastisation vs. Synchronous Synthesis Flows ....... 126

6.5.3 Benchmarks ............................................... 126

6.5.4 Experimental setup ...................................... 127

6.5.5 Results ................................................... 130

6.5.6 Discussion ............................................... 131

6.6 Conclusion .................................................... 135
7 Conclusions and Future Work

7.1 Summary of the Thesis

7.2 Contributions

7.2.1 E-Chart

7.3 Concluding Remarks

7.4 Directions for Future Research

7.4.1 Energy-centric Synthesis of GALS Systems

7.4.2 eTeak as a Backend for the “Go” Language

7.4.3 Parameterisable Synthesis and Simulation

7.4.4 Synchronous Dataflow Synthesis onto multiple FPGAs

7.4.5 Heterogeneous Synthesis of the Balsa Language

Bibliography
List of Tables

6.1 De-Elastisation algorithm applied over a set of general purpose processors 121
List of Figures

1.1 Gajski’s Y-Chart for discriminating the levels of abstraction. . . . . . . 22

1.2 Synchronous vs. Elastic Protocols: Three different realisation of a 3-
input Adder where a) follows a rigid clocked synchronous protocol, 
b) adopts an asynchronous handshake protocol to enable timing vari-
ation tolerance and c) exploits synchrony and asynchrony by using the 
synchronous elastic protocol which discretises the timing variations, yet 
resilient against them. . . . . . . . . . . . . . . . . . . . . . . . . . . . 23

1.3 Control- vs. Data-driven realisation of the input program: A) Control-
driven model where instructions are aligned based on a global sig-
nal (clock) and operands are fetched almost every cycle from a cen-
tral memory, B) Data-driven model where variables are defined as local 
memories and data flows through these elements with control tags for 
steering the flow and C) A distributed realisation of the data-driven 
model where every variable is defined as a separate process which can 
communicate through channels with bounded storage (FIFO) on every 
link. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27

1.4 A dataflow description in Balsa (a) and its corresponding graph (b) vs. 
its RTL counterpart in Verilog (c). . . . . . . . . . . . . . . . . . . . 28
2.1 Three different implementation style of a system with four subsystems.
   (a) a fully synchronous control-driven design where communications are
governed by a FSM. (b) represents the same design in a fine-grained elastic
data-driven style where data and control information propagate
  together and (c) a synchronous dataflow realisation where coarser elastic
grains (shaded areas) are established through statically scheduling the
circuit to overcome the elasticity overhead of the control layer. 44

3.1 The Kahn Process Networks model vs. The Communicating Sequential
   Processes Model 52
3.2 The Teak synthesis system 53
3.3 Dataflow synthesis example: a) Balsa description of an iterative func-
tion; b) macromodule realisation generated using primitive components (Buffer,
Join, Fork, Merge, Steer and Computation/Operation) to manipulate
data. The shaded box on the left is a function in Balsa that is synthe-
sised into a F-C-J structure on the right. 56
3.4 A data-driven memory on the left implemented using push channels and
   a control-driven memory on the right which allows the computation to
   use pull channels for fetching data. 57
3.5 A latch-based SELF controller capable of instructing a pair of latches
   independently in the datapath. 60
3.6 Charlie diagram of an elastic pipeline scaled vs. its number of tokens 62
3.7 The behaviour of an elastic pipeline is depicted such that a stop signal
   (occurred between stage A and B at cycle 0) can delay the token flow
   by at least one clock cycle. 63
3.8 A regular pipeline extended by insertion of join and fork components.
The clock tree is replaced with a layer of Elastic Blocks (EBs). The
   EBs are able to communicate through channels according to the SELF
   protocol explained earlier. 64
3.9 A Petri-net model of a system on the left and a clustered version of it
   on the right partitioned based on the communication and computation
costs. 66
4.1 eTeak synthesis flow as a GALS extension to the Teak system.

4.2 eTeak Software Architecture with a back annotation loop (shown in red) which allows the Teak networks to be refined iteratively. This feature is employed to apply optimisations in further chapters.

4.3 eTeak SELF-adopted parametrisable component set

4.4 eTeak Elastic Join component behaviour

4.5 SELF-adapted macromodule example realising a sequential/parallel specification in Balsa.

4.6 Top structure is an asynchronous loop (ring) needs at least three buffers/latches to ensure deadlock-freedom. Middle structure follows a similar protocol. It shows that a storage element (Variable) in the loop can act as a buffer, therefore only two extra buffers are required to ensure correctness. Bottom structure is a SELF-adapted loop needs only one Elastic Block to work properly which could be a single Variable.

4.7 The Backward-interlock interference problem is shown when more than one join component exists in a combinatorial path which causes starvation when the successor Join’s stop becomes active and propagates backwards. This signal must get captured by a sequential element (an EB) otherwise it would cause unnecessary stalls in the token flow.

4.8 Using Eager Fork instead of regular Fork can resolve the combinatorial F-J problem in the fine-grained eTeak circuits.

4.9 The shown path is a False-Path which exists in the design and commercial synchronous Static Timing Analyser (STA) includes it in the critical-path report. To avoid this problem STA needs to be aware of similar paths in advance.

4.10 SELF adapted environment of eTeak

4.11 The sparkler shifter unit written in Balsa a) Case statement on Shift value for data manipulation in Shift-body procedure b) shows the main body of the Shifter procedure written recursively.
4.12 the structure on the left is a two-stage asynchronous dataflow shifter generated using Teak. To ensure deadlock-freedom a three-stage buffer is inserted into the loop. On the right a synchronous elastic version of the shifter is illustrated which needs only one buffer per loop. 83

4.13 The asynchronous QDI implementation of SSEM vs. its synchronous elastic counterpart in terms of cell area. 84

4.14 The results in first pair of columns belong to a fully buffered SSEM. The second pair belong to a SSEM with high-effort area optimisation which achieves $4.5 \times$ area improvement. 86

5.1 The communication states associated with each link in a synchronous elastic dataflow processor is monitored. 89

5.2 a) A graph representing a random Teak dataflow Network b) The grey area is a random cut consisting of an arbitrary number of handshake components c) The selected part is transformed into a SSM with additional buffering at its input and output arcs to resolve plausible deadlock hazards in the network. 90

5.3 eTeak Dataflow Network to RTL Transformation algorithm. 91

5.4 Three different patterns of macromodules in the Teak Dataflow Network realised by the Fork/Join and Steer/Merge primitives: Case A) concurrency, Case B) dependency and Case C) arbitration. 93

5.5 Sparkler shifter graph with two stages of data manipulation which is simplified to depict the control and the dataflow separately. 96

5.6 Verilog SSM for the shifter example with 56 states. 98

5.7 The graph on the left shows a concurrent dataflow implementation of a computation block and the graph on the right shows the SSM transformation of the same block with 354 states which is re-synthesisable through commercial EDA. 99

5.8 Patient SSM of the shifter module with buffers at input and output ports inserted back into the ALU dataflow. 101
6.1 Sources of uncertainty in a typical synchronous processor: a) IO inter-fac- ing with the environment and peripherals, b) bursty memory reads/writes, c) inter-core computation through buses and task queues for application usage, d) cache delays due to data misses and f,e) intra-core data dependencies such as speculation. ............................ 105

6.2 The difference between the conventional high-level synthesis flow (a) and the proposed flow (b) is shown. The De-elastisation flow exploits the asynchrony persistent at software domain to reduce the complexity of SoC design. ................................. 108

6.3 Type 1: Join & Fork in a macromodule guard the pull channels shown by dotted arrows. Fork splits request & Join gathers data within 0≤t≤n; n denotes a bounded number of clock cycles as latency. Non-determinism has an internal source and is posed by a memory block. .......................... 109

6.4 Type 2: Elastic Join & Fork in a macromodule guards push channels (A) & (B). Join can receive data on either within 0≤t≤n; n denotes a bounded number of clock cycles as latency. Non-determinism has external source and is posed by the input channels. ............................... 110

6.5 The energy and performance impact of Decomposition and De-elastisation on a eTeak-generated dataflow circuit: a) shows the dataflow realisation of the C1 ; [(C2 ||C3) ? C4] expression as a single cycle, b) is a decomposed implementation of (a) into two shorter cycles and c) is the de-elastisised version of a cycle from (b) where extra elastic blocks are inserted to balance the pipelines and remove the Join/Merge components. The critical path associated with each cycle is shown with red dots. ................................................................. 111

6.6 First phase: assigns depth value to Joins and Merges in the network. 113

6.7 Second phase: balance branches by deploying buffers. ....................... 114

6.8 Asynchronous, Synchronous Elastic and De-Elastic implementations of SSEM Processor compared in terms of area and performance. ........... 116

6.9 Asynchronous, Synchronous Elastic and De-Elastic implementations of Sparkler Processor compared in terms of area and performance. ........ 117
6.10 Asynchronous, Synchronous Elastic and De-Elastic implementations of Bitonic Sorter compared in terms of area and performance. 118
6.11 SELF adapted dataflow network of a 16-input Bitonic sorter 119
6.12 eTeak generated dataflow network of a 16-input Bitonic sorter 120
6.13 The De-elastisation flow: a system-level perspective. 123
6.14 Elasticity at different levels of granularity: a) fully elastic macromodule at component level; b) stage-level De-elastisation; c) loop-level De-elastisation (preserves elasticity only at boundaries); d) step-persistent bundling of concurrent computations (steps are depicted as shaded boxes) 124
6.15 Teak-generated 16-stage Spatial Sorter dataflow with 130 Joins and Forks which all removable by De-elastisation. In this case data-dependent performance is obtained. 128
6.16 SELF adapted dataflow network of a Spatial sorter 129
6.17 Probe loop and the 16-iteration LFSR-based function implemented as producer/consumer model using the Balsa language 130
6.18 Evaluation of the DBMS algorithms considering different computational models in terms of Area, Power, Throughput, Lines of Code (as a measure for productivity) and Compile/Synthesis time. 132
6.19 Averaged criterion of Area, Power, Throughput, Lines of Code (as a measure for productivity) and Compile/Synthesis time for every individual EDA flow. 133

7.1 E-Chart: Elasticity at different levels of granularity and abstraction. 140
The forward-looking design trend in Very Large Scale Integrated (VLSI) is Systems-on-Chip (SoC). SoC aims to integrate multiple computation, communication and storage components into a single chip and targets high performance systems by elimination of most off-chip communication costs. It is agreed that running SoC components under control of a single clock is not feasible and clock distribution has been revealed as a critical obstacle. Asynchronous techniques can be exploited to relax strict timing constraints of traditional design methodologies. A less radical solution is Globally Asynchronous Locally Synchronous (GALS) systems which offer potential advantages in this respect, as it preserves system modularity and concentrates on communication aspects. The problem with GALS design is the relative lack of familiarity of traditional designers with this approach. To deal with this, a methodology is proposed to allow designers implement GALS systems at a higher abstraction level which is independent of technology, protocol, data encoding or any other details of circuit design.

With the recent advances in concurrent programming, Communicating Sequential Processes (CSP) has gained popularity again. The CSP-based programming languages, like Go, have emerged to allow software designers to exploit the model toward implementing scalable softwares. CSP has a long history since 90’s in the hardware domain, mainly utilised by the Asynchronous community. In this thesis, a novel high level synthesis framework is proposed, called eTeak, which enables the designers to implement GALS-like systems in a CSP-based language (Balsa) without concerning about the timing issues at system level.

The proposed approach in this thesis takes advantage of synchronous elasticity to introduce a common timing discipline to the circuit which transforms it into a latency-insensitive system. A latency-insensitive system is able to tolerate dynamic changes in the computation and communication delays. This feature enables eTeak to raise the level of abstraction to the data-flow representation where functionality is separated from timing details. Therefore, it is possible for a designer to specify a large scale system by only concentrating on its functionality and postpone timing complexity to when synthesis takes place.

Unlike many previous systems, the proposed design flow employs data-driven synthesis style to distribute controllers through the network which contributes to its modularity and enhanced concurrency. This facilitates partitioning into elastic blocks and is supposed to pave the road for further optimisations, such retiming and re-synthesis, using commercial EDA tools.
Declaration

No portion of the work referred to in the thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.
Copyright Statement

i. The author of this thesis (including any appendices and/or schedules to this thesis) owns certain copyright or related rights in it (the “Copyright”) and s/he has given The University of Manchester certain rights to use such Copyright, including for administrative purposes.

ii. Copies of this thesis, either in full or in extracts and whether in hard or electronic copy, may be made only in accordance with the Copyright, Designs and Patents Act 1988 (as amended) and regulations issued under it or, where appropriate, in accordance with licensing agreements which the University has from time to time. This page must form part of any such copies made.

iii. The ownership of certain Copyright, patents, designs, trade marks and other intellectual property (the “Intellectual Property”) and any reproductions of copyright works in the thesis, for example graphs and tables (“Reproductions”), which may be described in this thesis, may not be owned by the author and may be owned by third parties. Such Intellectual Property and Reproductions cannot and must not be made available for use without the prior written permission of the owner(s) of the relevant Intellectual Property and/or Reproductions.

iv. Further information on the conditions under which disclosure, publication and commercialisation of this thesis, the Copyright and any Intellectual Property and/or Reproductions described in it may take place is available in the University IP Policy (see http://documents.manchester.ac.uk/DocuInfo.aspx?DocID=487), in any relevant Thesis restriction declarations deposited in the University Library, The University Library’s regulations (see http://www.manchester.ac.uk/library/aboutus/regulations) and in The University’s Policy on Presentation of Theses.
Acknowledgements

First and foremost I would like to acknowledge EPSRC and the University of Manchester for providing a full three year scholarship for this work under research grant GAELS EP/I038306/1. During the course of my PhD studies I was a collaborator on the GAELS project. The early retirement of the PI for this project (my prospective supervisor, Doug Edwards) just before the start of my PhD allowed me to pursue the development of my research ideas somewhat more independently than is perhaps usual. I believe this thesis would not have been possible without the support of my supervisor Jim Garside over the last three years. I am thankful for the opportunities he provided and for all I learnt from him.

During these years, I have had several discussions with people that have helped me to better understand electronic design automation (EDA) in general and asynchronous circuits in particular. My gratitude to Will Toms, Andrew Bardsley, Danil Sokolov, Jordi Cortadella, Luis Plana, Alex Yakovlev, Steve Furber, Geoffery Ndu, Vasilis Pavlidis, Javier Navaridas and also to the EDA community in Europe. I am also thankful to Siamak Mohammadi from for his team’s valuable comments on eTeak.

I am indebted to John Mawer, Abbas Kiasari, Alan Stokes, Steve Temple, Viv Woods and Dirk Koch for creating a great environment at the APT group. They have enriched these years with hours of discussions in topics as diverse as latex and programming tips, research ideas, sports, politics. And Alex Rast for providing us with top-notch chocolate from around the world! I must also thank Andrew Leeming and Guangda Zhang for scheduling badminton and football sessions on weekdays. These all made the final writing of this thesis an exciting meanwhile tolerable period. Thanks also to the everyone who I have met in the APT group. I am sure I will forget many names if I try to list them. Finally, thanks to all my family and friends for all the love, patience, encouragement and always being there for me when I needed most.
To my parents, my sister and my brother for their true support and encouragement.
Chapter 1

Introduction

The ever increasing complexity of System-on-Chips (SoCs) calls for new design technologies to enhance designer productivity. It is becoming necessary to abstract hardware design at a higher level, describing dataflows rather than Register Transfer Level (RTL) interactions. Much like a software description, these high-level flows neglect implementation details such as clocks and are, naturally, asynchronous and elastic, i.e. there can be an indeterminate number of data in a given place at any instant. This abstraction allows the designer to focus on specifying the system functionality and postpone issues of timing to subsequent stages of the synthesis flow. Consequently, synthesis from a specification in an algorithmic language is much more productive than a low-level realisation of a complex system using Hardware Description Languages (HDLs) [Wak04]. It is potentially beneficial in the context of newly emerged large-scale technologies, such as the Internet-of-Things (IoT), where a global infrastructure is described in a high-level, unified language and then realised as customised computations interacting across different networks and domains in a distributed fashion.

According to the UK housing energy facts [PC13] energy consumption associated with Information and Communication Technology (ICT) has grown by a factor of three since 1970 and is proportionately the fastest growing sector (compared to lighting, heating, etc). In the ICT sector, it has been shown that 37% of power for computation is consumed by the general purpose processors and 17% by the memories [FWB07]. These numbers are critical in the context of mobile devices where the power budget is limited. Extreme heterogeneity is believed to act as a cure for the power issue in
CHAPTER 1. INTRODUCTION

the domain of the general purpose processors through coprocessor-dominant architectures where CPUs are augmented with more efficient dedicated hardware blocks or application specific architectures [Int12].

In recent years, industry has been successful in adopting heterogeneity at the coarse level (e.g. ARM's big.LITTLE architecture) that enables efficient utilisation of the energy budget [Gre11]. Follow-up research has shown that energy efficiency can be further improved through fine-grained intra-core explorations [LPD+12]. The rapid growth of demand for high-performance, energy-efficient devices calls for advanced design and synthesis technologies. High-Level Synthesis (HLS) addresses this demand and offers a flexible, productive and customisable design methodology for the realisation of complex systems. It is claimed that raising the abstraction level enables designers to have almost $10 \times$ higher chance to improve power closure by architectural exploration [ZCDC15]. Rising the level of abstraction to algorithms is not restricted to the synthesis area; researchers in the field of neuroscience have also shown that modifying the algorithms, or in other terms, altering the 'way of thinking' for implementing ideas can provide, at least, $2 \times$ better resource utilisation in the large-scale systems [MKSF15].

The clockless nature at higher abstraction level (aka C level) is potentially convenient in that another problem with large SoCs is that synchronous clock distribution is difficult and, as different units may want to process at different rates, not always appropriate. Moreover, regarding the physical barriers in SoCs, variability in terms of power and clock speed beside gate/interconnect delays mismatch imposes significant impact on the efficiency of the large scale circuits.

An approach necessarily neglecting clock cycles has been taken in the asynchronous community. Asynchronous circuits, in contrast to synchronous systems, are not synchronised based on a clock signal. Instead, the components interact through local handshaking to perform synchronisation. Asynchronous systems typically comprise a number of semi-independent processes, communicating by passing messages or tokens. This gives them an elastic nature where zero or more tokens can be buffered between elements at any time and makes them more robust to variable latencies in communication and computation. Moreover, elasticity allows the separation of timing and functionality at design time, therefore raising the level of abstraction to the
untimed description level where clocking issues are not reflected to the designer thus contributing to productivity.

This thesis consists of two major components: HLS technology and elasticity. In the past decade elasticity has been explored mainly at micro-architecture level. By raising the level of abstraction it is possible to apply it at different levels of granularity. The explorations in this work provide an insight into elasticity in the design space and its employability at the system level for efficient heterogeneous synthesis. This work also pursues an answer to the question of why industry is reluctant to adopt elasticity at a finer level of granularity.

1.1 High-level Synthesis

Synthesis, as is well defined by Gajski, is the process of converting the given behaviour into a structure on each abstraction level [GK83]. With the rapid advance of semiconductor technologies the synthesis process in the recent decade has been forced to move to higher levels of abstraction which, significantly, enables the designers to tackle the complexity of SoCs and enhances productivity. Regarding the different levels of abstraction Gajski-Kuhn chart (or Y-Chart – Fig.1.1) has been developed as a comprehensive illustration platform to show the relations between different methodologies and tools in the design and synthesis context [GK83].

A Y-Chart distinguishes the three main different design approaches at various levels of abstraction represented by four nested circles: System, Processor, Logic and circuit; whilst design models are differentiated by three axes representing: Behavioural (or functional), Structural (or netlist) and Physical (or layout) models. It should be noted that each level of abstraction requires its own database of components with different levels of granularity such that the system level designs comprise processors, memories, buses and other processing components while at the processor level special hardware units such as memory controllers, arbiters, bridges, routers and interface components supply the custom or application-specific processors.
1.2 Elasticity

In the area of logic synthesis significant effort has been put into optimising control logic where cycle-accurate behaviour, posed by the designer, is considered to meet global timing requirements. Therefore, for large systems with many timing details, modifications to this behaviour can damage the functionality. Elasticity [CMSV01] allows the separation of functionality from timing. This methodology ensures that the composition of modules in a latency-insensitive framework will result in a correct behaviour. The elastic nature of these systems offers a common timing discipline to overcome global restrictions and enables systems to be tolerant against changes in timing of communication and computation.

Asynchronous circuits are considered as elastic systems. These circuits, in contrast to synchronous systems, are not synchronised based on a clock signal. Instead, the components interact with each other through handshaking to perform synchronisation. The semantic for this is defined in the form of two entities as sender and receiver and handshake signals between them which are typically called request and acknowledge. Regarding the differences between asynchrony and synchrony, fine grained concurrency is the potential advantage of asynchronous circuits. Besides that, low power consumption and modularity are other considerable characteristics that can result from flexibility of timing constraints [SF01].
1.2. ELASTICITY

Figure 1.2: Synchronous vs. Elastic Protocols: Three different realisation of a 3-input Adder where a) follows a rigid clocked synchronous protocol, b) adopts an asynchronous handshake protocol to enable timing variation tolerance and c) exploits synchrony and asynchrony by using the synchronous elastic protocol which discretises the timing variations, yet resilient against them.

In synchronous design, timing constraints must be met in the worst-case conditions to ensure the correct functionality. This strategy determines the clock frequency. Therefore timing closure affects the system performance. However, in asynchronous design, handshake based communication and computation relax timing constraints, thus the circuit speed adapts to the conditions that may arise.

The major disadvantage of asynchronous design is that the EDA tools in this area are not mature compared to synchronous design tools. Since designers are more experienced in designing synchronous circuits, they prefer to use synchronous EDA tools even if these tools do not provide potential features of asynchronous circuits. Furthermore, performance analysis for asynchronous systems is difficult because their behaviour cannot be determined simply by counting cycles of a fixed clock. Consequently, test and verification is challenging in these circuits. Finally, area overhead is another obstacle for using a pure asynchronous methodology due to its full interlocked handshake behaviour between its components, resulting in greater cost in silicon area.

To adapt the asynchronous design style and leverage its potential benefits, elasticity may be discretised so that it can be implemented within a synchronous system whilst preserving some benefits of asynchrony. Handshake signals are then synchronised with the clock and corresponding controllers implement a simpler synchronous protocol instead of a full-interlocked asynchronous one.

Figure 1.2 (a) shows an example of a conventional synchronous circuit performing addition. The circuit receives new inputs and computes the corresponding outputs on each cycle. The main assumption here is that all operations take only one cycle
and the rest of the system relies on it. The circuit is re-timed through buffers insertion (blue) to ensure the input elements arrive at the adder simultaneously. Elasticity relaxes this behaviour in a way that no cycle accuracy is required. Non deterministic delays are acceptable as long as they do not violate the order of data tokens and the sequence of valid data should be preserved. Figure 1.2 (b) represents an elastic adder where operands can arrive at the computation block with some bounded delay; and Figure 1.2 (c) shows a synchronous elastic version of the same circuit, in which the empty registers represent non-valid data or bubbles. As in synchronous systems, time is discretised, but if data items are not available during one clock cycle, the successor module can wait for a bounded number of clock cycles for the item to arrive. Unlike synchronous systems, for correct functionality insertion of extra buffers is not necessary in elastic systems. It can be observed that the sequences of valid data items for each signal are the same as in the non-elastic circuit. The implementation details of elasticity are discussed in depth in Chapter 3.

1.2.1 How Elasticity Contributes

Elasticity and its applications have emerged for network-on-chips (NoCs) [MD13] at system level [IM02] as it offers a flexible environment for heterogeneous communication and computation. Elasticity is able to deal with any sort of bounded non-determinism including cache delays [HB08], speculation in processors [GOCK09], IO interfaces, etc. Another source of non-determinism appears when data-dependent loops exist in computation – clearly a source of timing uncertainty in a dataflow.

In the asynchronous community several flows and languages such as Balsa [EB02], Haste/TiDE (formerly Tangram) [KP01][vBKR+91] and CAST/CHP [MN04] have been proposed in the past. These tools exploit elasticity to raise the level of abstraction to the description level. Although they have significantly contributed to the designer’s productivity in designing low-power processors [GFTW09][KKB+00], their performance has suffered from architectural drawbacks – typically from the need for handshake communications when transferring data at a fine-grained level rather than relying on evaluating within a prescribed cycle time – and area overheads due to the provision of the extra handshake circuitry.
Meanwhile, in the synchronous domain Bluespec SystemVerilog (BSV) [Nik08] has emerged as a high-level flow which abstracts the synchronous design flow more than languages such as Verilog and VHDL. BSV leverages elasticity through FIFOs at a coarse level, at user-defined boundaries, to handle communication between modules/rules/processes; this falls in the lines of Latency-Tolerant models [Kub97]. Unlike the asynchronous specifications, BSV relies on modern Electronic Design Automation (EDA) tools to deliver efficient results. Given that synchronous circuits are efficient, well understood and well supported by EDA, it is unlikely that this paradigm will be supplanted for most applications in the foreseeable future. Therefore, it is beneficial to take advantage of both synchronous EDA and elasticity towards an efficient system-level design flow.

1.2.2 What Degree of Elasticity is Enough?

Whilst elasticity gives the circuit timing flexibility, it comes with a cost. Communication must be synchronised by a handshake mechanism, imposing both an area and performance penalty compared to a synchronous model where assumptions about data readiness are statically engineered. In practice this overhead can overwhelm any advantages and not all the elasticity is useful.

The GALS design approach [Cha84] offers a solution to reducing the elasticity overhead, where synchronous ‘islands’ of logic run without continuous synchronisation between them and elasticity is preserved at the boundaries over the communication channels. The main problems of GALS design are the lack of methodology and tool support for efficient design space exploration. Classic methods are somewhat ad-hoc; only rudimentary design automation has been proposed [HMK+99] where the top-level hierarchy determines the boundaries of the synchronous islands.

The need for a framework to explore elasticity at different levels of granularity and abstraction is inevitable. This can be a methodology that allows the designer to explore partitioning of the system into synchronous islands. Recent studies demonstrate that using only the physical issues, or even structural factors (e.g. connectivity) as criteria for partitioning, instead of the functional aspect, i.e. the system’s data communication structure, may result in a sub-optimal design [DY07]. It is therefore beneficial to have an initial system with relatively finer granularity which can be used as building blocks
to define new communication/computation boundaries for system partitioning taking both functional and physical aspects of the design into account.

1.3 State of the Art

Three decades ago Chapiro introduced the GALS concept to the community [Cha84]. Since then academia and industry have attempted to exploit this concept in heterogeneous design which can potentially benefit from its multi-clocked behaviour. Broadly speaking, efforts to exploit the GALS methodology fall in the lines of ad-hoc design where synchronous blocks are glued together using interfacing logic and synchronisers [TGL07]. The work in the literature can be categorised into the following groups: a) optimising the interfaces for GALS systems, b) developing reliable protocols for communication and c) implementing efficient data-exchange patterns. The implementation of stable communication between clocked islands has been addressed with a handful of techniques including pausable clocks, asynchronous and loosely synchronous interfacing, etc. These techniques address the synchronisation and metastability issues [DGS] through explicit FIFO insertion which needs accurate considerations of timing details at circuit level and is survivable only by using an assemble-and-verify technique.

The idea of latency-insensitive systems was proposed by Carloni et al. [CMSV01] where a system-level control mechanism is used, based on the handshake protocol governed by relay stations, to realise coarse-grained elasticity and build latency tolerant designs. Their contribution is recognised as early work towards automatic GALS realisation. Although the idea was formalised in favour of automation, the inefficacy of relay stations restricted a flexible, fine-grained exploration of the design space.

Later, De-synchronisation [CKLS06][CCKT09] emerged as a technique to introduce elasticity into the synchronous domain by considering elastic controllers and join/fork components for every storage element and data conjunction in the design, respectively. Its fine-grained nature permits a set of flexible micro-architectural explorations including retiming and recycling, memory element bypassing and redundant storage removal to accomplish automatic logic synthesis [GOGCK11][CGOKS15]. However,
since De-synchronisation starts off from the register transfer level (RTL) these transformations are required to preserve the cycle-accurate behaviour of the system as the designer may be relying on it. Preserving cycle accurate behaviour imposed by the synchronous controller constraints the type of applicable optimisations.

As an example, it is not possible to restructure a pipeline using the mentioned techniques without considering modifications to the global timing requirements. For large systems, with many timing details, these modifications are not acceptable unless the system is following a common timing discipline at a higher level of abstraction. It has been demonstrated that cycle-accurate information at RTL (Figure 1.3 (A)) prevents a system being modelled using dataflows as the connections between datapath and control registers are not always explicit. It influences the partitioning task and makes it difficult to identify the boundaries in the circuit [SG13][SGE14] where elasticity has to be considered.

Recent contributions [SY14][SMS08] in this area have agreed that for efficient
GALS implementation, the task of partitioning or identifying appropriate clock boundaries is an extremely challenging task as it demands an efficient modelling scheme to take synthesis and optimisation into account rather than relying only on trivial techniques to establish a set of clocked domains. These contributions require laborious work and ad-hoc implementation of the system plus exhaustive simulation runs to extract the timing patterns which might influence either performance or power as the employed GALS techniques are considered at a coarse level [IM02][FKG+12]. Raising the level of abstraction offers stronger modelling of the system in a producer-consumer context (aka an ‘actor model’ in the software domain) to analyse the data exchange patterns between components. This method is applicable in dataflow models where the distributed nature of the controller allows effective partitioning (Figure 1.3 (B and C)).

GALS synthesis from a high-level perspective [SY14] provides a finer insight into the behaviour of the system which contributes to a more accurate exploration of the design space. Moreover, raising the level of abstraction to the description level enables elasticity to be considered selectively at algorithmic level where architectural decisions
may exert profound impact on the results at circuit level in terms of area, performance and power. High-level models such as dataflows are highly beneficial as they enable specifying hardware in the form of concurrent dataflows rather than thinking in a sequential way and squeezing the tasks into time boundaries. This work exploits the distributed nature of the dataflow model to apply regional transformations towards GALS synthesis [JMGTE14][JMG14].

1.4 Aim of this Research

The aim of this work is to free the hardware designer from concerns of clocking and cycle-by-cycle behaviour and allow freer design space exploration. At the same time the resulting circuit can be synthesised into synchronous blocks using existing, optimising CAD tools. The elasticity provided at the high level is selectively removed to give smaller, faster circuits at the ‘detailed’ end whilst retaining useful flexibility at the architectural level. This establishes a high-level synthesis perspective for GALS design. The proposed approach exploits the dataflow concurrency inherent in the asynchronous design and aims to raise abstraction from RTL to algorithmic level where system functionality is specified by dataflows, free from timing constraints.

Raising the designer’s abstraction level has some major benefits:

1. The designer is able to specify the hardware in the form of concurrent dataflows rather than thinking in a sequential manner and squeezing the tasks in time boundaries (Figure 1.4). This enhances the designer’s productivity in terms of development effort.

2. It provides synchronous designers with an interface to cover their unfamiliarity with asynchronous techniques, protocols or data-encoding in circuit implementation. Moreover, since ‘clock’ is automatically introduced to the circuit, designer can freely describe the functionality regardless of the timing details.

3. A higher level abstraction allows flexible exploration of the design space based on formal models, such as Communicating Sequential Processes [HR78], Petri Nets [Pet62] or Dataflow Process Network [LP+95] where it is possible to consider various analyses and measurements.
1.5 Contribution of this Research

This thesis revolves around eTeak, a synchronous elastic synthesis backend for the Balsa language which has been implemented solely by the author. The eTeak framework is exploited throughout this thesis to develop the ideas of re-synthesis and De-elastisation toward automatic GALS synthesis in Chapter 5 and 6, respectively.

The core synthesis process is borrowed from Teak, an asynchronous synthesis flow which is capable of transforming high-level descriptions into the circuits consisting of macro-modules [SOC67] with go and done handshake signals.

Initially the Synchronous Elastic Protocol (SELF) [CKG06]) is incorporated into Teak to introduce a common timing discipline to the system preserving its latency insensitive feature. Adopting SELF enables exploiting mature synchronous EDA for logic synthesis whilst preserving the fine-grained concurrency and the asynchronous properties discussed in Chapter 4. The proposed optimisation techniques falls in the lines of the powerful synthesis transformations in the synchronous domain: retiming and re-synthesis [JB06].

In the following chapter, a transformation mechanism is derived to partially convert eTeak [JMTG13] dataflow networks into Synchronous Sequential State Machines (SSMs) which reduces the level of elasticity by partially removing the inter-component communications. The transformed blocks are still linked by data flows with elastic connections where a variable number of data tokens may be queued on the communications channels. The resulting network is reminiscent of macro-modules although they may be clocked internally.

A heuristic is proposed in Chapter 6 that automatically partitions the network based on its high-level behaviour. In this regard, De-elastisation is introduced as a method to partially substitute the elastic protocol in a dataflow circuit with rigid clocked timing considering the architectural restrictions. classifying systems based on their behaviour and providing the designer with information about a ‘proper’ design style influences the partitioning process at higher abstraction level. This approach transforms the dataflow macro-modules selectively to low-overhead structures where fine-grained elasticity is removed whilst preserving the concurrent nature in place.

Replacing the asynchronous circuits at a low level conveys some demonstrable
gains: in terms of size 4-4.5× area saving is achievable whilst performance shows significant improvement of 3-4× in iterative and producer-consumer architectures. Against synchronous circuits the performance improvement is around 2× by leveraging circuit-level time borrowing with a moderate area overhead.

Regarding power, since the proposed approach relies on a clocked protocol it has 12-30% more power usage compared to its asynchronous counterpart. However, its data-dependent clock gating scheme makes it around 15% more power efficient compared to its synchronous implementation.

1.6 Thesis Organisation

The overall structure of this thesis is as follows:

Chapter 2 overviews the work in literature on high-level automation technologies and logic synthesis towards system-level design of GALS approach.

Chapter 3 reviews the background subjects in this thesis including high-level synthesis, elastic protocols and the synchronous counterpart.

Chapter 4 proposes the first contribution, eTeak, which is the SELF-adapted version of the Teak synthesis system.

Chapter 5 presents the second contribution which is a technique to partially transform the eTeak dataflow networks into Verilog descriptions to enable re-synthesis through conventional EDA flows.

Chapter 6 proposes a partitioning and optimisation technique, called De-elastisation, to selectively transform components from the elastic domain into the synchronous domain to lower the overhead of elasticity by exploiting the GALS design methodology.

Chapter 7 summarises this thesis in form of a complementary diagram to Y-Chart, derives conclusions from the three major contributions of this work and finally proposes open directions for further research.

1.7 Peer-reviewed Publications

The published papers associated with this thesis are as follows:

Alegre, Brazil. May 2016


Chapter 2

Related Work

This chapter reviews the related research over the two past decades in the area of automatic asynchronous logic synthesis, high-level approaches in the synchronous domain and recent contributions towards GALS synthesis. First, the state of the art in the asynchronous domain is summarised from two broad perspectives: data-driven and control-driven synthesis approaches. Moreover, the works that have extended asynchrony into the synchronous domain to be able to leverage commercial EDAs are reviewed. Later, a set of popular synchronous High-Level Synthesis (HLS) flows are discussed in depth from a system perspective. Finally, the recent contributions in the area of system-level synthesis and design towards GALSification are presented.

2.1 High Level Synthesis Flows

High-level Synthesis is the process of automatic generation of hardware circuits from ‘behavioural descriptions’ which is also referred to as a transformation from ‘behaviour to structure’. HLS was developed by CAD researchers to deal with growing complexity in today’s designs by automating the design process of the conventional manual design methodologies. HLS techniques have been around for almost three decades. Due to the potential advantages of asynchronous circuits, such as shorter cycles, on-demand computation and fine-grained pipelines [LVTS13][JE12a], over their synchronous counterparts asynchronous HLS tools have gained interest in recent years.

Along with rapid development of HLS tools for synchronous systems, asynchronous
designers have proposed a set of automated approaches, namely *syntax-directed compilation* and *logic synthesis*, to allow productive construction of large scale asynchronous circuits [SF01]. The syntax-directed fashion has emerged as a transparent compilation method which directly maps the language constructs onto hardware structures (aka handshake circuits). This paradigm allows construction of large scale circuits by the composition of small handshake components that are straightforward to implement in isolation. On the other hand, logic synthesis transforms a high-level specification into an intermediate level where control and data-path are separated and optimised individually by exploiting mathematical formalisms such as Petri nets [Pet77][Pet62].

In the next section asynchronous HLS systems for the synthesis of handshake circuits are broadly grouped into two categories: a) a control-driven model where operations are aligned based on a global enable signal and data/operands are fetched through ‘pull channels’ from a central memory and b) a data-driven model where variables are defined as local memories; data moves through these elements using ‘push channels’ with control tags enclosed in every data token to steer the flow. The following sections look in depth at the characteristics of these methods and the associated advantages and drawbacks in the context of the synthesis tools.

### 2.1.1 High Level Control-Driven Synthesis

**The Balsa Synthesis System**

In the late 90’s, Balsa [EB02] emerged as a mature compilation system which enables the designers to specify circuits at a higher abstraction level in a CSP-like language, named Balsa. The approach behind the Balsa framework is a *syntax-directed* translation of communicating processes into separate control and data circuits of handshake components. Balsa enjoys transparency: changes at source level may result in predictable restructuring in circuit implementation which simplifies the development process. Furthermore, this approach enables optimisations and design trade-offs to be explored easily at source level which is in contrast with a VHDL description where a small change in the specification may result in non functioning structures.

The Balsa framework, with the support of a set of features including parameterisation and a capability to implement recursive definitions, emerged to compete with
its industrial counterpart Tangram \cite{vBKR91}, produced at Philips Research Laboratories. Similarly, Tangram uses a CSP-based description language with a syntax more similar to traditional programming languages than CSP. Tangram has been used as a complete synthesis framework to compile extremely power-efficient asynchronous circuits, such as 8051 micro-controllers and smart-card chips \cite{KP01}. The Tangram synthesis system evolved into TiDE, and the newer version of the Tangram language is now known as Haste. Balsa promotes the use of more coarse grain parameterisable handshake set of components which differs from Tangram’s synthesis approach.

**Tackling the Control Overhead of Balsa**

The control-driven model suffers from major drawbacks imposed by the heavy nature of the central control unit. There are several works in the literature that attempt to mitigate the overhead. Plana et al. \cite{PET07} proposed a set of description-level techniques based on data-driven coding style along with some concurrent handshake components, including eager false variable, active input control and “concur” elements, to overcome the performance bottlenecks. The new set of components exploits parallelism and low overhead sequencing behaviour to achieve better performance. Later, their contribution to Balsa has been proposed as a data-driven framework which exploits push channels only for communication, computation and storage \cite{TEPTD10}. The data-driven model is discussed in Section 2.1.3. To demonstrate the impact of the proposed techniques, an asynchronous processor for smart card applications (SPA) \cite{PRB02} was synthesised and, on average, a 70% improvement in performance over the baseline approach was achieved. The accomplished results motivated several research groups around the world to focus their studies on data-driven computing models.

Another alternative is to re-synthesise the control unit and datapath of the circuits generated by Balsa separately \cite{NSM09,PTE05}. Essentially, this approach applies clustering heuristics on the control tree to enhance the performance. For this purpose, control synthesis tools such as Minimalist \cite{FNT99} and Petrify \cite{CKK97} are used to exploit the burst-mode technique proposed by Nowick et al. \cite{ND95} to re-implement the asynchronous controllers of Balsa whilst preserving the general behaviour of the circuit. The burst-mode method permits several input signals to change at the same time. Moreover, removing redundant handshake signals between data-path/control
and simplifying the communication between clusters of components in the control tree are techniques that have contributed to performance improvement of control-driven circuits as well [TN01].

Generally speaking, control re-synthesis is a limited approach for improving the speed of the control tree. Although clustering helps mitigate the control overhead, synchronisation between control and data maintains the sequential behaviour of operations which still is a main source of performance deficiency in such systems.

In addition to the control re-synthesis methods, a set of automated ‘source to source’ supervised transformations are proposed [HS08] which uses a variety of concurrency enhancing optimisations including automatic pipelining, arithmetic optimisation, automatic parallelisation, reordering of channel communication and loop unrolling. These transformations may not preserve the specified order of communications on channels, so a deadlock could arise. To avoid this, the designer needs to ensure that the target transformation is safe to apply. For this purpose the flow of data across channels should be determined early by the designer in form of a directed graph. Although this approach achieves significant performance improvement, it requires extra design effort from the designer which may reduce its usefulness.

A similar work at the Technical University of Denmark [NSJN09] has also attempted to develop a Haste-to-Haste synthesis front-end to employ source level optimisations. The presented tool, after analysing the extracted data-control flow graph, generates an optimised control and data-path in Haste considering the required constraints. The synthesis process also solves classic problems of scheduling and binding using a discrete time model as in a synchronous design. Since the output is a program in Haste, it still needs to be synthesised with TiDE in a syntax-directed fashion which results in poor speed. Although a set of low-level peephole optimisations are introduced at post-synthesis level to alleviate the control overhead of TiDE, the performance bottlenecks remain dominant.

In this context, there are some other works that concentrate on scheduling, bonding and resource sharing at higher abstraction levels to improve area, power or performance factors [HS10][AL11]. In general, scheduling algorithms are discussed in two different categories. The first uses the Integer Linear Programming (ILP) formulation which is not capable of handling large scale problems due to the well-known problem of state
explosion. The second is based on heuristics such as simulated annealing. Despite fast problem solving using heuristics, optimality is not guaranteed when dealing with large scale systems. To tackle these issues, researchers usually consider hybrid techniques which exploit mathematical formulations and heuristics together as an optimisation framework.

In this regard, Beerel et al. proposed a scheduling method based on Linear Programming (LP) \cite{BLDK06}. This work uses Petri Nets for modelling the problem. Then, to avoid state explosion, the number of states is reduced into a smaller set while preserving the scheduling characteristics. The relaxed timing characteristics of the asynchronous systems raise less restriction on the scheduling time intervals. This feature is exploited toward design space exploration. Results indicate that the proposed method with a reduced number of states is able to achieve a feasible solution.

To tackle the state explosion problem, Nielsen et al. \cite{NSM09} have proposed a behavioural synthesis flow for generation of asynchronous circuits which exploits conventional EDA tools for resource sharing, scheduling and bonding. As a back-end it leverages the syntax-directed compilation scheme of the Balsa synthesis system to allow generation of large scale asynchronous circuits. The behavioural synthesis scheme in this flow helps the designer to express the desired behaviour at algorithmic level in systemC or HDL, based on the input constraints, the EDA tool produces the appropriate input for Balsa.

This work allows EDA tools to perform optimisations over a control-data flow graph, extracted from the behavioural description of a circuit. Thereafter, Balsa code from the optimised graph is generated. This step considers some relaxations on discrete scheduling of the control unit. Finally, it uses the Balsa system to generate handshake components. The advantage of this approach is that the designer has full control over the resulting circuit. But the disadvantage is Balsa’s control overhead, as discussed earlier such as unnecessary synchronisation between control and data, is imposed to the final circuit.

2.1.2 Towards The Synchronous Behaviour

In the synchronous context, Peeters et al. attempted to introduce a synchronous version of handshake components to enable the Tangram system generate synchronous
circuits and, hence, implementable circuits on FPGA [PvB01]. In this way, the designer is provided with a test and verification framework to explore the design which also enhances design productivity. The proposed approach excessively converts each gates into sequential elements such as C elements in the control path to flip-flops (FF).

This work emerged as the first contribution to involve clock gating at higher levels of abstraction for the synthesis of synchronous systems. For this purpose, components are manipulated to gate the clock signal in the absence of data to save energy. However, clock gating only affects the registers of the data-path and keeping the FFs within the control unit untouched could not help much with power consumption. Due to the ‘heavy’ nature of the control part in these circuits, the energy consumption of the synchronous version is reported as 4 times greater than their single-rail asynchronous counterpart.

Furthermore, although a clock signal is introduced to the system, the full interlocked behaviour of the asynchronous protocol is still preserved between the stages which reduces the performance of the system. In contrast, Jacobson et al. [JKB+02] proposed a different protocol which exploits the cycle-accurate behaviour of synchronous circuits to relax the full interlocked signalling of asynchronous circuits. The main objective in this work is to leverage the interlocked behaviour of asynchronous pipelines in conventional synchronous systems to build Elastic Synchronous Pipelines (ESP). This approach applies fine-grained control over the flow of data and consequently is able to manage power consumption more effectively.

Two major benefits of asynchrony when applied to the synchronous pipelines are:

- The first benefit is the stage-level locality of signals in asynchronous interlocked pipelines. This avoids the global distribution of stall signals in synchronous systems. In fact, unit level or coarse-grained clock gating needs a global stalling mechanism which could affect data causality; therefore signal locality must be considered carefully in synchronous pipelines. Since stall signals are no longer distributed globally in such circuits, they would not influence the stall signals that are within the critical path. On the contrary, an asynchronous interlocked pipeline employs a fine grained request-acknowledge signalling between stages which ensures correct propagation of data through the pipeline (backward interlocking).
• The second benefit is that, in interlocked pipelines, computation is requested only when data is available. Therefore, in the absence of data, corresponding registers are switched ‘off’. This on-demand computation strategy reduces power consumption extremely. Asynchronous pipelines can naturally benefit from such a data driven style due to fine-grained distribution of ‘valid’ signals (so called forward interlocking). This mechanism not only reduces power consumption, but also avoids global distribution of valid signals which addresses the signal locality problem in synchronous systems.

The interlocked protocol consisting forward interlocking (Valids) and backward interlocking (Stalls) is presented in the context of 2-phase clocked pipelines. This work is the first known latch-based implementation of synchronous elasticity. Corresponding circuit-level controllers are presented following the detailed function of such pipelines.

A conceptually similar work – synchronous elasticity – emerged to leverage the synchronous interlocking protocol towards automatic implementation of elasticity. In this respect, a simplified communication protocol and corresponding circuit level control scheme proposed by Cortadella et al. [CKG06] which introduces a generic synthesis mechanism based on the synchronous elastic protocol. Additionally, a synthesis flow is also proposed which inherits the concept of de-synchronisation [CKLS06]. The synthesis process starts off from a synchronous RTL architecture and assigns elastic controllers to each register to enable them govern the flow of data in a flexible fashion. In this way, micro-architectural exploration of the design becomes possible. In this regard, a set of correct-by-construction transformations are proposed [GOGCK11] to consider restructuring the system towards pipelining and parallelism to improve the performance. Moreover, a work by Krstic et al. presents a formal framework to prove the correct functionality of this protocol and then elaborate it towards composability of elastic networks [KCKO06].

The idea is identical to the concept of the Latency-Insensitive (LI) systems [CMSV01] which define a formal framework to present ‘patient’ processes. The formalisation is done at the higher abstraction level and the latency tolerant behaviour is verified in terms of token (valid data)/bubble (non-valid data) which attributes to more complexity compared to the formalism introduced for SELF protocol (see definition in 3.2.3). In contrast, synchronous elasticity is formalised in terms of handshake signals which
make it easier to verify and implement. Prior to the theory Carloni et al. [CMSSV99] proposed a correct-by-construction methodology which uses the semantics of ‘Shells’ to provide control over modules to convert them to stoppable processes. Shells are defined as wrappers which can communicate through handshake signals and control the synchronous modules at a coarse level of granularity. Compared to SELF, the elastisation happens at system level over modules which limits the exploration space.

From a different perspective, there is a set of contributions that bring synchrony to a higher level of abstraction which, in contrary to above mentioned works reflect timing requirements to the designer. In late 80’s and early 90’s several languages such as LUSTRE [HCRP91] was emerged to allow designers to realise parallel machines by exploiting dataflow models of computation. A more recent work is Bluespec System Verilog (BSV) which has been successfully adapted in Industrial context and is discussed in Section 2.2.

2.1.3 High Level Data-Driven Synthesis

The overhead of the control-driven model imposes a prohibitive impact on performance. This motivates data-driven models to emerge as a solution. In this section a review of the contributions in data-driven synthesis is given.

Dataflow computer architecture was first introduced by Jack Dennis as an alternative to Von Neumann’s model of computing. Later, practical computers were designed based on the dataflow model at MIT and the University of Manchester, which were named Tagged-token Dataflow Machine and Manchester Dataflow Computer [GKW85], respectively. Due to the programming complexity of these machines they were never brought to the market. However, several programming languages were developed to help programmers. Despite the problems these machines encountered in the past, their parallel, distributed and demand-driven nature remains a major driving force for researchers in the area of computer architecture; Particularly, the idea of hardware rewriting systems by Arvind [H+00] which has led to the emergence of the Bluespec System Verilog (BSV) language and evolution of synthesis technologies for low-power circuit design by the asynchronous community:

Martin et al. [MM11] have proposed Communicating Hardware Processes (CHP) based on the CSP model along with a complete synthesis package, called CAST [MN04],
for fine-grained distributed asynchronous design. To address the performance issues, a Data-Driven Decomposition (DDD) mechanism is presented [WM03]. In this work, by preserving the language semantics, the tool tries to map system-level processes onto fine-grained structures of customised transistor-level pipelines. Since this approach relies on pipelining sequential programs it may not preserve the system behaviour compared to syntax-directed approaches such as Balsa. It is claimed by Taylor et al. [TEPTD10] that applying a DDD-type optimisation to a processor could break the memory interfaces and cause hardware malfunction. Instead they proposed an alternative approach which exploits a new compilation method to produce more ‘pipeline-like’ systems. The proposed dataflow compilation technique maps language semantics onto handshake components. The handshake circuit style is used for three reasons:

- This approach generates structures at the component level so it does not rely on a particular implementation style or communication protocol. Therefore, the synthesis process becomes more flexible compared to other synthesis styles.

- It allows transparent compilation which helps a designer modify the final architecture directly at the source level.

- It provides the designer with the capability to integrate control-driven and data-driven circuit styles. Moreover, component reuse is possible and it lets the designer choose an appropriate style for each part of a design. However, interfacing push and pull channels needs particular considerations.

To leverage these advantages, a subset of Balsa is developed to provide the designer with specific syntax to allow easy implementation of data-driven structures [TEPTD10]. However sticking with more traditional languages such as Balsa or Haste avoids restrictions on the input language. This problem was one of the initial motivations for developing Teak [BTE09] which is discussed later. To preserve the transparency between the code and the generated circuits, optimisations in terms of scheduling, binding, or resource sharing are not considered. Therefore, the presented research is limited to the development of a data-driven compiler to generate asynchronous circuits based on a set of handshake components.

This work demonstrates the achieved results by comparing a complex 32-bit microprocessor – nanoSPA – to the existing control-driven style generated using the Balsa
synthesis system. This processor implements a slightly cut-down version of the ARM instruction set. The overall architecture follows a traditional Fetch-Decode-Execute pipeline with a register bank and memory interfacing. In terms of area and energy, the data-driven style shows 50% and 85% overhead respectively. These numbers are even worse when a more complicate encoding (dual-rail instead of single-rail) is used instead. The register bank, due to its variable-based structure, is determined to be an obstacle for improvements in terms of energy and area. For this reason, an alternate version using the Balsa system is synthesised and integrated onto nanoSPA which employs the data and control driven styles in a hybrid fashion [Dua10]. Although this approach slightly improves the power consumption, it ends up with lower performance.

A few years ago, the Philips research laboratory proposed a data-driven implementation for the Haste language. The work introduces a flip-flop (FF) based element, named ‘Click’ [PtBdWM10], as a template to be replaced with ‘C elements’ in asynchronous circuits. Therefore, it enables conventional EDA tools to be used for timing analysis and optimisations. Moreover, the Click templates are well suited for data-flow compilation, which avoids much of overhead of control-driven syntax-directed compilation. Click elements are FF based structures comprising control and data parts. Control parts communicate based on a 2-phase handshake protocol. Based on the handshake signalling, an internal clock (enable) signal is generated for the corresponding data and control FFs. Therefore, this mechanism provides the circuit with a clock gating mechanism similar to that of self-timed systems. Despite the benefits of clock gating for the circuit in terms of power consumption, some special timing considerations including justification of hold and setup time, are required to guarantee the correct functionality of FFs.

Regarding the compilation process, first some optimisations including latch insertion and optimal initial token deployment are applied on a control-data flow graph, and then it is translated directly onto data-driven handshake components. Thereafter, generated handshake components are written into a SystemVerilog-based format to allow synchronous EDA tools to perform systematic optimisations on combinational logics in the data-path. Regarding the asynchronous nature of the graph, latch insertion should be done carefully on each cycle to avoid deadlock.

In terms of throughput, results are much improved compared to the syntax-directed
2.2. POPULAR SYNCHRONOUS SYNTHESIS FLOWS

synthesis style of Haste. It should be noticed that circuits with 2-phase handshaking in this work are compared to circuits with 4-phase handshaking. In short, this work aims to improve the performance of the self-timed asynchronous circuits by moving to a data-driven synthesis style and removing the overhead of control-driven approach. In addition, it employs conventional EDA tools to perform optimisation on components within a datapath.

Figure 2.1 shows how the prior art in the control- and data-driven synthesis area differs. In the control-driven design (a) the flow of data in the datapath is governed by a central control unit whereas in the data-driven design (b) control propagates with data, therefore local synchronisations are necessary. It is also shown in (c) that by defining logic clusters (aka bundles or localities) it is possible to remove away the control overhead of synchronisers. Static scheduling of latency insensitive systems [MPP11] [CJCK11] have emerged to address the synchronisation complexity of elastic systems in the control and data-driven contexts. This concept is adopted in this thesis to pursue a general approach toward automatic GALS synthesis using high-level CSP models (see De-Elastisation - Chapter 6).

2.2 Popular Synchronous Synthesis Flows

In recent years, system-level design has been explored widely both in academia and industry. The major portion of the effort has been devoted to the compositional techniques [LPC12] which usually rely on user defined boundaries or cuts. These methods mainly start off exploring the design space with a set of Intellectual Properties (IPs) and are generally limited with the timing requirements of each IP. Raising the level of abstraction to higher levels provides the tool with another degree of freedom: where to define the boundaries. Moving the boundaries and exploring the design space demands an ecosystem with a flexible timing discipline which allows flexible architectural exploration without affecting functionality of the localities in the system. In this section some recent contributions in this area are reviewed and criticised.

Nowadays there are several approaches for SoC design, some of which are selected to evaluate De-elastisation in this chapter. HDL-Verilog is known as the most popular language in IC design that benefits from well-established libraries and EDA tools.
Figure 2.1: Three different implementation style of a system with four subsystems. (a) a fully synchronous control-driven design where communications are governed by a FSM. (b) represents the same design in a fine-grained elastic data-driven style where data and control information propagate together and (c) a synchronous dataflow realisation where coarser elastic grains (shaded areas) are established through statically scheduling the circuit to overcome the elasticity overhead of the control layer.
2.2. **POPULAR SYNCHRONOUS SYNTHESIS FLOWS**

However, the RTL abstraction of HDL-Verilog is fundamentally *timed* and forces designers to squeeze tasks into clock boundaries. The majority of HLS flows raise the abstraction to a timing-free level. As output, RTL Verilog is generated to reuse the state-of-the-art logic synthesis flow and established techniques for individual optimisation of data and control paths. The De-synchronisation [CKLS06] technique also largely relies on RTL models. It introduces fine-grained asynchrony to cope with the uncertainties in timing and power consumption (as opposed to the worst-case timing margin in synchronous design).

For the sake of productivity it is desirable to abstract hardware design at a higher level, describing dataflows rather than RTL interactions. Much like a software description, these high-level flows neglect implementation details such as timing and are, naturally, asynchronous and elastic. Below is an overview of the most popular HLS flows developed both in academia and industry:

**BlueSpec** [Nik08] is an atomic rule-based synthesis scheme capable of synthesising dataflow networks of rules. It uses an HDL-like Bluespec Verilog (BSV) language that exploits SystemVerilog as a front-end and adopts Haskell’s state-based computation model for the scheduling rules (these are embedded in higher-order functions that are guarded by conditions). This model is fundamentally parallel, in contrast to sequential programming models (such as C++) which leverage an extension to model parallelism. Rules in BSV are defined by the user and employ the all-at-once updating approach similar to Verilog when a clock executes the *always* blocks forcing the state-holding elements (registers) to update simultaneously. BSV uses *Atomic Transactional Memories* to handle communication between modules/rules/processes and follows the *Latency-Tolerant Shared-Memory* model [Kub97].

**Chisel** [BVR12] is a synthesisable-by-construction, graph-based synthesis scheme that allows meta-programming features (e.g. recursion and parameterisation) to be exploited for scalable hardware design. It uses an extended version of the Scala programming language for defining modules, wires and registers. Although a recent contribution proposes an elastic NoC fabric [FFDMS14] for large-scale multiprocessor chips and SoCs using Chisel, its coarse-grained nature prevents intra-process exploration of elasticity towards energy efficient design.

**LegUp** [CCA+11], similarly to other C-to-Verilog compilation paradigms, takes
C functions and transforms them into a mid-level representation using the LLVM compiler. It exploits advanced compilation features, such as loop unrolling, vectorising and parallel realisation. Thereafter, LLVM code of each function is translated into a Finite State Machine (FSM) that has access to the registers implementing global and local variables. The communication between the FSMs is through a bus that also provides shared access to a general purpose processor. This processor is employed to run hard-to-synthesise or non-synthesisable functions of the C code. LegUp leaves the tasks of decomposing the code into functions and considers parallelism using pthread for the designer. Unlike domain-specific languages and tools, a C-based synthesis flow provide a facility for software developers to reach hardware level implementation without involving timing. However, as C was not designed for IC synthesis, some functions can result in an inefficient implementation. To improve cycle time within pipelines a De-pipelining technique has been proposed [HCS+15]. This work follows similar principle between synchronous and asynchronous pipelines: over-buffering a synchronous circuit might have positive impact on clock period but it will have negative effects on throughput [JMGTE14].

For implementing multiple-clocked designs, the programmer needs to define and instantiate separate Clock types and set up the frequencies regardless of the physical constraints. A recent work in this domain proposes a new language to enable designers to specify a control unit in the context of SELF protocol. The work aims to provide a synthesis flow exploiting SELF protocol at higher abstraction level to address the variability problem. However, apart from the control unit, the designer should implement the datapath components in VHDL. This way, the control actions should tightly be connected with the datapath components which generate complexity for design space exploration. In other words, restructuring demands major modifications to the central control unit.

2.3 System-level GALS Design and Synthesis

GALS design techniques have been widely studied in the past [TGL07] [KGGV07]. There are several methodologies for GALS design which mainly fall in the lines of ad-hoc design where synchronous blocks are glued together using trivial interfacing logic
2.3. SYSTEM-LEVEL GALS DESIGN AND SYNTHESIS

and synchronisers. Teehan et al. have reviewed the advantages and drawbacks of a set of these methods including pausible clocks, asynchronous and loosely synchronous interfaces [TGL07]. Another study by Krstic et al. [KGGV07] has demonstrated through a handful of industrial applications, including cryptography hardware, a telecommunication application based on Network-on-Chip (NoC) [BDM02] and a radio transmitter, that FIFO based GALS systems have better throughput than pausible clocked systems and handshake based techniques. However, their associated area overhead can be addressed through the optimisation techniques such as buffer resizing [BJC08]. The promising results of the GALS NoC design methodology have motivated researchers to extend this paradigm towards the application level and consider Dynamic Voltage Frequency Scaling (DVFS) where dynamic characteristics of the application are taken into account [RSMF11] to lower the overall energy usage. From a different perspective, the scalability feature of GALS NoCs has enabled the designers to build low-energy large-scale computation infrastructures, such as SpiNNaker SoC [PFT+07]. The problem that all the above mentioned techniques share is the designer’s productivity (e.g. designing the SpiNNaker SoC took 30 Man-Months). To tackle this problem, high-level GALS synthesis methodologies has been developed by academia which highly relate to the contribution of this thesis: Register Transfer Level GALSification [SG13][SGE14] and Locality and Bundled Centric GALSification.

2.3.1 Register Transfer Level GALSification

The Asynchronous Verilog Synthesis (AVS) approach [SG13][SGE14] was developed to enable exploiting HDL-Verilog for system design at RTL whilst leveraging asynchrony to realise a GALS design automatically. AVS considers analysing the RTL code through dataflow extraction and aims to apply partitioning to the circuit by introducing asynchronous channels between synchronous localities. This method is fairly comparable to the de-synchronisation technique which introduces elasticity into the synchronous domain by considering elastic controllers and join/fork components for every storage element and data conjunction in the design, respectively.

The major drawback that AVS shares with de-synchronisation [CKLS06] is that since both start off from the register-transfer-level (RTL), the process must keep the
cycle-accurate behaviour of the system because designer may be relying on it. Preserving the cycle-accurate behaviour puts constraints on the type of the optimisations applicable.

As an example, it is not possible to GALSify a pipeline using the mentioned techniques without considering modifications to the global timing requirements. For large systems with many timing details these modifications are not acceptable unless the system is following a common timing discipline at a higher level of abstraction.

To confront this problem, AVS exploits dataflow modelling of the RTL specification to identify the connections between the clocked entities (or registers) in both the control and data-path. Describing a system at RTL needs the designer to implement the control and data-path separately. This not only reduces designer’s productivity, but also makes it hard and error prone to recognise the connections between control and data registers [SG13][SGE14]. AVS classifies the data holding patterns according to their control logic: registers, controllers, finite state machines and counters, to simplify the partitioning task.

In general, GALSification at RTL, despite its disadvantages, allows the designer to use the most popular language for hardware design, HDL-Verilog, which makes it possible to reuses a large library of IP for Verilog designers, e.g. on OpenCores.

### 2.3.2 Locality and Bundled Centric GALSification

High-level GALS synthesis demands strong models to make the process feasible with minimum limitations. Regarding the limitations of Marked Graphs in modelling conditional systems, Sokolov et al. [SY14] have employed Petri Nets to study the concept of GALS partitioning into localities based on the behavioural factors from a higher abstraction level. Their proposed approach discusses a set of high-level patterns for behavioural partitioning and the comparisons against structural partitioning is claimed to be addressed as future work. Burns et al. has extended this work toward GALS verification and synthesis using an in-house framework, Workcraft [BSY15]. Another Petri Net formalism in this area is proposed by Fernandes et al. [FKPK+13], as the theory of Step Persistency, which extracts a reachability graph of the model and prunes the signals to establish maximally concurrent ‘bundles’ so that they can be scheduled to the same clock tick. To adopt this technique in a synthesis flow, the impact of this
theory has to be evaluated in a realistic application.

2.4 Conclusion

The related work to this thesis has been reviewed in this chapter. Three different classes of contributions in the high-level synthesis context were discussed including: asynchronous, synchronous and globally asynchronous locally synchronous. First, an overview of the optimisation techniques in the asynchronous domain is presented. Next, a set of popular synthesis flows are detailed in the synchronous domain and finally, the recent works in the area of high-level synthesis of GALS systems are inspected.
Chapter 3

Background

This chapter discusses the two fundamental concepts upon which the contributions of this thesis are established: a) The asynchronous synthesis approaches of Balsa and Teak, b) The elastic protocols for communication and computation among which synchronous elasticity is introduced as a discretised elastic protocol. First the Teak system architecture is discussed and elaborated. Next, preliminary definitions of elastic systems are discussed and compared against synchronous protocols. Finally, the chapter briefly provides a review of the clustering scheme used to trade-off between synchrony and asynchrony.

3.1 The Teak Synthesis System

3.1.1 Input CSP-based Language: Balsa

As mentioned earlier Balsa is a channel-based language which relies on the Communicating Sequential Process (CSP) model of Tony Hoare [HR78]. The CSP model was mainly utilised to describe concurrency in industrial context. Its communication model has evolved significantly in recent years and has influenced several languages in both software and hardware domains including Go and Balsa, respectively. According to the Balsa language manual [EBJ+06] its descriptions are dividable into procedures with communication channels for message passing. Channels can be also defined for interaction between commands (actions). A command may include: concurrent or sequential commands, local procedures or read/writes from/to variables. For further
The powerful model of CSP is expressive and allows description of non-deterministic behaviour such as arbitration and conditional statements based on the values of tokens. This enables the designer to exploit CSP for modelling a vast range of applications, such as speculation and cache coherency, where non-deterministic and conditional behaviour are possible to be modelled. On the other hand, Kahn Process Networks (KPNs) and Synchronous Dataflow Models (SDF) have emerged to restrict the CSP model by making deterministic assumptions on communication which makes them straightforward for behavioural analyses.

The overlap between Kahn Process Networks (KPNs) and Communicational Sequential Processes (CSPs), shown in Fig. 3.1, is by introduction of synchrony to the non-deterministic domain of CSP such that a deterministic behaviour is achievable. KPNs are popular for describing deterministic applications, particularity in the data-streaming domain where the depth of the FIFOs are already known [LNZ14] such as in multimedia, imaging and signal processing applications [SZT+04]. Moreover, synchronous behaviour enables the tool to perform cycle-accurate analysis and optimisations. There are several works in literature [SMS08][BEZ15] that have considered KPN and Synchronous Dataflow (SDF) models for GALS analysis but a major drawback with this approach is the lack of enough formalism to model non-determinism in realistic applications.
3.1. THE TEAK SYNTHESIS SYSTEM

![Diagram of the Teak synthesis system]

Figure 3.2: The Teak synthesis system
3.1.2 Syntax-directed Synthesis of Balsa

Teak was proposed as a new compilation method and component set for data-driven implementation of Balsa descriptions. The compilation method is syntax-directed, the same as Balsa. Teak, compared to the work presented by Taylor et al., is more powerful in the sense that its input language is Balsa and the set of components is small which enables building simple networks. Regarding the data-driven style, Balsa descriptions are implemented based only on push channels which contributes to performance improvement compared to the control-driven style where control signals are unavoidable they are implemented in the same way as data channels but carry 0 bits of data.

As is shown in Figure 3.2, Teak consists of three main layers, including a parser which generates a parse tree, a network generator which generates a mid-level representation using handshake components and a netlist generator which produces a gate-level Verilog netlist using the provided technology files and component set. The optimisations including latch insertion, clustering and scheduling are applicable to the mid-level representation where control and data dependencies are available.

The Teak component set operating based on the 4-phase handshake protocol includes: (The SELF adapted set is discussed in the next chapter)

- **Steer** which works as a data driven de-multiplexer; Teak maps ‘case’ statement onto this component. Each parameterised output independently matches the conditions of input. With 0 bit data width, Steer works similarly to the Balsa case component.

- **Merge** this component works as a data-driven multiplexer. Its service, carrying input bits to output, is based on first come first served policy. Inputs should be mutually exclusive.

- **Join** is an unconditional parameterised synchroniser. It concatenates data bits of arriving inputs. A two-way join of n and 0 bits can be used as a conjunction of data and control tokens.

- **Fork** this component can be parameterised to carry any number of bits from input to output. It is frequently used for producing control tokens from n bit
data tokens to determine the completion of a process. A two-way Fork of \( n \) and \( 0 \) bits can be used to separate data and control tokens.

- **Buffer** data storage and channel decoupling. Buffers are used to separate pipeline tokens. They can be inserted on channels independently to achieve any desired degree of storage.

- **Variable** permanent storage. This component is a buffer with an extended control over its write and read activities. \( \text{write-go}(wg)/\text{write-done}(wd) \) and \( \text{read-go}(rg)/\text{read-done}(rd) \) pairs make all data initialisations and terminations possible. The number of read/write ports of a Variable is parameterisable and Teak is able to generate multiple read/write ports for each variable. In data-driven networks variables are local synchronisation points between data and control incurring the cost of always moving control along with data.

- **Operator/Computation** the only components that can manipulate data. Inputs are formed into a single word through a Join. Teak generated operators are adapted to the dual-rail handshake protocol [SF01] which results in larger robust circuits with inefficient performance. Further research in this area may include exploration of efficient architectures using commercial synchronous CAD tools.

### 3.1.3 Teak Architectural Properties

The following are the architectural properties of the Teak system. Later these features are exploited to develop further optimisation methods toward GALS design.

Features of Teak can be grouped into communication and computation facets. From the communication perspective Teak networks are synthesised in a syntax-directed compilation manner from a CSP-like language. The primitives of the language, including channels and processes, are preserved which form **point-to-point communication** between the computation blocks at hardware level which contributes to concurrent message passing.

- A **Slack Elastic** system can be pipelined with any degree of storage (e.g. arbitrary size FIFOs) on its communication channels. This behaviour was first formalised for the distributed computation systems which were described in a CSP-like
Figure 3.3: Dataflow synthesis example: a) Balsa description of an iterative function; b) macromodule realisation generated using primitive components (Buffer, Join, Fork, Merge, Steer and Computation/Operation) to manipulate data. The shaded box on the left is a function in Balsa that is synthesised into a F-C-J structure on the right.

language, CHP [MM98]. Slack Elasticity provides a flexible communication environment for the computational blocks in the system. We take advantage of this feature in Teak to optimise the processes without affecting the overall functionality of the system. Composition and decomposition of modules towards GALSification benefits from elastic communication which is not available in the synchronous domain where rigid timing controls the communications.

- The Macromodule logic was introduced to enable designers implement complex circuits using simple data processing building blocks [SOC67]. Later, this concept was used to simplify the asynchronous control design [CKK+02]. Teak employs this technique to perform the control interactions locally instead of having them as a separate central unit which has significant performance implications. We exploit the local (aka distributed) control behaviour to apply De-elastisation within macromodules which results in defining new boundaries in the network. Figure 3.3 shows a sample macromodule synthesised using Teak from a high-level specification in the Balsa language. The distributed nature of macromodules enables optimising one module without affecting the correct functioning of the other macros in the network.
3.1. THE TEAK SYNTHESIS SYSTEM

Point-to-Point (PTP) communication enables a module to have independent rates of data streaming from different sources which contributes to a higher level of concurrency and accordingly effective throughput. This model of communication has been exploited in different forms and follows the same model as the Actor model which is a popular concurrency model by Carl Hewitt from MIT introduced in the '70’s [Hew77]. This method allows development of parallel computations both in software and hardware domains. In an industrial context it has been leveraged for scaling computations [Kel10], meanwhile it has been employed in academia to implement low power computation platforms [FB09]. As an intuitive example lets assume that module A with the input set of \{a,b,c,d\} and the output set of \{x,y\} is capable of performing two functions f,g which are independent. The function f takes \{a,b\} as input and g takes \{c,d\}. if we assume that input values are supplied with different rates of a’, b’, c’ and d’ where a’ is the slowest rate then g can operate and produce output independent from a’ which results in higher throughout of module A. The PTP communication is closely compatible with the data-flow style of computation in Teak modules.
• The Dataflow architecture: Dataflow machines emerged as an alternative design style to reduce the centralised control effect and speed up the computation by prioritising the data [AC86]. In the Teak networks dataflow architecture joined with PTP communication realises concurrency and eases the modules decomposition process. Accordingly, modules are controlled locally through handshaking so whenever data becomes available computation can start [AC86]. Based on this concept, data-dependent computation is allowed by data streaming could exist within a module which can significantly influence the performance of the circuit. In addition, it allows the tool to perform functional decomposition over a module and define new boundaries.

• Quasi Delay Insensitivity: Delay Insensitive (DI) circuits can have unbounded delays on wires and elements. In DI every data transfer has to be acknowledged and no change is allowed before the acknowledge signal is withdrawn. Teak generates a class of DI circuits called quasi-DI where forks are isochronic [SF01]; This implies that the delays of the forking wires are identical, therefore an acknowledgement on a single end can safely indicate data arrival to a signal that is routed to several places.

• The Memory Architecture: The ‘push’ nature of the dataflow systems pose a significant design barrier when it comes to memories and storage elements. In conventional designs memory holds data tokens; processing units keep reading them when required through ‘pull’ channels. In dataflow architectures memories are handled in a different fashion such that after every read the associated memory location needs to be re-written to avoid data loss in the next cycle of operation which imposes area and performance drawbacks [TEPTD10]. Teak exploits a hybrid architecture to confront the drawbacks of the memories by considering pull channels for handling storage and push channels for dealing with computation (Figure 3.4).

3.1.4 Teak Visualisation Engine

Dataflow programming benefits from a visual environment to represent graphically how nodes connect in a dataflow-based application which provides the user with a better
understanding of the concurrent interactions in the application, providing the possibility of end-user programming. Teak enjoys a powerful visualisation engine capable of illustrating components at different levels of granularity, from handshake primitives to coarse, high-level processes interacting with each other through channels. This provides a debugging mechanism for the designer to enable monitoring concurrent operations in a large-scale dataflow system. What distinguishes Teak from other visual environment such as Quartz [Sch09] and LABVIEW is that it allows the designer to browse and check different levels of granularity while using the runtime simulation facility to chase the data token in a design. A more detailed description of using Teak for debugging synchronous elastic circuits is given in chapter 4.

In Teak networks datapath/control interactions are done through the local handshaking by Forking/Joining of control and data channels. This feature enables the structures to perform control interactions using separate ‘go’ and ‘done’ channels. The separation of go/done channels leads to fine grained concurrency and allows pipelining to be applied over the generated circuits. Moreover, the decoupling behaviour of Teak buffers introduces token storage over channels and paves the way for further investigation on circuit parallelism.

Teak extracts parallel entities from the high-level Balsa code, produces a Control-Data Flow Graph (CDFG) and then maps it onto the macromodules with local control handshaking through the go-done channels. Therefore the resulting circuit benefits from a distributed control scheme. This feature allows exploring different architectures by replacing the communication-heavy asynchronous designs with Finite State Machines (FSMs) which would trade off the elasticity and the concurrency level inherent in the asynchronous design. Chapter 5 looks in depth at this possibility.

The major difference between the Teak and Balsa synthesis systems is in the optimisation strategy. Balsa produces control-driven handshake circuits which, for the sake of efficiency, require further structural optimisations. This leads to extremely complex structures. In the Teak dataflow circuits optimisation is modular and can be achieved through buffer insertion. This significantly simplifies the optimisation process. The following chapters exploit this feature of dataflows toward high-level GALS synthesis.
3.2 Asynchronous vs. Synchronous Elasticity

Synchronous elasticity has emerged to exploit some of the advantages of asynchrony in the synchronous domain by transforming rigid, clocked systems to latency insensitive systems. In this way, the timing constraints of a synchronous circuit will be relaxed which provides the circuit with more flexibility in tolerating communication and computation delays. In this respect, SELF [CKG06] is proposed as an efficient communication protocol along with a component set. Accordingly, a possible synthesis flow is presented which starts off from synchronous RTL and applies elasticity at this level. Furthermore, a set of correct-by-construction transformations is introduced including buffer insertion, early evaluation, elastic retiming and recycling to enable a designer to explore micro-architectural possibilities towards performance improvement.

3.2.1 Elastic Blocks

Elastic Blocks (EBs) are the most important entities in elastic networks as they are responsible for proper transmission and storage of data tokens. In latency-insensitive design they are known as relay stations [CMSV01]. An EB resembles a flip-flop with some extra gates to implement the control mechanism based on the handshake protocol. As is shown in Figure 3.5 EB can be viewed as a pair of latches operating at opposite clock phases. In a normal mode latches operate as a flip-flop where one holds a bubble, the other holds data at any time. A bubble is a latch containing no data.
This state of the EB delivers maximum throughput as it always allows exchange on every clock cycle. This state may change, for example due to a stall, the stall indication can pass back up the pipeline, one stage at a time, with the ‘last’ bubble. This ensures that interaction is still local. In general, an EB is a FIFO, which has the capability of storing and transmitting tokens. A FIFO can have an arbitrary latency of token delivery from its input to its output. Initially an EB contains bubbles. Since bubbles contain no data, they can be inserted within any channel and yet the system will function equally well. Similarly, Carloni et al. have used empty relay stations to split long wires in any stage of the design flow. In the synchronous domain this method is known as retiming and is commonly used to break down long interconnects in a design into smaller ones.

3.2.2 Elastic Channels

A channel is a communication medium between modules. In synchronous systems only data is transferred via channels in a datapath. The assumption is that data always move on a clock cycle, thus space will become available for each data token. To handle pipeline stalling a global signal is required to be distributed throughout the network. This method suffers from the existing problems for global clock distribution. In asynchronous systems interaction is local, typically controlled by Req/Ack signals which indicate validity of data and compilation of an operation respectively. Pipeline stalling is handled locally. In case of stall Ack remains false and data will ‘bunch up’. Communication may not proceed until Ack becomes true. Elasticity is therefore implicit.

3.2.3 Elastic Protocols

Different states of an EB are formulated as a protocol, called SELF, which implements delay tolerant communications in an elastic network. The corresponding controller is shown in Figure 3.5. Unlike AMBA AXI protocol [Ltd11], which uses double flip-flops to realise a handshake protocol for SoC integration, SELF controllers are latch-based units and hence occupy less area. In contrast to the stoppable process definition, SELF emerges as simple signal-level handshaking between sender and receiver. As in
conventional asynchronous protocols, two wires – valid (V) and stop (S) – are used to implement different states of the protocol and the components should interact in terms of these states:

- (I) Idle, (-V): valid data is not provided by the sender.
- (T) Transfer, (V, -S): valid data is provided by the sender and the receiver is available to capture it.
- (R) Retry, (V, S): valid data is provided by the sender but the receiver is busy to accept it. Therefore, the sender should ‘retry’ in the next cycle.

3.2.4 Back-pressure

In an asynchronous pipeline data move when a token exchanges places with an empty space, a bubble. The problem with this is that, if the pipeline latches are all full, forward progress is impeded by backward bubble propagation. In this regard a principle states that for identical forward and backward latencies, it is necessary to keep buffers half full to ensure the maximum throughput [EFS98]. As demonstrated in Figure 3.6, the maximal throughput achievable is usually below the ideal point, due to the non-‘ideal’ implementation. In the left part the throughput is limited by the number of tokens, whereas in the right part it is limited by the lack of bubbles. A dotted curve is also shown in the diagram, corresponding to an elastic circuit with no data tokens or with no bubbles which may lead to starvation or deadlock, respectively.
Figure 3.7: The behaviour of an elastic pipeline is depicted such that a stop signal (occurred between stage A and B at cycle 0) can delay the token flow by at least one clock cycle.

This behaviour in a synchronous circuit emerges in a different form known as backpressure. The reason for this problem is that each decision is made on a clock boundary, so the quantisation of time is more severe, especially important when processing because bubble propagation is still limited to one stage/clock. Synchronous elastic systems address this issue by considering extra places in the context of relay stations in latency insensitive systems, one location is for data propagation in absence of backpressure, whilst the other place is for the on-the-fly data token which is sent before noticing the blocking stall indicator. As is shown in Figure 3.7 backpressure can cause throughput degradation due to unnecessary stalls in the pipeline: At cycle 0 Valid and Stop signals are both high between stage A and B; This forces the associated controller to remain in the Retry state (see section 3.2.3) until Stop is withdrawn.

### 3.2.5 Non-linear and Conditional Pipelines

The Fork/Join and Steer/Merge component pairs are used to expand the linear communication of EBs and generalise them to any number of input and output channels. Fork sends data into two or more independently controlled pipelines and Join synchronises data from two or more incoming pipelines. The same is true for Steer and Merge with more controllability such that the control information determines which channel
should be used for steering and merging data.

The control associated with Join asserts its output valid signal only when both inputs are valid. Otherwise, the incoming valid inputs are stopped. In elastic pipelines Join is the only component capable of producing a stop signal for predecessor stages; other components can only propagate stop signals as appropriate. The controller associated with Fork is rather simple; It waits for both receivers to be ready before sending the data.

It is possible to create elastic modules with an arbitrary number of input and output channels by chaining several Join or Fork structures. The example shown in Figure 3.8 illustrates a function block (C) wrapped with EBs. Fork and Join components are used to expand the linear pipeline.

The conditional nature of the Teak dataflows discourages this work from using Marked Graph (MG) or Petri Net (PN) models for measuring circuit-level criteria such as power, performance or area. To use MGs for modelling elastic circuits a probabilistic extension has to be considered and, similar to PNs, they would encounter the well-known problem of state explosion. In the following chapters designs are evaluated by simulating implementations to avoid the problems mentioned above.

### 3.2.6 Elastic Components

Elastic components of Join and Fork are analogous to Teak’s component set presented in section 3.1. These components are considered to expand the linear communication of
EBs and generalise them to any number of input and output channels. Fork sends data into two or more independently controlled pipelines and Join synchronises data from two or more incoming pipelines. The control associated with Join asserts its output valid signal only when both inputs are valid. Otherwise, the incoming valid inputs are stopped. In elastic pipelines Join is the only component capable of producing a stop signal for predecessor stages; other components can only propagate stop signals as appropriate. The controller associated with Fork is rather simple; it waits for both receivers to be ready before sending the data. It is possible to create elastic modules with an arbitrary number of input and output channels by chaining several Join or Fork structures.

### 3.2.7 Clustering Elasticity

A disadvantage of elasticity is the overhead in providing extra places (aka ghost latches) which, in many cases, remain empty. To avoid this, grouped blocks can be controlled and stalled locally in a conventional synchronous manner; this is possible because the stall signal can be distributed within a single cycle. This is less feasible on a large scale, so a large circuit can be broken down into ‘grains’, each with an elastic interface. Choosing the most appropriate breakdown of grains and where to insert elastic buffers is not simple and opens a huge design space. Modelling this by building alternatives would be prohibitively expensive. Thus abstract modelling of the system is required. This work exploits high-level synthesis to raise the level of abstraction which largely favours the analyses proposed in this research.

Figure 3.9 illustrates a symbolic view of a system modelled in Petri Net where some clustering technique is employed to create groups based on an optimisation criteria in terms of communication or computation costs.

Particularly, in this system each block has been assigned an EB controller to perform clock gating over the global clock and govern the flow of data within the system. Therefore, all the registers of the target block are enabled with only one signal supplied from the corresponding EB. In this way, when data are not available, an EB is able to freeze the internal state of the module and stop its computation. Moreover, to handle the back-pressure problem, ghost latches should be considered along with EB controllers to transform the blocks to stoppable processes. In this way, each module
Figure 3.9: A Petri-net model of a system on the left and a clustered version of it on the right partitioned based on the communication and computation costs.

has elasticity on its periphery but may be simplified internally. This may be more practical than elasticity at the ‘stage’ level.

3.2.8 Conclusion

This chapter reviews the key concepts a reader will need to take along to grasp ideas proposed throughout this work. First, the Teak synthesis system is elaborated in depth as a data-driven approach upon which the contribution of the future chapters are based. Then, Teak’s architectural properties are described and their corresponding advantages are reviewed toward GALS synthesis. Finally the synchronous elastic protocol (SELF) is explained with regard to its basic constituents. The next chapter proposes eTeak as a SELF adapted synthesis framework.
Chapter 4

eTeak: Synchronous Elastic Dataflow Synthesiser

As was discussed in the previous chapter, high-level synthesis in the asynchronous domain has moved towards the data-driven style to combat the overhead of the global control in the control-driven approach. The data-driven approach provides a distributed control mechanism which contributes to modularity, fine-grained concurrency and controllability over power consumption. Moreover, it allows on-demand computation to be realisable which is known as the better-than-worst-case design scheme which allows the design to exploit the average-case performance [JE12b].

In the synchronous domain, Carloni et al. [CMSV01] have proposed the concept of Latency Insensitivity (LI) to tolerate uncertainty in communication and computation timings at system-level. In this context, a conceptually similar work, synchronous elastic protocol (SELF) [CKG06], was proposed to provide an EDA-friendly approach to deal with these uncertainties. The protocol introduces the asynchronous handshake signals to a synchronous circuit and transforms it to an elastic counterpart which is synthesisable using commercial EDA flows. Unlike LI its fine-grained nature provides better controllability over power and performance.

The proposed framework in this chapter is named eTeak. It aims at raising the design abstraction to algorithmic level to provide the designer with flexibility in implementation of concurrent hardware regardless of timing. This approach conveys potential advantages in the context of GALS systems. In this respect, the advantages of the data-driven synthesis style and SELF are combined for two reasons: a) exploiting
commercial CAD flows towards automating the *GALSification* process, b) introducing a common timing discipline to the circuit to facilitate cycle-accurate analysis of concurrency.

In the light of the features mentioned for the asynchronous data-driven synthesis style, the Teak system is employed as a baseline synthesis flow to investigate the synchronous elastic protocol from a high-level synthesis perspective. By incorporating SELF in Teak fine grained study of elasticity becomes available. Moreover SELF could be applied into a vast range of architectures. In this thesis eTeak is considered as the baseline synthesis framework to explore automatic *GALSification*. The following section details the different layers of eTeak and briefly describes the contributions of the next chapters on eTeak.

### 4.1 eTeak Synthesis Flow at a Glance

This section details the main constituents of the eTeak flow shown in Fig. 4.1 where (a) corresponds to the main asynchronous flow of Teak which is capable of synthesising the
4.2. SYNCHRONOUS ELASTIC COMPONENTS OF ETEAK

timeless specifications in Balsa into a network of dataflows. Teak provides a technology mapping facility which is refined and adopted to the SELF protocol by incorporating a new library (b) into Teak which is elaborated in the next section. By adopting SELF the behaviour of the Teak macromodules changes both at register and gate level. The related advantages and challenges are discussed in sections 4.4 and 4.5, respectively. Steps (c) to (f) are explained in the following chapters: step (d) and subsequent pipeline shows a system-level optimisation that transforms Teak dataflow networks into Synchronous Sequential Machines (SSMs) through mature EDA re-synthesis; and (e) reduces the overhead of Teak networks by removing the inter-component communications and introducing synchronous rigidity to the system. Finally (f) is a feedback path that enables system-level explorations, possible through back-annotation of the generated hardware.

The principle architecture of the Teak system is depicted in Figure 4.2. The red coloured feedback link which connects the output Teak network file to another network file shows the path leveraged in this work for iterative optimisation and buffer insertion. Particularly, this feature enables the eTeak framework to consider both high-level patterns and physical constraints toward effective GALSification (see Chapter 6).

4.2 Synchronous Elastic Components of eTeak

To incorporate synchrony in Teak, its existing component library is adapted to the SELF protocol and buffers are converted to time decoupling controllers to govern the flow of control and data based on the elastic protocol. The component set is depicted in Figure 4.3. SELF uses latches instead of C-elements to enable synchronous EDA for re-synthesis. Section 4.5 looks at the potential issues SELF would cause in a data-driven context.

Adopting SELF preserves the functionality of Teak’s component set (see Section 3.1) but it modifies its communication behaviour. The new behaviour relaxes the full interlocking between components which results in a simplified implementation. In this section an example demonstrates the new behaviour of the eTeak components. In this regard, the Join component is selected due to its rather complex function: Join is the only active component that can pause the token flow, the rest are passive components
Figure 4.2: eTeak Software Architecture with a back annotation loop (shown in red) which allows the Teak networks to be refined iteratively. This feature is employed to apply optimisations in further chapters.
and are supposed to pass the signals over. In other words, Join synchronises the data pipelines and it is redundant wherever synchronisation is not necessary.

Figure 4.4 shows two different plausible scenarios taking place for a 2-input Join component in two sequences \{0a, 1a, 2a\} and \{0b, 1b, 2b\}. \( \delta \) denotes the associated combinatorial delays with the Join component. The first scenario happens when one input becomes Valid but the other is not. The requesting channel must therefore be paused for one or more cycles until the other arrives. At cycle 0 (0a), Valid goes high on one of the channels indicating data availability. After \( \delta \) delay it gets a Stop since a Valid has not appeared on the other channel yet. Note that Join is combinatorial and \( \delta \) is a continuous value. At least one cycle later (1a), a Valid appears on the other channel which allows Join to ‘turn off’ the Stop signal and produce a Valid for the successor component. At this point, if the successor component does not return a Stop before next cycle, data propagates and all of the corresponding signals go inactive after \( \delta \) delay (2a).

The second scenario takes place when the successor component returns a Stop which may prevent the system from progressing for an unlimited number of cycles. After \( \delta \) delay Stop propagates backwards and moves the channel states from ‘Transfer’ to ‘Retry’ (0b) – For the states of the protocol see the previous chapter. The channels remain in the ‘Retry’ state until the successor component removes Stop. Here we assume that Stop is withdrawn in the next cycle. Consequently Join turns off Stops on the both input channels and, after \( \delta \) delay, the channels move to the ‘Transfer’ state (1b). Finally, in the next cycle data propagates and all the corresponding signals
This example shows how the asynchronous rendezvous is mapped to the synchronous library regardless of latencies that may happen at any point. The presented scenarios delay the communication by (at least) one cycle which may cause performance deficiency. Chapter 6 proposes a method to deal with this deficiency at system level.

4.3 SELF-Adapted Macromodules of eTeak

Macromodules, as were shown in the previous chapter, are the building blocks of the Teak dataflow networks. Figure 4.5-(a) depicts a Teak synthesised macromodule consisting of combinational elastic components including Join (J), Fork (F), Merge (M), Steer (S) and sequential Elastic Blocks (EBs) to implement SELF. The block is synthesised from a high-level procedure in the Balsa language: \([(C.1; C.2) || C]\). The generated structure in this example encapsulates the C1, C2 and C computations in a Merge-Join-Fork independent iterative structure able to communicate with the environment through channels. The following scenario describes how computation and communication is performed in the SELF adapted macromodules of Teak: in (b) a data token
arrives at the Join component but it receives a stop since no token is available on the other branch yet, (c) a control token ‘Go’ appears on the other branch as an activator from a predecessor macromodule. Join allows the data token to proceed and sends a valid to the successor EB, (d) the data token propagates on the next clock cycle and the successor EB captures it and sends a valid to the next EB, (e) in this cycle a new data token arrives at the Join but it is not allowed to propagate since a computation is still in progress. It should wait until a ‘Done’ token appears at the Join indicating that the computation is processed which is the case for the final scenario (f). The longer the inner pipeline, the more is the waiting time for the second data token to get processed. Therefore, it is important to have macromodules already decomposed into shorter cycles for performance gains.

4.4 Advantages

By incorporating SELF in the Teak flow, the CSP-based networks of Teak are transformed to synchronous dataflow machines whose properties are potentially beneficial for hardware modelling and synthesis as their behaviour is cycle-accurate, and hence comparatively deterministic. The following are the advantages these machines have over their fully asynchronous counterparts.
4.4. ADVANTAGES

- **Simplified loop structures regarding correctness**: In an asynchronous loop each cycle must always have enough buffering for a lead token to move forward and leave space for the following token plus there must be at least one ‘space’ to separate the tokens. Consequently, at least three buffers/latches are required in a cycle to ensure correctness. In a SELF adapted loop one master-slave register is enough as all tokens’ movements are synchronised with clock. Fig. 4.6 depicts a simple loop structure in the form of a macromodule to realise iterative operations. According to the results reported in Section 4.8 this leads to almost $10 \times$ less cell area usage in a realistic example.

- **Extended correctness by construction**: To ensure correct system behaviour the SELF protocol is formalised and is shown that insertion of elastic buffers does not change the behaviour of the system [CKG06] and the order of data tokens is preserved. However, the insertion of buffers is constraint to empty buffers (aka bubbles). The slack elastic nature of the $\epsilon$Teak dataflow networks not only preserves the order of data tokens but also relaxes the bubble constraint and allows the designer to insert data buffers without need to re-design the system (see Section 4.8).

- **Reduced dynamic power due to lower switching activity**: The SELF protocol uses forward-interlocking to transfer data. The backward path goes active occasionally when data gets blocked. Therefore, full-interlocking for each transfer using Request and Acknowledge (aka handshake signals) is not required which contributes to lower switching activity between the sender and receiver stages. Moreover, the time-decoupling controllers in SELF employ clock-gating (see Section 3.2.1) which reduces the switching activity in general.

- **Datapath improvement using synchronous EDA library**: Data manipulation components generated by Teak use functional decomposition technique [GAFN15] and QDI encoding for computation [Tom06] necessary for communication as the environment uses the same encoding. By adapting SELF, exploiting synchrony in the computation units becomes possible as the communication fabric follows the same timing behaviour. Within a synchronous framework it is possible to use ‘conventional’ datapath computation elements, which are considerably simpler.
Figure 4.6: Top structure is an asynchronous loop (ring) needs at least three buffers/latches to ensure deadlock-freedom. Middle structure follows a similar protocol. It shows that a storage element (Variable) in the loop can act as a buffer, therefore only two extra buffers are required to ensure correctness. Bottom structure is a SELF-adapted loop needs only one Elastic Block to work properly which could be a single Variable.
Figure 4.7: The Backward-interlock interference problem is shown when more than one join component exists in a combinatorial path which causes starvation when the successor Join’s stop becomes active and propagates backwards. This signal must get captured by a sequential element (an EB) otherwise it would cause unnecessary stalls in the token flow.

than QDI units. This concept triggers the idea of partial re-synthesis proposed in the next chapter.

• **Cell area reduction:** The Teak back-end generates 1-of-2 4-phase QDI circuits with no need for timing assumptions. This class of circuits occupy almost four times more area than bundled-data circuits [CKK+02] which assume bounded delay for signal transfer. SELF employs bundled-data encoding for the clocked circuits which removes the need for considering additional timing assumptions. Therefore, our SELF adapted dataflows are four times smaller than the equivalent Teak circuits.

4.5 Challenges

There are several issues with adapting the SELF protocol in the Teak flow. The following summarises the synthesis challenges a synchronous designer would encounter:

• SELF only signals ‘Stop’ in response to a ‘Valid’ request. A Join component which needs to ‘Stop’ on input signals back to a Fork to prevent its ‘Valid’ reaching the other branch and retain synchronisation. However this removes the request to the Join as well, removing the Stop, and so on. With the standard
components this sets up a ring oscillator if a single Valid reaches the Join. This problem is named the Fork-Join interaction (F-J). It becomes critical in the dataflow context where there are several join and fork components that take care of the dependencies in the network so the probability of F-J occurrence is quite high. Moreover, Backward-interlock interference occurs when more than one Join component becomes active along a combinational path as shown in Fig. 4.7.

There are several ways to resolve these issues:

- As is suggested by Carmona et al. [CCKT09] to use an Eager Fork instead of a regular Fork to avoid this problem as shown in Figure 4.8. However, the associated area, and consequently static power consumption, needs to be taken into account when elasticity is applied at fine-grained level.

- Buffer the links that cause this problem so that direct interaction between Fork and Join is prevented.

- Remove Joins and Forks from the design. This approach is explained in detail as a product of Chapter 5 and 6.
• Merge/Steer components are combinatorial entities in the Teak dataflow networks. These are the only decision making components which rely on the control token propagating with the data. The problem arises when there is a race between the data and control values as they may arrive at the component from different sources. The asynchronous implementation avoids this problem by employing a full-interlocked protocol which guarantees a Delay-Insensitive (DI) behaviour in both control and data path, but the synchronous elastic implementation has a simpler handshake protocol and this issue would arise unless the tokens are synchronised. A cure for this issue is to make sure that data/control tokens are synchronised by a sequential element before they arrive at these components.

• Memory Management: since Teak relies on a hybrid model (Push and Pull channels) for handling the memories or storage elements, SELF adoption has to be applied with careful consideration. For loading or storing data from/to a memory block, a set of signals including data, enable and read/write has to be set or reset simultaneously. Insertion of EBs for deadlock avoidance or optimisation would violate this rule and hence, cause inefficiency in data fetch or store. However, this will never damage the functionality as the interfaces are elastic and
any bounded delay in data transfer is tolerated.

- Regarding test and verification, it needs to be noticed when feeding in the new tokens that the valid signal needs to be held for a clock cycle and when a stop signal is received it should be handled based on the SELF protocol. To make sure that the protocol is followed completely it is recommended to buffer all input/output ports so that the interaction of the design with the environment is guaranteed to be based on the SELF protocol.

- Commercial synchronous tools will identify timing paths even when they cannot be active. This can cause the presented critical path to be longer than the true critical path, as illustrated in Figure 4.9. By False-Path specification in the data-driven networks the tool is prevented from reporting unnecessary critical paths.
4.6 Using eTeak for Debugging Purposes

To monitor the concurrent operations in a parallel dataflow design one can use eTeak as a powerful visualisation engine to spot the deadlocks and starvations in a design. This paradigm is extremely helpful when implementing multiple-clocked systems where the rates change from one domain to another, which complicates chasing the tokens. Figure 4.10 shows the SELF-adapted environment of eTeak where the top-right window lists the modules of the target design. The main window visualises the modules in form of dataflows where links are shown in colours based on their widths. By browsing the time window, on the bottom-right, it is possible for the designer to set the clock frequency for simulation and debugging purposes. By running simulation the flow of tokens are illustrated on every link with red and blue colouring which represent Valid and Stop states of SELF, respectively. Links are ‘off’ when there is no data being transferred via the link which represents the Idle state.

4.7 Motivating Example

Figure 4.11 shows a Balsa procedure, named ‘shifter’, with three input and one output ports. The top level procedure consists two local procedures: shift_body, part (a) in the figure, and shift_n which has an input/output port. This allows cascading the procedures and constructing a pipeline. The shifter procedure receives ‘distance and shift’ values through channels and passes it over to the chain of shifter_n procedures which are instantiated in part (b) in a recursive fashion. Each shifter_body procedure receives data (i) through a channel and manipulates it based on the control values of ‘distance and shift’. In this particular example the ‘distance’ parameter is set to two to generate a pair of shifting stages.

The Teak and eTeak generated dataflow of the shifter example are illustrated in Figure 4.12. Both are consist of two pipeline stages which are wrapped in a single feedback loop (aka algorithmic loop). According to Section 4.4, to ensure deadlock freedom, the asynchronous version of this 32-bit shifter (on the left) needs a 3-stage latch whilst the synchronous version running at 550 MHz (on the right) needs only one elastic buffer. It should be noticed that to break the Fork-Join loop, as discussed in the previous section, buffers are considered per loop.
procedure Shifter(
    input shift : ShiftOp;
    input distancel : 5 bits;
    output result : Word;
    input arg : Word
) is
begin
    loop
distancel, shift -> then
    local
        procedure shift_n (  
            parameter distanceBit : cardinal;
            parameter distance : cardinal;
            input i : Word;
            output o : Word
        ) is
            local
                constant remaining = 32 - distance
            function PackWordLeft (lsw : distance bits; msw : remaining bits) = (#lsw @ #msw as Word)
            function PackWordRight (lsw : remaining bits; msw : distance bits) = (#lsw @ #msw as Word)
        channel c : Word
        procedure shift_body (  
            output o : Word
        ) is
            begin
                l -> then
                local
                    function i_lswLeft = (#i[remaining-1:0] as remaining bits)
                    function i_mswRight = (#i[31:distance] as remaining bits)
                    begin
                        If #distance[distanceBit] then
                            case shift of
                                {left, ?} then o <- PackWordLeft (0, i_lswLeft [])
                                {right, 0} then o <- PackWordRight (i_mswRight [], 0)
                                {right, 1} then o <- PackWordRight (i_mswRight [], 0, -1 as distance bits))
                            end
                        else o <- i
                            end
                        end
                        if distance > 1 then
                            shift_n (distanceBit - 1, distance / 2, c, o) || shift_body (c)
                        else
                            shift_body (o)
                        end
                    end
                end
            end
        end
    end
end

Figure 4.11: The sparkler shifter unit written in Balsa a) Case statement on Shift value for data manipulation in Shift-body procedure b) shows the main body of the Shifter procedure written recursively
Figure 4.12: the structure on the left is a two-stage asynchronous dataflow shifter generated using Teak. To ensure deadlock-freedom a three-stage buffer is inserted into the loop. On the right a synchronous elastic version of the shifter is illustrated which needs only one buffer per loop.
The overall latency of the synchronous version is 7.2ns which shows 15% improvement. This experiment depicts that the forward-interlocked behaviour of SELF can potentially dominate the performance of the full-interlocked asynchronous protocol in the context of the pipelined architectures.

4.8 Case Study: SSEM Processor

As a case study the Manchester Small-Scale Experimental Machine (SSEM) [Lav98] is exercised in this section. The high level specification of this computer is developed in Balsa and has been synthesised to hardware using the Balsa synthesis system [Bar98]. This machine comprises three separate stages which resembles commodity processors. Due to its simplicity, it is chosen as a case study to practise synchronous elasticity on a general purpose processor. Its Balsa description is synthesised using Teak and the new flow to generate asynchronous and synchronous elastic versions with the same level of granularity.

In Figure 4.13 the area cost associated with truly asynchronous and synchronous
4.8. CASE STUDY: SSEM PROCESSOR

elastic design styles is depicted. The results demonstrate that the eTeak flow achieves a substantial impact on area: up to $4.5 \times$ improvement. Each column is fragmented based on the entities existing in the circuit. Buffers are used to remove deadlock. For synthesis purpose Synopsis’ Design Compiler is leveraged to extract transistor level information. For the experiments in this work the UMC 130nm library is used.

This experiment also confirmed that SELF preserves slack elasticity which is the key property for further investigations in this thesis. The first pair of columns in Figure 4.14 shows the results in terms of area and performance for a fully buffered SSEM in which each link has storage (200 buffers). A GCD program with 30 iterations was run on SSEM in this experiment. According to the results, although the fully buffered synchronous SSEM has shorter critical path delay, its throughput is $3.7 \times$ degraded relative to the asynchronous dataflow.

As expected, fine-grained buffering the asynchronous dataflow can improve the overall throughput as it reduces the cycle time ($T_{\text{request}} + T_{\text{acknowledge}}$). This buffering policy can be too expensive in the synchronous domain for two reasons: a) each buffer consumes one clock cycle, b) unbalanced pipelines force the Join components to stall the data, especially in a processor architecture where data and control dependencies prevent efficient pipelining.

As discussed in section 4.4, SELF simplifies the loop structures in the dataflow network and allows using synchronous CAD tools to optimise the circuit, particularly computation-heavy data manipulation units and detect the combinational loops for sake of deadlock freedom. The second pair of columns in the figure demonstrate results with ‘reasonable’ buffering to ensure deadlock freedom. In this experiment the asynchronous SSEM has 65 buffers based on an algorithm for deadlock detection [Dua10], whilst the synchronous elastic version needs only 6 buffers to avoid deadlock.

Although eTeak results in a significant reduction in area, there are costs associated with this improvement. The synchronous elastic SSEM degrades throughput by a factor of $1.3 \times$. To tackle this, further slack matching [MM98][BLDK06] is required to balance the pipelines in the design and avoid unnecessary stalls. Chapter 6 proposes the De-elastisation technique which enables further improvements in this context.
Figure 4.14: The results in first pair of columns belong to a fully buffered SSEM. The second pair belong to a SSEM with high-effort area optimisation which achieves $4.5 \times$ area improvement.

4.9 Conclusion

This chapter proposes eTeak as a SELF adapted framework for synthesising synchronous elastic dataflows from fully asynchronous concurrent specifications. This approach not only preserves the properties of asynchronous dataflows, but also allows mature CAD tools to be employed for further improvements in the synchronous domain.

According to the challenges proposed in this chapter for fine-grained synchronous elasticity in communication and computation, a re-synthesis approach is proposed in the next chapter to replace the expensive dataflows of eTeak with Synchronous Sequential Machines (SSMs) in which rigid timing is introduced locally to overcome the fine-grained overhead of elasticity.
Chapter 5

eTeak Dataflow Re-synthesis

This chapter strives to reduce the fine-grained communication overhead of Teak Dataflow Networks (TDN). In this regard it presents a re-synthesis technique which exploits the rigid timing discipline of synchrony to lower the overhead. In dataflows, the eminent number of dependencies may bottleneck the performance because complex buffering algorithms are needed to capture data at the required time instants [HS12]. In this regard, a method is proposed that enables re-synthesis of TDNs using conventional synchronous Electronic Design Automation (EDA). The aim of this transformation is to target a particular kind of architecture: control-dominant systems. The distributed nature of Teak networks allows applying local transformations and optimisations. This paves the way for realising a co-design framework where data-dominant (GPU-like) segments of the high-level code are synthesised as dataflows whilst control-dominant (CPU-like) segments are realised as Finite-State-Machines (FSMs).

The idea of re-synthesis in this chapter leverages the Latency-Insensitive (LI) property of the Teak networks [CMSV01]. The LI theory emerged as the foundation of a correct-by-construction approach for designing complex systems by integrating pre-designed IP blocks. Its distributed nature allows composition of functional modules that exchange data on communication channels. Teak dataflow networks follow the same concept where grains are macromodules composed either in parallel or sequence and communicate through elastic channels.
The next section describes SSMs and, following that, the technique of transforming macromodules into RTL for re-synthesising is proposed. Later, Section 5.4 describes how to insert the re-synthesised entities back to the dataflow network by desynchronising them.

## 5.1 Synchronous Sequential Machines

Every digital design consists of state holding elements and combinatorial logic to implement some function. Transitions between states can be controlled either by a global signal, clock, or local synchronisation through local handshakes. The former is known as ‘synchronous’ or sequential logic and the later as ‘asynchronous’ logic. A handful of tools has been proposed to transform (or synthesise) logic behaviour into circuits. The term commonly used for these tools is Electronic Design Automation (EDA). Nowadays EDA is powerful enough to produce efficient large-scale circuits and prevent designers from getting involved in circuit-level details such as critical path measurement for setting up the clock, power estimation and placement/routing. Popular commercial EDA, such as Cadence’s Design Compiler or Synopsys’ IC Compiler, mostly target Synchronous Sequential Machines (SSMs) as these are straightforwardly analysed and generated.

**Definition 1.** A SSM is a network of combinational logic, such as binary gates, and sequential logic, such as registers. In an SSM a cycle consisting only of combinational elements is not allowed. Synchronous EDA is able to synthesise SSMs from behavioural or structural HDL specifications into hardware.

The most common form of SSMs are Mealy and Moore machines modelled as Finite State Machine and Datapath (FSMD) [Bro07]. A FSMD consists state elements to perform scheduling based on time steps, transitions between states and signals to instrument the functions in datapath. Unlike dataflows, in FSMD concurrency has to be realised by the designer with a careful arrangement (aka operator scheduling) of the transitions between the states. A comprehensive study of different scheduling techniques is proposed by Bezati [BEZ15].

The state-based nature of SSMs allow synchronous EDA tools to consider several optimisations, such as redundancy check and merging the states to make the design
5.2. TRANSFORMING ETEAK DATAFLOW NETWORK TO SSM

Figure 5.1: The communication states associated with each link in a synchronous elastic dataflow processor is monitored.

as efficient as possible. However, the ease comes with cost: designer is supposed to implement the system at register transfer level in terms of states using a Hardware Description Language (HDL), such as Verilog [TM02]. This, significantly decreases the productivity as the designer has to specify system’s functionality taking timing into account.

To exploit the synthesis capability of EDA in eTeak and avoid its costs an automatic transformation technique is proposed which is explained in the next section.

5.2 Transforming eTeak Dataflow Network to SSM

Introducing synchronous rigidity to a fine-grained elastic network for removing the unnecessary communication overhead is the motivation for this section. The proposed method enables synchronous EDA to be employed for re-synthesis of selective portions from the Teak dataflow networks.

Figure 5.1 shows a sample of the cycle-accurate switching activity of the 243 channels in a eTeak generated CPU. The processor is a SELF-adopted iterative architecture running at 435 MHz. In each cycle symbols indicate the activity on that communication channel based on the SELF protocol: Idle, Retry, Transfer. The red triangles represent a state where a channel is stalled and a transfer ‘retry’ is requested. In an
ideal system, retries should not be needed as they waste clock cycles; although the communication delays are tolerated in a latency-insensitive system, the fine-grained nature of the network may impose a profound impact on performance. If all the retries can be eliminated the control mechanism can then be removed in the structure. This reduces the logic and the critical path, allowing a performance gain and area reduction. Moreover, due to the fine-grained combinational communication between components, several glitches may trigger on each channel before they stabilise and clear for the next cycle. Although, as in any synchronous system, the glitches resolve before the next clock edge, they are potentially influential on power. Therefore moving towards coarse-grained structures will handle the aforementioned issue with fine-grained elasticity, and yet benefit TDNs with coarse-grain elasticity at system-level.

To alleviate the handshake overhead, synchronous rigidity is introduced to the network to enable synchronous EDA re-synthesise the system locally to reduce the fine-grained communication overhead whilst the environment enjoys the inherent properties of asynchrony. Figure 5.2 shows an abstract view of the hierarchical transformation where a random cut of a TDN is implemented as a SSM and reinserted into the graph. In this regard an RTL transformer which interfaces eTeak with synchronous EDA is proposed. The following explains the transformation algorithm and section 5.1 discusses the details of binding a synthesised synchronous machine into the Teak dataflow network.
5.2. TRANSFORMING ETEAK DATAFLOW NETWORK TO SSM

1: procedure ExtractFSM(part)
2:   DFS visited comp
3:   if partialVisited visited comp ← return visited
4:     else
5:       comp ← getNextComps
6:       FSM ← insertFSM . visitComps
7:       newVisited ← fold (markJoin part) visited comp
8:       fold DFS newvisited comp
9:       return FSM
10: end procedure

Figure 5.3: eTeak Dataflow Network to RTL Transformation algorithm

5.2.1 Code Generation: The Algorithm

The method described in this section performs state-space exploration of the network which is essentially the same as executing the graph. The proposed approach extracts all the possible data manipulation scenarios from the graph by traversing the control flow. It takes out the operations and maps them into clocked states where dataflow is implicitly governed by clock and data assignments, explicitly done by evaluating the expressions and moving the values into the variables. This way, the enclosed control signals in the datapath are separated from data, which allows a realisation of the system in the form of control-driven RTL.

The algorithm, shown in Figure 5.3 starts traversing the Control-Data Flow Graph (CDFG) from a given component (source) using a Depth First Search (DFS) policy by invoking the top-level function, ExtractFSM. The function terminates when it reaches an output link or a given arbitrary component (sink). In this way, all possible data paths get detected. Whilst searching, the visitComp function visits the components in the path and translates the associated functionality to Verilog expressions. Thereafter, insertState does processing and inserting them into the state graph. Finally an FSM writer function outputs the states with associated data expressions and variable assignments considering the order of the states.

Definition 2. [State Graph] is a directed graph denoted by a triple: \( SG = (S,C) \), where \( S \) is a finite set of states representing expressions, assignments and statements with a finite set of arcs denoting input and output links. \( C \) is the number of states \( |S| \)
CHAPTER 5. ETEAK DATAFLOW RE-SYNTHESIS

in the SG graph used to generate the sequence in Verilog.

**Definition 3.** [Execution Scenario] is a sequence of eTeak components (see Section 3.1) that the search function extracts from a source to a sink in a macromodule. A source/sink can be either an already visited component or an input/output port.

In the hardware/software co-design context, scenario extraction is a well known technique used for transforming concurrent high-level descriptions to sequential assembly code runnable on general purpose CPUs.

A linear pipeline (choice-free) is a singleton execution scenario that receives a set of data from its input ports, manipulates the data, and finally writes to the output FIFO, similar to a software thread. Due to presence of Steer (Choice) in the eTeak network the control flow is not linear. Therefore, the extracted scenarios for a non-linear network might encounter the well known problem of state explosion. To tackle this, there are two solutions: a) Merge the scenarios to reduce the state space, (aka multi-threading in the software domain) or b) Choose target sub-networks with a limited number of components for the transformation. The proposed method employs both of the solutions as the dataflow network is extremely fine grained.

5.2.2 RTL Implementation of eTeak Dataflows

Next the RTL implementation of the Teak primitives are explained. It should be noticed that the transformer does not preserve the handshake property of the primitives. In this regard, to ensure the correct functionality additional buffering may be required.

1. **Fork**: introduces concurrency to the circuit which: a) activates two or more macromodules (MM) at the same time and b) supplies them with data. It is a parametrisable component capable of carrying any number of bits from input to outputs. In case (a) as long as the macromodules are independent they will function in parallel. As illustrated in Figure 5.4 – (Case A), MM1 and MM2 are activated by a control Fork. Due to MMs’ independent nature they are translated into concurrent computations in Verilog, but in (Case B) due to the dependency between MMs the translator considers them in sequence. In (b) the Fork is translated to assignments of output links by the input link.

2. **Join**: concatenates data inputs. A two-way join of n and zero bits can be used as
5.2. **TRANSFORMING ETEAK DATAFLOW NETWORK TO SSM**

**Case A**
When Root is a Fork and MM1 / MM2 are independent:

always @ (posedge CLK) : FSM_A1
Out_1 <= φ1 (A, B)

always @ (posedge CLK) : FSM_A2
Out_2 <= φ2 (A, B)

assign Out = Join (Out_1, Out_2)

**Case B**
When Root is a Fork and MM1 / MM2 are dependent:

always @ (posedge CLK) : FSM_B
State1: Out_temp <= φ1 (A, B)
State2: Out_2 <= φ2 (A, B, Out_temp)

assign Out = Out_2

**Case C**
When Root is a Splitter/Steer:

always @ (posedge CLK) : FSM_C
State_Root:
  Case (A, B)
  1: State1
  2: State2
State1: Out_1 <= φ1 (A, B)
State2: Out_2 <= φ2 (A, B)

assign Out = Merge (Out_1, Out_2)

Figure 5.4: Three different patterns of macromodules in the Teak Dataflow Network realised by the Fork/Join and Steer/Merge primitives: Case A) concurrency, Case B) dependency and Case C) arbitration.
a conjunction of data and control. When the input links are not control signals, the translation is a simple concatenation followed by an assignment, but when there is at least one link with zero width, it implies control dependency which means that other data links should wait for a task to complete. The ExtractFSM algorithm (Figure 5.3) considers this Join after making sure that the associated scenario with the zero width signal is already extracted: \textit{markJoin} function takes care of that. This kind of Join acts as a sequencer [EB02] in Balsa's terminology. Later, buffering will be required to ensure that data arrive at the Join at the same time.

3. \textbf{Steer}: chooses the output path based on the incoming data value, so it functions as a data driven de-multiplexer. Therefore it is able to change the control flow which is similar to ‘if/else’ or ‘case’ statement in RTL. The transformer produces a case switch in Verilog whenever this primitive is encountered while traversing the graph as shown in Figure 5.4 – (Case C).

4. \textbf{Merge}: multiplexes input links based on first-come-first-served policy, so inputs should be mutually exclusive. This component is also parametrisable, meaning that Merge could act as a data or control multiplexer. In the RTL context, a data Merge is where several scenarios come together. Therefore their successor components are the same. ExtractFSM considers this to detect the overlapping scenarios and remove the redundant states. A control Merge will get removed since it is not a part of the dataflow.

5. \textbf{Variable}: stores data permanently. A variable in the Teak dataflow network has a single write port and multiple parametrisable read ports. The read-after-write (RAW) and write-after-read (WAR) links in the control flow, shown in Figure 5.5, allows distinguishing reads and writes and put them into separate states. In the RTL context, initially all the variables are defined as multi-bit registers in the beginning. A read from a variable is translated as assigning the content of the register to the output bus. Similarly, a write to a variable is translated as assigning the current value of the input bus to the register.
5.2. TRANSFORMING ETEAK DATAFLOW NETWORK TO SSM

5.2.3 A Case Study: Sparkler Shifter Unit

To evaluate the proposed method the motivating example, shifter, from the previous chapter has been exercised. The shifter macromodule is a pipeline-like structure synthesised from a recursive high-level description in Balsa. The generated circuit is shown in Figure 5.5 which has internal control dependencies which makes it a suitable candidate for explaining the flow clearly. It is a two-stage 32-bit shifter unit from Sparkler processor, a cut-off version of the SPARC v8 architecture [SPA], implemented in the Balsa language. It is capable of shifting at most two bits per iteration. For sake of simplicity, the number of stages is reduced from five to two yet preserving the same control flow. The corresponding Balsa code is depicted in the previous chapter (see Figure 4.12).

The code is written in a recursive way to demonstrate the power provided by the language. The case statement within the local shift-body procedure determines the core functionality of the circuit. In this example shift-body is called twice so the Teak compiler unfolds it and creates a separate stage for each. The ‘distanceI-shift’ variable captures the input values from ‘distanceI and shift’ input ports upon which the number of bits for shifting the data (i) at each stage gets decided.

Due to the syntax-directed translation, one-to-one binding exists between the statements in the code and the generated structures for variables (V), operations (O), Steer (S), Merge (M), etc. The figure shows the shifter dataflow graph in which the wide coloured links are for carrying data and narrow black for control tokens. Every variable has a single write port ($W_0$) and may have several read ports ($R_n$).

The whole graph can be viewed as a separate macromodule with four input and two output ports where one of the outputs is fed back as input to form an iterative structure. After receiving a go, the data provided by ‘shifter and distanceI’ input ports flows into the network and gets stored in the distanceI-shift variable (C3). Thereafter, a read-after-write (RAW) link allows the arg port to input the data through an independent channel. The input value gets stored in variable i (C6). Then, the associated RAW link becomes active and acknowledges the control flow (C7) that the data is available and processing data can start.
Figure 5.5: Sparkler shifter graph with two stages of data manipulation which is simplified to depict the control and the dataflow separately.
5.3 RTL for the Shifter Case Study

The RTL code generated for the shifter case is depicted in Figure 5.6. The red dotted route from Figure 5.5 shows a possible dataflow scenario in the graph whose associated states \{1-19\} are depicted in the figure. The code is in the form of an FSM in which the state flow is extracted from the CDFG of the shifter unit. In this section the main target is to produce functionally correct and synthesizable Verilog code so that the transformation, described in the next section, can be applied. Code optimisation is out of scope of this work and it could be addressed in future.

In the graph, the RAW link from the distanceI-shift variable (C3) gets distributed by a control Fork (C4) between the stages. It implies that to start computation this variable has to be read by more than one stage concurrently. Due to data dependency between the stages at C11 it is not possible to execute them as concurrent FSMs. Therefore the algorithm ignores (C4) and inserts the corresponding states with each stage into the state graph in sequential order. In Figure 5.6 states \{7-11\} are for the first stage and \{12-19\} are connected with the second one.

Another possible scenario may take place when Steer (C9) chooses the second option which transfers the control to state 20 where data is read from (C3) and fed to (C17), a three-choice Steer, which is pointing at (C12) with four read ports. Each branch is able to take the data from the variable and carry it to the output (sink).

Therefore each Steer is able to span the state graph by factor of \(O(N)\) where \(N\) denotes the number of branches a Steer has. In this case, due to data dependency between the first and second stage, the Steer components interfere which results in \(O(3+1)\ast O(3+1)\) different scenarios. The proposed method can partially transform a TDN to an FSM. In this case study it extracts 16 execution scenarios with 56 states in total. It should be noted that scenarios are merged (line 8 and 15 in Figure 5.6) to reduce the state space. The output for this process is shown in Figure 5.6. Figure 5.7 visualises the generated FSM for a more complicated dataflow graph in the eTeak environment.

For further analysis it is possible to investigate the graph based on the scenarios and extract the computation rates associated with reads/writes from/to output/input.
module teak_Shifter (go, shift, distanceI, result, arg, clk, reset);

    input go;
    input [1:0] shift;
    input [4:0] distanceI;
    output [31:0] result;
    input [31:0] arg;
    input clk;
    input reset;

//internal links defined as wires
//variables defined as registers
//next_state and state variables defined here

always @ (posedge clk) begin : FSM_SEQ
    if (reset == 1'b1) begin
        state <= #1 1'b1;
    end else begin
        state <= #1 next_state;
    end
end

always@(*) begin : FSM_COMB
    case(state)
        0: next_state = 1; //go
        1: L128 = {shift,distanceI}; next_state = 2;
        2: L150 = L128[6:0];L151 = L128[2:2];L147 = L128[1:1];next_state = 3;
        3: distanceI-shift = L150; next_state = 4;
        4: L72 = {arg}; next_state = 5;
        5: i1 = L72; next_state = 6;
        6: L187 = {L151}; next_state = 7;
        7: L188 = L187[0:0]; next_state = 8;
        8: case (L188): 0: next_state = 9; 1: next_state = 19;
        9: L78 = i1; next_state = 10;
        10: L87 = L78; next_state = 11;
        11: L35 = {L87}; next_state = 12;
        12: i2 = L35; next_state = 13;
        13: L184 = {L147}; next_state = 14;
        14: L185 = L184[0:0]; next_state = 15;
        15: case (L185): 0: next_state = 16; 1: next_state = 33;
        16: L105 = i2; next_state = 17;
        17: L112 = L105; next_state = 18;
        18: result = L112[31:0]; next_state = 1;
        19: L174 = distanceI-shift; next_state = 20;
        //...
        default : $display("ERROR in FSM_COMB");
    endcase
end
endmodule

Figure 5.6: Verilog SSM for the shifter example with 56 states
Figure 5.7: The graph on the left shows a concurrent dataflow implementation of a computation block and the graph on the right shows the SSM transformation of the same block with 354 states which is re-synthesisable through commercial EDA.
FIFOs. This enables the tool to record a high-level communication and computation pattern for every transformed portion of the network which can be used towards GALSification.

So far the method of transforming macromodules in a TDN into an intermediate HDL representation is presented. The generated HDL code is in FSM format extracted from CDFG with enclosed expressions within the states to realise the datapath components. Considering this method, it is possible to use synchronous EDA to synthesise and optimise TDNs partially. EDA takes the extracted FSM and re-synthesises it into a synchronous circuit. To insert the circuit back into the system and enable it to communicate with other components, it needs to adopt the SELF protocol, the global timing discipline in TDNs. The following section describes the corresponding process.

5.4 Synchronous Machines to Patient Systems

The previous section describes how the Teak Dataflow Networks are transformed to RTL and re-synthesised using conventional EDAs. In this section the process of inserting the transformed structures back into the dataflow network is described. The major concern of this section is to preserve correct functionality and allow the Re-synthesised Blocks (RB) to cooperate with the dataflow network. Due to the slack elastic [MM98] nature of the dataflow networks bounded delays are tolerated and timing is not an issue. The rigid timing of RBs make them intolerant against delays, such that only one-cycle latency of the input data would fail the whole computation. There are two solutions for this issue: a) Transforming RB into a Patient System [VA09a], or b) Transforming RB into an elastic circuit [KCKO06] before inserting it back into the network.

As Vijayaragavan et al. [VA09a] define it, a patient SSM is a latency-insensitive machine [CMSV01] whose registers are controlled by a global enable signal. When this signal is low, the state of the sequential elements freezes; no state updates occur. Any SSM is transformable into a patient SSM [VA09b].

Theoretically, this allows defining random cuts possible when decomposing SSMs into very fine-grained entities. Although the technique of transforming random SSMs to patient systems extends Carloni’s model of designing latency insensitive systems and
ensures correct functionality, its ‘global’ nature can impose performance bottlenecks. Regarding that the computation model employed in this work is distributed a method with ‘local’ nature is preferable. Therefore this work relies on the de-synchronisation technique for transforming RBs into elastic circuits, no matter what size the block is. The following defines a patient SSM based on the de-synchronisation technique:

**Definition 4.** A patient SSM is a latency-insensitive machine whose registers are controlled locally through handshake signals of Valid and Stop operating based on the SELF protocol which is previously explained in Chapter 3.

After generating the SSM for the corresponding partition/module using the proposed method in the previous section, the machine is transformed to a patient system (see Definition 4) whose clock is controlled locally by elastic blocks (see section 3.2.3). Figure 5.8 shows the ALU module from the Sparkler processor implemented in the Balsa language. The shifter module from the ALU is re-synthesised and reinserted into the ALU dataflow.

According to the slack elastic property of eTeak dataflow networks, buffering the input and output ports for any degree does not affect the functionality. This property is demonstrated in Section 4.8 where every link of a processor is fully buffered and its asynchronous and synchronous elastic behaviours are studied. For synthesis purposes
Synopsis’ Design Compiler is used to synthesise the shifter unit. There is no reason why this method cannot be applied to more complicated synchronous machines. However further code optimisations will be required.

After re-synthesis extra buffer insertion is required to balance the pipelines to avoid unnecessary stalls which contributes to the performance. This technique is known as slack matching [BLDK06] and is the major motivation for the next chapter. In terms of power, a reduction in power consumption is expectable as the re-synthesised blocks exploit ‘less’ concurrency compared to their dataflow counterparts in general.

After re-synthesis the ALU is evaluated. The experiments have shown that the critical path associated with the re-synthesised shifter has improved from 2ns to 1.2ns (40% shorter) which means almost 300 MHz faster architecture. However, the area has grown by $5\times$ due to the excessive use of flip-flops. To apply this method on a complicated system, like the SSEM, with more than 130 handshake components and 12 cycles this technique has to be optimised otherwise it would end up with extremely large circuits. However, the generated circuits have shorter critical path and follow a rigid timing discipline.

### 5.5 Conclusion

This chapter starts off with analysis which show that elasticity at component level suffers from prohibitive costs in terms of performance as communication overhead dominates computation. Later, a re-synthesis approach is introduced to replace the fine-grained elasticity and overcome its overhead locally. In this respect, a technique is presented which transforms dataflow macromodules of eTeak into Synchronous Sequential Machines (SSMs). Synchronous machines are expressed in the form of Verilog code which are synthesis-able by conventional EDA flows. To reinsert the re-synthesised block into the network safely, it has to be transformed into a patient system which is defined based on the de-synchronisation technique.

In spite of the evidence that demonstrates the efficiency of the proposed technique at small or medium scale, control space explosion could be encountered for large-scale systems. Therefore a decomposition method needs to be employed to reduce the design space. In the next chapter a scalable technique is proposed, called de-elastisation, which enables eTeak for further system-level explorations.
Chapter 6

De-elastisation: from Elasticity to Local Rigidity

This chapter presents the De-elastisation technique which enables eTeak to selectively transform elastic dataflows from a high-level description into rigid synchronous circuitry in hardware, hence providing a mechanism for synchronous/asynchronous co-design exploration.

As discussed in the previous chapters the large overhead of fine-grained elasticity in the eTeak dataflow networks (eTDNs) could be prohibitive in different aspects, which motivates the research proposed in this chapter. As was shown earlier, local architectural transformation of eTDNs into Synchronous Sequential Machines (SSMs) may encounter the well-known problem of state space explosion and demands further optimisations to preserve the intrinsic concurrency of macro-modules after the transformation. An alternative approach, which is explored in this chapter, is to rely on the macromodule architecture of eTeak and exploit a buffering (or re-timing) technique to alleviate the elasticity overhead selectively from macro-modules. The major advantage of this technique is that it preserves the fine-grained pipelines and, consequently, shorter cycles of asynchronous design to yield higher performance versus synchronous RTL counterparts.

De-elastisation is evaluated and compared against some popular high-level synthesis technologies, namely LegUp, Bluespec, Chisel and Balsa using a set of benchmarks from the domain of Database Management Systems (DBMS) accelerators and general
CHAPTER 6. DE-ELASTISATION: FROM ELASTICITY TO LOCAL RIGIDITY

purpose processors. Algorithms such as bitonic sorting, spatial sorting, median operator and hash joins are considered to represent fine-grained, GPU-like computation, pipelining, aggregation and producer/consumer models respectively, whilst processor case studies are exercised to evaluate control dominant algorithms. The experiments demonstrate the efficacy of De-elastisation in terms of energy and performance on the selected range of applications and its advantages in exploring the design trade-offs against De-synchronisation.

6.1 Introduction

Asynchronous systems facilitate fine-grained pipelines and, consequently, shorter cycles yielding higher performance [LVTS13] [CKLS06], particularly in streaming data applications. The elastic nature of asynchronous systems makes them tolerant to variable latency in communication and computation. With data-dependent computation times elasticity may contribute to average- instead of worst-case performance [JE12a].

As was shown earlier, while elasticity gives the circuit timing flexibility, it comes with a cost: communication must be synchronised by a handshake mechanism, imposing both an area and performance penalty compared to a synchronous model where assumptions about data readiness are statically engineered. In practice this overhead can overwhelm any advantages and not all the elasticity is useful. The method employed here selectively removes elasticity from an elastic circuit, adopting a clocked protocol without handshaking in selected parts of the circuit without forcing synchronisation on the system as a whole. In other words, rigidity is introduced locally into the fine-grained networks of eTeak whilst preserving elasticity globally. This results in a GALS (Globally Asynchronous Locally Synchronous [Cha84]) system; regions may be run by different clocks or the intervening elastic buffers may use synchronous handshakes.

Whilst De-synchronisation [CKLS06] enables synchronous designs to exploit asynchronous advantages such as re-timing, De-elastisation allows asynchronous systems to exploit the rigid timing behaviour of synchrony to alleviate the communication overhead and bring the handshakes to a coarser level where it is possible to run parts with different clocks whilst the ecosystem remains asynchronous. This provides eTeak with
6.2 Elasticity for Handling Uncertainty

Elasticity emerged as a solution to uncertainty in computation and communication delays. The idea was introduced by Carloni et al. [CMSV01] at system level and was formalised by Carmona et al. [CKG06] for exploitation in CAD flows, which made it applicable from transistors to the system level. Elasticity comes with costs: if the designer chooses to apply elasticity at a fine-grained level, its communication costs may prohibitively dominate computation costs.

Elasticity is able to deal with various sorts of non-determinism; for example in a multiprocessor computer the delays of the different memory subsystems may vary – for instance depending on whether a particular item is cached or speculated in...
the processor pipeline. Figure 6.1 illustrates where elastic buffers could offer more flexibility in such cases [HB08] [GOCK09].

Elasticity and its applications have appeared for GALS at system level [DDK13] [JMGTE14] and network-on-chip [MD13]. Another source of non-determinism appears when data-dependent loops exist in computation – clearly a source of timing uncertainty in a dataflow. This paper shows how these loops can be classified as blocking loops with bounded latency in terms of clock cycles. Elasticity is kept at the boundaries of these sorts of structures to address their non-deterministic behaviour.

‘De-elastisation’ should be considered with some circuit level restrictions. In this regard, the macromodule circuits of eTeak are classified into two categories: blocking and non-blocking loops [LNZ14]. Blocking loops have data access from the environment through channels that push data into the circuit; non-blocking loops are where data is requested (or pulled) and no stall is required. These models are analogous to the ‘polling’ and ‘interrupt’ mechanisms at system level. Section 6.2.1 describes discriminates the loops in the dataflow context.

The contributions of this chapter is to propose a method to partially substitute the elastic protocol in a dataflow network with rigid clocked timing, considering the architectural restrictions. In this regard, classifying systems based on their behaviour becomes necessary, hence the designer is provided with information about different design styles which contributes to his/her efficient decision making at higher abstraction level. The technique also proposes a comprehensive solution to cover data-dependent loops, which are the main source of non-determinism in system-level design. De-elastisation decouples them from the rest of the design and preserves elasticity at the boundaries (selectively) to tackle their non-deterministic behaviour.

### 6.3 De-elastisation

There are various techniques in the literature proposed to improve asynchronous circuits. For performance, prior art has addressed the slack matching [BLDK06] problems for both unconditional (choice-free) and conditional circuits. Unconditional circuits, like synchronous circuits, can be re-timed by buffer insertion. For conditional circuits, Beerel et al. [BX02] have proposed transforming such circuits to unconditional
by unfolding their scenarios. Although this technique achieves a solution in a reasonable time, it may encounter state-explosion in large-scale problems and may not be scalable. In a more practical approach, Martin et al. [MNP+03] considered a microprocessor as an unconditional circuit and used a synchronous re-timing technique to slack match the circuit; this might result in over-buffering the circuit. Works by Gill et al. [GGS08] and Najibi et al. [NB13] also address the existing non-deterministic behaviour through estimating the worst-case/upper bounds for the performance of these circuits. Simulation-based techniques have also been investigated [YFIR13][VG06] through iteratively tracing the signals in the circuit. Carmona et al. [CJCK11] have also tackled the control overhead of the synchronous elastic circuits by replacing the handshake signals with static schedulers that can produce control signals for a cluster of registers. This technique is successful in reducing the control overhead of elasticity, however the target benchmarks for evaluation have deterministic behaviour. This technique may need an extension similar to the proposed work for handling non-deterministic behaviour in the general purpose circuitry.

Other optimisations, closely related to this work, include resynthesis and peephole techniques whose aim is to modify the behaviour of the components, either by protocol change or component composition based on Petri-net models and tracing-theory [AKM+11]. In this context, Tarazona [Dua10] and Dimou et al. [DBL14] have proposed a set of compositions at component level to improve Teak networks and a clustering approach at gate level to form asynchronous coarse-grained pipelines, respectively. A similar approach was used to transform Balsa circuits into coarser structures to improve performance [CN02] [PTE05].

De-elastisation inherits from slack matching and the re-synthesis techniques above. De-elastisation slack matches pipelines using static scheduling to remove non-determinism from ‘Choice’ (Steer) elements and is able to replace/resynthesise elastic Join/Fork/Buffer components with their inelastic counterparts, significantly boosting potential clock frequency. The flow proposed here transforms the design from an elastic dataflow to a synchronous solid structure in which elasticity is occasionally preserved based on the architecture classification. This technique is believed to be the first to optimise the elastic dataflow networks by considering circuit level bottlenecks. This work also contributes to behavioural partitioning of the system by classifying the architecture.
De- elastisation maintains dataflow concurrency at system level whilst proposing a rescheduling and re-synthesis method to reduce the level of communication between the state holding entities with respect to their behaviour and physical timing characteristics. In other words, De-elastisation removes fine-grained communication and corresponding overhead by introducing local timing rigidity to the design. The proposed transformation can be viewed as a technique that intends to trade-off communication at software level and efficiency of computation at hardware level towards a lower overhead and enhanced productivity.

Figure 6.2 illustrates how De-elastisation differs from the conventional approaches in high-level synthesis [BVR+12] which almost neglect the elasticity persistent at higher abstraction level whilst spending significant effort to handle timing complexity at lower levels, such as clock distribution which is one of the most important phases in digital synchronous design. De-elastisation preserves elasticity between the computing blocks at interconnect level. This contributes to the designer’s productivity as retiming the circuit after place and route will not be necessary. Accordingly, a comprehensive study of the high-level models in the dataflow context is proposed, along
with a programming guide for the designers, in the next section.

### 6.3.1 Loops in eTeak Dataflow Networks

To safeguard functionality loops are categorised into *blocking* and *non-blocking* architectures. These models are commonly used at different levels of abstraction. For instance, at operating system level for handling IO devices the ‘polling’ and ‘interrupt’ mechanisms are considered which follow the exact communication pattern at higher level between devices. These mechanism are categorised under asynchronous IO or a non-blocking IO processing model that allows other computations to continue before the communication has terminated. Dataflow systems are well-known for their concurrent nature which is in a close relation with the implementation style that exploits push channels – where data tokens instigate a transfer. In contrast to conventional dataflows [TEPTD10], Teak provides pull channels (triggered by a *want* of data) to read data which significantly simplifies its storage structures.

Accordingly loops in the Teak dataflow networks are categorised into two types; Type 1: the loops that pull data on-demand become non-blocking (suitable for modelling determinism) and Type 2: which receive data through push channels emerge as blocking (suitable for modelling non-determinism). In Type 1 (figure 6.3) data is pulled so a bounded degree of latency is expectable and deterministic, therefore the on going computation will not be blocked; Unlike Type 1, in Type 2 (figure 6.4) data is pushed...
throughput is denoted by
selectively
non-determinism in eTeak dataflows so that De-elastisation can be applied
into the loop and it may be blocked by Join/Fork guard until the on going compu-
tation terminates. The proposed classification is leveraged to mark determinism and
non-determinism in eTeak dataflows so that De-elastisation can be applied selectively.

6.3.2 The Intuition

Before describing the De-elastisation algorithm an example is presented here which
is necessary for understanding the impact of the process on performance as De-
elastisation may improve critical path delay and throughput at the same time.

An intuitive understanding of the De-elastisation technique can be obtained by
analysing the example shown in Figure 6.5, where (a) is a dataflow realisation of
the high-level expression of C1 ; [(C2 ||C3) ? C4] in Balsa, where C2 and C3 are
specified as parallel computations and are merged with C4. C1 is also considered in
sequence with the rest. Each computation could represent a simple arithmetic unit or
a complicated module, depending on the design’s abstraction level. For performance
analysis blocks are annotated with arbitrary delays in this example. The question mark
in the expression implies data-dependency and infers a Steer-Merge pair (aka Choice)
with \( \alpha \) and \( \beta \) probabilities on the \( a \) and \( b \) branches, respectively. The associated
throughput is denoted by \( \theta \) and is proportional to:

\[
\theta \propto \frac{\sum m(e)}{\gamma \cdot \delta}
\] (6.1)
6.3. DE-ELASTISATION

Figure 6.5: The energy and performance impact of Decomposition and De-elastisation on a eTeak-generated dataflow circuit: a) shows the dataflow realisation of the C1; [(C2 || C3) ? C4] expression as a single cycle, b) is a decomposed implementation of (a) into two shorter cycles and c) is the de-elastisised version of a cycle from (b) where extra elastic blocks are inserted to balance the pipelines and remove the Join/Merge components. The critical path associated with each cycle is shown with red dots.

where \( m \) denotes the sum of active tokens on the edges (e) of the loop, which is assumed to be one in this example; \( \delta \) represents the critical path delay which, in this example, equals to the associated delay of C3 (4ns) plus the delay of the combinatorial components, which are assumed to be 0.1ns each. Therefore the overall delay is 4.4ns (the red dotted path). \( \gamma \) also represents the Local Cycle Time (LCT) for every single cycle in the design. LCT is defined as the time between the arrival of a token and the time that the stage is ready to receive another token [BOF10]. In this example \( \gamma \) equals to 4.\( \alpha \) + 3.\( \beta \).

Figure 6.5(b) depicts a decomposed implementation of (a) into two shorter cycles. Decomposition may improve the overall throughput by increasing the number of active tokens in the system and decreasing LCT. Since in the synchronous elastic circuits of eTeak time is discretised, we measure LCT according to the clock. In this example the number of active tokens is doubled after decomposition and the new LCT is:

\[
LCT_{\text{new}} = \max\{((\alpha \cdot 2 + \beta \cdot 1), 2)\}
\]  

(6.2)

However decomposition does not guarantee an improvement on the critical path. In this example \( \delta \) associated with the second cycle has increased to 4.6ns which forces the system to run at slower clock rate unless a multiple-clock discipline is exploited.
In this way, each sub-system can run at different rate whilst exploiting channels for communication. Adopting this scheme has energy advantages and is a subject for future work.

To ensure that decomposition yields an improved throughput it is accompanied with De-elastisation to boost the clock rate by re-timing the system. Figure 6.5(c) shows a re-timed version of the second cycle where elastic components are selectively removed by balancing the pipelines via buffer insertion. Outer Join/Fork pair is untouched as De-elastisation preserves elasticity at the boundaries (or interfaces). Same as synchronous circuits, re-timing invalidates the data-dependant property \( \alpha = \beta \) and forces every cycle in the design to have identical delays. In other words, the worst-case scenario is considered for every cycle in the design. It should be noted that De-elastisation is a selective process and calls for an appropriate decomposition at first place to avoid unnecessary buffering which may have prohibitive impact on the throughput of the elastic circuits [MKG16]. After all, in this example De-elastisation along with decomposition yields a new throughput with 45% improvement.

This example demonstrates how critical rate at higher level of abstraction and critical path at circuit level are connected in the dataflow context. To explore elasticity at different levels of granularity architectural information has to be taken into account. Section 6.5.1 explores elasticity with respect to the higher level patterns.

6.3.3 Algorithm

The De-elastisation algorithm is implemented in Haskell and integrated into the eTeak flow. It initially identifies loop structures and marks the communication ports/channels. The first three steps correspond to deadlock detection which might vary according to the protocol (see section 4.4). It has been shown that for ensuring deadlock freedom in the asynchronous QDI circuits of Teak three latches per cycle (or 2N+1 for N tokens) are necessary [Dua10] whilst in the synchronous elastic domain a single buffer (double latch) per token is enough [JMGTE14]. In step 2 a Depth-First-Search (DFS) is required to group the edges into fore-, back- and cross-edges (see definition of DFS).

1. Remove the algorithmic loop

2. Mark the back/forward/cross edges in the graph
6.3. **DE-ELASTISATION**

1: **procedure** SetReachabilityDepths(*network*)

2: let

3:  *inpPorts*  \(\xrightarrow{\triangleright}\) array of input ports

4:  *inpComps*  \(\xrightarrow{\triangleright}\) array of first adjacent comp. to each port

5:  \(\textit{visited} \leftarrow \emptyset\)  \(\xrightarrow{\triangleright}\) (previously DSFed components, *depth*)

6:  *stack* \(\leftarrow \emptyset\)  \(\xrightarrow{\triangleright}\) stack for DFS

7:  *finalList* \(\leftarrow \text{foldl (DFS stack) visited *inpComps}\)

8:  DFS stack visited *comp*

9:  if *comp* is in *stack* return *visited*

10: else push *comp* into *stack* and

11: Update *visited* with *comp*

12: foldl (DFS stack) visited *nextComps*

13: *nextComps* \(\leftarrow \text{AdjacentComps (comp)}\)

14: **return** *finalList*  \(\xrightarrow{\triangleright}\) List of Joins and Merges with associated depths

15: **end procedure**

Figure 6.6: First phase: assigns depth value to Joins and Merges in the network.

3. Buffer the back-edges to ensure deadlock freedom

4. Run the first phase: **Set Reachability Depths**

5. Run the second phase: **Deploy Buffers**

6. Remove Elastic Joins, Eager Forks and elastic buffers except the Joins that are hooked to push channels

**Phase 1: Set Reachability Depths**

The first phase (figure 6.6) assigns each component a value, ‘depth’, which indicates the maximum number of clock cycles it takes data to reach the target component from the input ports (*reachability depth*). This algorithm runs a DFS on each input port and finds reachability depths for every component in the design.

The algorithm implicitly avoids cycles in the graph. The associated complexity is \(O(V * N)\) where \(N\) is the number of input ports and \(V\) denotes the number of components DFS visits in worst case. The multi-input components {Join, Merge, Variable} may be visited multiple times by different DFS runs resulting in different
CHAPTER 6. DE-ELASTISATION: FROM ELASTICITY TO LOCAL RIGIDITY

```
1: procedure DeployBuffers(network)
2:    let
3:        table ← SetReachabilityDepths(network)
4:        compLinks ⊿ array of (comps,input links)
5:        map (eachComp table) compLinks
6:            where eachComp table (comp, links)
7:                = BufferLinks links $ DP table links
8:                where BufferLinks links List
9:                $ List is generated by DP, it contains the required number of buffers for
balancing the incoming links
10:   return
11: end procedure
```

Figure 6.7: Second phase: balance branches by deploying buffers.

depth values for a single component. The function Update (line 11) takes the maximum depth value. For simplicity, the depth value is not shown. Depth is incremented on each iteration of a DFS when the algorithm encounters a clocked entity, either a variable (through a write port) or a link which is already buffered to avoid deadlock.

Phase 2: Deploy Buffers

Having the multiple-input components annotated with their reachability depths in the first phase, the algorithm considers balancing the depths associated with the input links by deploying buffers (Figure 6.7).

The Dynamic Programming (DP) technique employed here uses the annotated list of phase one (table: line 3) to calculate the number of buffers for each input link of Join/Merge components. It should be noticed that the buffers and variables are the only clocked elements in the graph, so their proper insertion can influence the critical paths in the design. After passing the second phase, the BufferLinks function deploys buffers either before or after the combinational components (C* in Figure. 6.4 and 6.3) or closer to Joins as they are usually in critical paths.
6.3.4 Potential Improvements

The combinational nature of the Elastic Joins, especially those with high fan-in, contributes overhead to the critical paths. As was discussed in chapter 4 Eager forks [CKG06] are considered in eTeak circuits to avoid establishing combinational loops with the joins. By removing Elastic Joins in the course of De-elastisation, the fork components become redundant and are safe to be replaced with non-elastic components. The transformed circuits have reduced area and boosted clock frequency due to the removal of unnecessary combinatorial logic. Subsequent resynthesis using synchronous EDA may give further improvements.

Unlike classic GALSification techniques, De-elastisation enjoys an architectural degree of freedom for establishing synchronous localities/domains in the design. Regarding the slack elastic and distributed nature of the eTeak networks, it is possible to insert any bounded number of buffers on every link without affecting the functionality. In a FIFO based GALS design, FIFOs, which are basically elastic buffers, are inserted at the boundaries. To determine their size, the worst-case scenario is usually considered. This may impose area and energy overhead to the system [KGGV07]. De-elastisation can avoid this by reconsidering the elastic boundaries with regard to the intra-domain buffering requirements which may lead to more appropriate FIFO sizing, and thus less expense.

6.4 De-Elastisation vs. Elastic Circuits

To evaluate the proposed De-elastisation technique against the fully elastic eTDNs three different case studies are exercised and analysed: (a) SSEM, an iterative processor, (b) Sparkler, a 4-stage pipelined processor and (c) Bitonic sorter, a fine-grained GPU-like computation model. The purpose of this experiment is twofold: a) to compare De-elastisation to the state of the art techniques in the asynchronous domain, b) to study the impact of De-elastisation with regard to the high-level computation patterns and models.
6.4.1 SSEM Iterative Processor

The Small-Scale Experimental Machine (SSEM) [Lav98] is implemented in Balsa. This machine has three stages described as a single macromodule. Therefore it has a single algorithmic loop. In this example it runs a GCD program [Bar98] with 30 iterations. De-elastisation is applied to the Teak-generated circuit and the results are compared with their asynchronous and synchronous elastic counterparts (Figure 6.8). According to the classification proposed in Section 6.3.1, SSEM uses pull channels (Type-1) to read data from memory. Since the memory model used for evaluating the SSEM processor is a basic model with no caching capability, it has a deterministic latency and it can be de-elastisised completely. As discussed in section 4.5 Teak relies on a hybrid model (Push and Pull channels) for handling the memories or storage elements. SELF adoption has to be applied with careful consideration. For loading or storing data from/to a memory block, a set of signals including data, enable and read/write has to be set or reset simultaneously. Insertion of buffers for de-elastisation would never violate this rule as it essentially balances pipelines including the memory signals.

6.4.2 Sparkler Pipelined Processor

Sparkler is a cut-down version of the SPARC v8 architecture [SPA] implemented in the Balsa language. It comprises the four stages of Fetch-Decode-Execute-WriteBack and
Figure 6.9: Asynchronous, Synchronous Elastic and De-Elastic implementations of Sparkler Processor compared in terms of area and performance.

is pipelined, thus falling into the second category (Type-2). Sparkler’s pipeline stages are described as separate macromodules, thus each individually contains an algorithmic loop. This, in contrast to the SSEM, results in shorter cycles and allows multiple tokens get processed by the pipeline which contributes to a higher performance. Macromodules can communicate through channels. In this case study, separate memories are considered for the data and instructions. Unlike SSEM, it is assumed that the fetch stage pulls the instructions from the memory and pushes them into the successor stage in a producer-consumer fashion. Therefore, the process never get stalled. For evaluation, the Dhrystone loop is run on the asynchronous, synchronous elastic and De-elastisised processors. Figure 6.9 illustrates the results with each implementation in terms of area and performance.

6.4.3 Bitonic Sorter

The Bitonic sorter (Figure 6.11) is a pipelined sorting network comprising $O(\log^2 n)$ stages each with $n/2$ comparators which results in throughput bounded by $O(w \cdot n \cdot f)$, where $w$ denotes the bit-width of input elements, $n$ is the number of elements and $f$ is the clock frequency. A dataflow architecture of a 16-input 64-bit bitonic sorter is implemented in Balsa and synthesised using eTeak (Figure 6.12). While this hardware-friendly sorting algorithm is capable of achieving a high throughput, its resource usage
6.4.4 Results and Discussion

Table 6.1 compares De-elastisation against the state of the art in the asynchronous domain. In the last 8 years several buffering methods have been proposed and integrated to the Teak System: 1) Async [Dua10] – a deadlock-free buffering scheme for the Teak networks proposed by Luis Tarazona in his PhD thesis; 2) SCP [RWSE12] – a critical path aware technique for buffer insertion to improve performance; 3) SE [JMGTE14] – a technique to improve area and performance by incorporating the synchronous elastic protocol in Teak. The De-elastisation technique (Sync), proposed in this thesis, reaches a higher performance whilst reducing the buffer count for improved area (and performance).

The major difference between the first pairs (Async and SCP) and the second pairs (SE and Sync) in Figure 6.9 is due to the substitution from a clock-less to a clocked protocol. Minimizing the number of buffers in SE and Sync improves area and gives a significant performance boost. In the asynchronous domain, extra buffers increase performance by reducing handshake cycle latency, whilst in the synchronous domain buffer insertion improves clock rate but wastes clock cycles especially when buffers are...
Figure 6.11: SELF adapted dataflow network of a 16-input Bitonic sorter
Figure 6.12: eTeak generated dataflow network of a 16-input Bitonic sorter
inserted in non-critical paths. For instance, the execution time for an over-buffered asynchronous Sparkler is 2.01 \mu s whilst it is 7.2 \mu s for the synchronous elastic version (at 1.7 GHz) which is about 3.5\times worse.

For the sake of consistency, all the designs are evaluated using UMC 130 nm technology at typical operation conditions. The input Balsa descriptions are all the same and comparison against a fully synchronous RTL implementation is subject for the work in the next section. To measure area, Verilog netlists are technology mapped using the Teak back-end, \textit{Synopsys DesignCompiler} and the \textit{DesignWare} library to synthesise the synchronous functional units (operations). To measure performance \textit{Modelsim SE 6.3a} from \textit{Mentor Graphics} is used. Our algorithm runs almost 10\times faster than the SCP method [RWSE12] due to its light weight nature.

### 6.5 De-elastisation: A System-level Design Flow

Raising the level of abstraction provides the chance of applying elasticity at different levels of granularity. In this section an answer to the following question is pursued: what degree of elasticity is enough? This enquiry is highly connected with partitioning and the size of elastic grains in our explorations. Architectural investigations here, together with the explorations from the previous section, allows hypothesising about the recent contributions of De-synchronisation and SELF technologies: Why industry is reluctant to adapt elasticity at fine level of granularity? The studies in this chapter show that the selected architectures for global elastisation have not been appropriate enough.
In contrast to De-synchronisation, De-elastisation does not lend itself to a full synchronous design flow, instead it exploits the timing-free, software-like nature of asynchronous (or ‘elastic’) dataflows to neglect implementation details such as clocks and lets the designers focus on specifying the system functionality and postpone issues of timing to subsequent stages of the synthesis flow. De-elastisation [JMGE15] selectively removes elasticity from an elastic circuit, adopting a clocked protocol in selected parts without forcing synchronisation on the circuit as a whole results in a fine-grained GALS structure where regions may be run at different frequencies and the intervening elastic buffers may use synchronous handshakes within a region.

Figure 6.13 shows an abstract view of the De-elastisation flow applied at system level: a) a 1024-bit RSA key generator in form of macromodules which has a non-deterministic behaviour as its number of the encryption rounds depends on the input data, b) a fine-grained mid-level representation of the system generated using eTeak and c) selectively De-elastisised circuit based on the high-level patterns and low-level constraints where elastic buffers from (b) are converted to flip-flops selectively.

6.5.1 Patterns for Design Space Exploration

To explore the design space a set of experiments are coordinated with regard to the different computation patterns.

In this section De-elastisation is explored in a bottom-up approach at three levels of granularity: component-level, stage-level, and loop-level. Note that elasticity at gate and transistor level impose an area overhead of 4× and exhibit profound power drawbacks, and therefore are out of interest for this work.

The granularity of elasticity is illustrated in Figure 6.14(a) using an abstract dataflow example that comprises concurrency and data-dependent choice. This depicts a SELF-adopted dataflow of eTeak implemented in macromodule style with activation (go) and termination (done) signals. In this example control is implicit and propagates with data. Therefore, contrary to RTL models, a separate control unit does not necessarily exist. Initially the graph is extracted from a software-like, timing-free description and is fully elastic. Later a notion of time is introduced into the circuit by adoption of SELF. In this way data moves in separate time intervals that could be clock cycles. In this example it is assumed that the computation blocks take one clock
6.5. DE-ELASTISATION: A SYSTEM-LEVEL DESIGN FLOW

Figure 6.13: The De-elastisation flow: a system-level perspective.
Figure 6.14: Elasticity at different levels of granularity: a) fully elastic macromodule at component level; b) stage-level De-elastisation; c) loop-level De-elastisation (preserves elasticity only at boundaries); d) step-persistent bundling of concurrent computations (steps are depicted as shaded boxes).

cycle each to generate output and the primitives are all combinational.

Figure 6.14(b) shows the macromodule De-elastised at the level of individual stages/functions. The inner pipelines between Fork and corresponding Join are balanced (or rescheduled) by the insertion of a buffer, thus the associated elastic Fork and Join can be safely removed. Note that the input branches of Merge can have different arrival times which allows the loop to exploit a data-dependent behaviour. Removing the Merge and balancing the branches may boost the clock frequency, but could also degrade the shorter path that can deliver data one clock cycle earlier.

Figure 6.14(c) shows the result of De-elastisation at the level of algorithmic loop. The shaded Join and Fork components serve as data guards and are preserved to handle inter-loop elasticity. To maintain architectural non-determinism, the elasticity is kept at coarse-grained level. The proposed method exploits the distributed behaviour of local control in dataflows to apply De-elastisation within macro-modules. This enables local optimisation of the modules without affecting the timing discipline of the whole circuit.

To tackle the area overhead, Step Persistency [FKM+15] is leveraged to ‘bundle’ the concurrent computations, so that they can be scheduled to the same clock tick. This technique is proposed as a formalism for Petri net models of GALS systems where
a reachability graph of the model is extracted and pruned to establish maximally concurrent bundles. Step Persistency is employed in the dataflow circuits where data and control propagate together; thus identifying concurrent signals in both data and control paths is more straightforward. On the other hand, this idea fits very well with the De-elastisised circuits where pipelines are balanced by buffer insertion. Consequently the architectural modifications to the design contribute to an enhanced recognition of concurrent bundles and hence a reduction in area overhead. To the best of the author’s knowledge, this work is the first in demonstrating application of Step Persistency in a realistic benchmark. Figure 6.14(d) depicts a step-persistent dataflow where the concurrent computational blocks and their corresponding elastic controllers are bundled to reduce area overhead whilst preserving their advantages regarding performance and power.

Exploiting SELF at component level has power advantages since it offers clock gating over every register in the design. In addition, cell area can benefit from this protocol compared to asynchronous fully-interlocked protocols, however it still suffers from a 40% area overhead over a rigidly clocked synchronous circuit. For performance, regarding SELF controller’s latch-based structure, it can offer time (slack) borrowing which is supported by commercial EDA flows (see section 3.2.1). To take advantage of time borrowing, SELF needs to be applied at a coarse level of granularity to overcome its overhead. By using De-elastisation, the handshake components are removed from the design, then SELF is applied at register transfer level where its area overhead is tolerable.

Adopting SELF not only permits fine-grained elasticity to be preserved but also simplifies buffer insertion at the scheduling phase and helps to avoid deadlocks. Moreover, when allocating the computation blocks, commercial synchronous EDA can use the RTL library to optimise the circuit which improves the area by a factor of four [JMGTE14]. Regarding the Slack Elastic property [MM98], Teak circuits can be pipelined with any degree of storage on their communication channels. This property provides a flexible communication environment for the computational blocks. SELF does not violate this property and safely optimises the computational blocks without affecting the overall functionality. It was shown in Chapter 4 that fine-grained buffering the asynchronous dataflow can improve the overall throughput as it reduces the cycle
CHAPTER 6. DE-ELASTISATION: FROM ELASTICITY TO LOCAL RIGIDITY

Time (T_request + T_acknowledge); this buffering policy can be prohibitive for throughput in the synchronous domain as each buffer consumes one clock cycle [JMGTE14], especially in a processor architecture where data and control dependencies appear as a barrier to efficient pipelining.

6.5.2 De-elastisation vs. Synchronous Synthesis Flows

To evaluate the De-elastisation technique against the popular synchronous synthesis flows a good benchmark is required to which the dataflow De-elastisation and decomposition can be applied. The benchmark should realistic applications in real-life. It should include a variety of computing models, such as parallelism and concurrency, determinism and non-determinism. Finally the benchmark suite and the corresponding programs have to be obvious to display potential power and performance gains from De-elastisation.

6.5.3 Benchmarks

For this section a database acceleration benchmark [ONS+14], including some time-consuming operations, such as sorting, aggregation and search in the dataflow context, is selected. Since the purpose of this work is to compare state-of-the-art HLS flows against De-elastisised dataflows regarding their computation and communication models, these algorithms are selected to act as candidates for multi-threaded fine-grained, pipelined and producer/consumer computation models:

Bitonic sorter (fine-grained computation model)

The Bitonic sorter is the first element in this benchmark. This was introduced in section 6.4.3. Its throughput is governed by w, n, f parameters and because it is hardware friendly it achieves high-throughput of n significant cost in resources.

Spatial sorter (pipelined computation model)

The Spatial sorter consists of n sorting nodes each comprising a comparator, two registers and two multiplexers is shown in Figure 6.15. The asynchronous, synchronous elastic and De-elastisised implementations of the dataflow architecture of a 16-input 32-bit Spatial sorter is synthesised and experimented. The sorter inputs elements every
clock cycle. An input element goes either through the bypass route or the comparator within each node, which takes one or two cycles, respectively. This particular example can exhibit a data-dependent performance when implemented as dataflow. To preserve this feature De-elastisation is not applied inside the nodes as it balances the routes by inserting an extra buffer in the bypass route. To keep the execution time bounded by $n$ cycles as best-case (when the input stream is already sorted) and $2 \cdot n$ cycles as worst-case (when the input stream is sorted in reverse order) the synchronous elastic implementation is selected for the study. The sorting array also allows a parallel access to the sorted elements to be read in a single cycle. In contrast to Bitonic sorter, it requires $O(\log^2 n)$ times less resource.

**Median operator (aggregation computation model)**

This algorithm abstracts the data sets with their median values and is widely used in database queries specially in finance applications. To implement this operator, a sliding window, implemented as a shift register with parallel-read enabled is connected to a 16-input bitonic sorter which fetches elements in every cycle and outputs the mean of the 8th and 9th elements after $O(\log^2 n)$ cycles with a throughput of one element per cycle.

**Hash probe (producer/consumer computation model)**

The hash probe function is one of the popular methods leveraged by database search engines to look up the entries of a hash table in a much larger repository and build a new output table. The dataflow implementation of a hash probe comprises two loops, one used for probing the larger table (Figure 6.17: lines 1-15) and the other one is a 16-iteration LFSR-based function in this case, to map queries from an smaller table (T2) onto the large table (hTable) (Figure 6.17: lines 16-23). If the hash indices of two keys collide, the latter key is inserted in the next consecutive empty slot of hTable. If an empty slot is found, the current query is skipped.

### 6.5.4 Experimental setup

To measure area *Synopsys DesignCompiler* is used along with *DesignWare* library to synthesise the synchronous functional units (operations). For sorting algorithms, the
Figure 6.15: Teak-generated 16-stage Spatial Sorter dataflow with 130 Joins and Forks which all removable by De-elastisation. In this case data-dependent performance is obtained.
Figure 6.16: SELF adapted dataflow network of a Spatial sorter
1: **procedure** PROB(hashT2, result, tableAddress, tableRead)

2: ... 

3: hashT2 -> then 

4:   Stopped := 0 \parallel Index := hashT2.key; 

5: **loop while** (not Stopped) **then** 

6:   ReadTable (); 

7:   if (hashT2.key = hTableEntry.key) **then** 

8:     result <- #(hTableEntry.ptr) @ #(hashT2.rin) as OutRes 

9:     \parallel Stopped := 1 

10:   else 

11:     IncrementIndex() 

12:   **end** 

13: **end** 

14: **end procedure** 

15: **procedure** HASHFUNCTION(keyIn, Index) 

16: ... 

17: keyIn -> then 

18:   xBit := (((#keyIn[15] xor #keyIn[13]) 

19:     xor #keyIn[12]) xor #keyIn[10]) as bit 

20:   ; index <- #keyIn[14..0] @ #xBit as EntryT2 

21: **end** 

22: **end procedure** 

Figure 6.17: Probe loop and the 16-iteration LFSR-based function implemented as producer/consumer model using the Balsa language

input data is pairs of 32-bit keys and data values. The median operator takes 32-bit inputs. The hash probe uses pairs of 16-bit keys and 32-bit values, a hash table with 64K 48-bit buckets and a load factor of 0.6. The size of T2 is 400 MB and the size of hTable is 600 MB. In eTeak’s case, RTL code is produced for the datapath units, such as comparators and arithmetic units, so that the generated netlist can use Design Compiler library for the datapath components, as for the other three HLS flows. To measure the maximum achievable frequency, the target clock period was turned down in 0.05 ns steps until the synthesis fails to converge. To measure performance ModelSim SE 6.3a from Mentor Graphics is used for simulations. Experiments have been run on a Laptop with a 2.6 GHz Intel Core i5 processor and 8 GB of RAM.

### 6.5.5 Results

Figure 6.18 summarises the results of these experiments and Figure 6.19 proposes the associated geometric means. It presents area, performance, power, Lines of Code (LoC\textsuperscript{1}) as a productivity measure and compilation time obtained for each of the benchmarks

---

\textsuperscript{1}One of the possible future research directions is to leverage a hardware description sizing metric adapted to HDLs and HLS to avoid using LoC as a primary criterion.
which are synthesised using the flow presented in this work.

Hand-coded Verilog implementations are considered to represent a traditional RTL flow in these evaluations. To measure performance, the throughput metric is considered in a way that the number of cycles is counted in simulation and multiplied by critical path delay which is achieved by compiling, mapping and synthesising the generated RTLs using Synopsys Design Compiler. The same approach is leveraged to synthesise the eTeak netlist as its datapath components are behavioural Verilog modules, synthesisable by commercial EDA. For the other components eTeak’s technology mapping service is used.

For energy analysis two separate scenarios are considered to perform a fair comparison between the synchronous and elastic designs: a) when the pipeline is full which means all of the stages are kept busy every cycle and b) when a single stage is busy every cycle and the other stages are idle. Power measurements associated with these scenarios are averaged and reported. It should be noted that power consumption is normalised at 500 MHz to perform a fair comparison between the implementations. This way energy is measured independent of how fast the design runs.

6.5.6 Discussion

According to LoC which is one of the measurable factors of designer’s productivity, LegUp is ranked first (on the software extreme) whilst HDL-Verilog is last (on the hardware extreme). Between these are Balsa, BSV and Chisel, in that order. Comparatively speaking, the learning curve for the Balsa language is quite smooth compared to BSV and Chisel as it avoids reflecting any timing information to the designer.

Another measure for productivity could be the hardware details reflected to the designer. For instance, expecting the designer to define clock types and handle timing at higher abstraction level reduces the degree of productivity. Balsa exploits eTeak to virtualise timing such that clock is introduced automatically [MKG16] whilst BSV and Chisel need the designer to define clock in the code. LegUp produces single clocked hardware, however this work could be a possible approach toward timing exploration in LegUp as well.

Regarding the synthesis process, eTeak inherits predictability and transparency
Figure 6.18: Evaluation of the DBMS algorithms considering different computational models in terms of Area, Power, Throughput, Lines of Code (as a measure for productivity) and Compile/Synthesis time.
from its syntax-directed translation scheme whilst applies a set of peephole optimisations in further levels to generate the Verilog netlist. This light-weight mechanism accomplishes almost $2 \times$ faster synthesis and compilation time. This gives the tool a notion of data communication rates between the procedures, that can be used for binding high-level entities with the low-level physical characteristics automatically. Fast compilation mechanism paves the way for exploration tools like SynTunSys of IBM to take architectural refinements into account [ZLG+16].

To compare the tools regardless of the input descriptions, elasticity is considered at RTL. The tools associated with each benchmark are sorted based on the level of elasticity. According to the theoretical discussions in Chapter 2.2 on architectural differences, the hand-coded HDL-Verilog implementations is on the inelastic extreme whilst eTeak fine-grained elastic circuits are on the elastic extreme. Next to eTeak is Bluespec with its coarse-grained elasticity: every input/output channel in the design has \textit{enable} and \textit{ready} signals which can be interpreted as a handshake. Thus Bluespec design entities can exploit coarse-grained elasticity which enables data-dependent computation by activating rules to fire whenever data becomes available. Chisel and LegUp do not exploit elasticity and occupy the third and fourth places, respectively.

The experiments conducted confirm that in De-elastisised dataflows time borrowing contributes to about $2 \times$ speed-up in every case except hash probe. The latter has a non-uniform logic distribution so its speed-up is limited to 11%. Time borrowing, considered by the synchronous EDA, demonstrates maximum efficiency when
the combinatorial logic is distributed equally between the pipeline stages of a latch-based circuit. Without this feature throughput appears to be roughly close to that of Bluespec and Chisel.

Meanwhile, bundling the elastic controllers, as explained in the previous section, reduces the associated area overhead by 12% whilst preserving the other power and performance related features.

As in the dataflow model, data and control propagate together and get latched at the same time, switching off a latch not only prevents unnecessary activity in the datapath but also disables the corresponding activities in the control path. This feature improves power efficiency in the control-dominant context (hash probe example) by 24% compared to BSV implementation and by 11% compared to HDL-Verilog implementation. Circuits obtained by LegUp run slower, but exhibit better energy utilisation against the other inelastic implementations (i.e. Verilog and Chisel). The major part of the energy budget is attributed to the memory controllers that frequently perform memory read/write operations requested by the FSMs.

It should be noted that the hash probe example exhibits non-determinism influenced by the number of cycles taken to fetch data from memories and the loading factor which indicates the application statistics in the hash table. The results show that elastic dataflow models are more efficient in terms of power and performance compared to other HLS models considered in this study for handling non-determinism.

This experiment is conducted to demonstrate how the ‘intrinsic’ elasticity in the eTeak generated circuits influence power, performance and area compared to fully synchronous synthesis flows without exacerbation of the productivity factors. Of course, one can use third party techniques, such as de-synchronisation, to apply DVFS or clock gating onto the circuits generated by any of the synchronous compilers and results may be close to what we have reported here, but it would be different from our ‘self-contained’ tool experimentation strategy.

Compared to the Teak generated dataflows [JMGTE14] [Dua10] that leverage component-level elasticity, De-elastisised dataflows achieve 15% higher performance in loop-free architectures (bitonic sorter and median operator) where De-elastisation is applied at stage-level. This improvement is even more significant ($3 \times$) for the iterative architectures (spatial sorter and hash probe) where loop-level De-elastisation
is considered. Note that iterative dataflows can encounter deadlock without proper buffering. In the asynchronous domain, deadlocks can be avoided by buffer insertion (at least three per loop – see Chapter 4) which may impose prohibitive area overhead. In terms of power, on average, the asynchronous dual-rail version consumes 40% more power (at nominal voltage) compared to the synchronous counterparts due to the circuit size and switching activity of its fully-interlocked 4-phase protocol both in computation and communication.

6.6 Conclusion

This chapter proposed a novel high-level GALSification technique, called De-elastisation, that lowers the overhead of fine-grained elasticity in terms of performance, power and area by selectively removing handshake circuitry from the design. De-elastisation is evaluated with regards to different computation patterns at diverse levels of abstraction and granularity: In the asynchronous domain, De-elastisation shows 3 - 4× improvement in terms of performance, a significant reduction in area and about 40% better energy utilisation. In the synchronous domain, De-elastisation exhibits about 2× improvement in terms of performance, about 15% reduction in energy usage with a moderate area overhead. Moreover, to assess De-elastisation in the context of EDA flows, measures such as productivity and compilation time are also considered where the proposed technique shows comparable results versus a set of popular synchronous synthesis flows.
Chapter 7

Conclusions and Future Work

The rapid growth of demand for complex SoCs in the computing market raises two major questions: 1) Can current design methodologies sufficiently support this demand? 2) Are the current design methodologies efficient enough to supply the market with high-quality products?

This work shows that a high-level Globally Asynchronous Locally Synchronous (GALS) synthesis approach can address these questions by providing productivity for the designer and, meanwhile, augment general purpose CPUs with more efficient dedicated hardware blocks or application specific architectures. In this regard, a set of contributions has been proposed in the context of a framework, named eTeak, which enables high-level synthesis and exploration of synchronous elastic systems towards GALS design. This chapter gathers the conclusions derived from the previous chapters and discusses some research lines opened by this thesis as potential future work.

7.1 Summary of the Thesis

In this work a high-level synthesis framework, named eTeak, was presented which enables designers to automate the GALS design process independent of technology, protocol, data encoding or other details of circuit design. eTeak starts off from a high-level CSP-based asynchronous specification, introducing a common timing discipline by adopting the synchronous elastic protocol (SELF) for the asynchronous dataflows. The advantages and drawbacks of this transformation were discussed in Chapter 4.
To alleviate the overhead of elasticity, yet still adopting SELF, a transformation technique is derived to ‘partially’ re-synthesise dataflows into Synchronous Sequential Machines (SSMs). The method was described in Chapter 5 which followed the lines of De-synchronisation. Although the generated machines run faster than eTeak circuits, as they exploit synchronous rigidity to reduce the level of elasticity, they cannot leverage the available concurrency in dataflow models.

An alternative method was explored in Chapter 6, named De-elastisation. The proposed method transforms the clusters of fine-grained handshake components into rigid synchronous blocks automatically, to save area by removing redundancy and gain performance by eliminating handshake combinatorial components. The blocks are still linked by dataflows with elastic connections where a variable number of data tokens may be queued on the communications channels. The resulting network is reminiscent of macro-modules [SOC67] with go and done activation signals although they may be clocked internally. The ability to partition the elastic circuit automatically allows an exploration of many optimisation possibilities as the elasticity is reduced.

### 7.2 Contributions

The starting point of this thesis is based on the two novel works in high-level synthesis of asynchronous logic and Latency-Insensitive (LI) design in the synchronous domain. The work on synthesis exploits a distributed model without global control requirements which is suitable for applying local transformations and optimisations (e.g. re-synthesis) on the system whilst functionality remains unaffected. On the other hand, the latency-insensitive methodology allows synchronous intellectual property to communicate in a distributed fashion with a handshake protocol. Contrary to the asynchronous protocols, LI offers a discrete timing protocol based on tokens and bubbles which facilitates cycle-accurate analyses of the system in terms of energy and performance.

This work shows that a combination of these two paradigms, raising the level of abstraction to a ‘timeless’ description level, provides an effective means for exploring latency-insensitivity at different levels of granularity and abstraction. From this idea, an automated framework has been proposed to enable the designer to explore the
7.2. CONTRIBUTIONS

design space. In particular, this work has shown that it is possible to automatically build a GALS system starting from a timeless, software-like specification. The ability to condense parts of the system into synchronous blocks allows the removal of the internal asynchronous control overheads, resulting in faster and much smaller logic systems. These can be further optimised through mature EDA flows.

For GALS synthesis proper modelling and formalism is necessary. The CSP-based data-driven model, exploited in this work, not only avoids the state explosion of Petri Nets and the global control overhead of RTL, but also enjoys the distributed nature of the data-driven model where the locality of the control signals enable using effective partitioning methods. In a data-driven system control is realised by scheduling buffers. This largely simplifies the GALSification process and paves the way for further improvements.

The power of CSP in modelling non-determinism dominates the Kahn Process Networks, which are mostly used for studying real-time streaming applications and enables the proposed framework to be used for implementing complicated systems, such as general-purpose processors.

A partitioning method to define localities within a global elastic ecosystem is essential to enable automatic exploration of GALS designs. GALS realisation should not be restricted to the task of running several IPs within a system at their convenient clock speed. Based on the observations in this work it is learned that partitioning (or defining boundaries) is highly connected with the formal modelling of a system. Partitioning can be applied in three different ways with regard to the following: a) physical characteristics (e.g. critical path), b) architectural behaviour or high-level patterns which focus on the data exchange rates between computation blocks, and c) a hybrid of the previous two which is more effective. In this respect, the proposed De-Elastisation method connects the high-level patterns, such as pipeline, GPU-like or producer-consumer, to the circuit level constraints, such as clock speed.

7.2.1 E-Chart

The contributions of this thesis in the context of elasticity can be formulated in the form of a chart, named an E-chart, which explores elasticity at different levels of granularity and abstraction (figure 7.1). E-chart is an exploration framework upon
which the answer for determining the degree of elasticity can be pursued. To avoid complication numbers are not shown in this chart; the reader is referred to the previous chapters for further details.

The chart depicts four abstraction levels from a timeless high-level description (aka Functional Level) on the top, via cycle level (CL) and register transfer level (RTL), to gate-level (GL) on the bottom. The circuit level implementations are grouped into four major categories: Asynchronous, Synchronous Elastic, De-Elastic (GALS-like) and Synchronous. The curves at CL and RTL represent the level of elasticity incorporated into different implementations: control- and data-driven styles. For instance, asynchronous designs enjoy short cycles; they are perfectly suitable for producer-consumer realisations and exhibit better energy utilisation, almost 30% against their synchronous counterparts, in a non-deterministic data-dependent situation, whereas the synchronous model is proper for critically timed systems where the control has to be squeezed into clock cycles and elasticity may impose an overhead in terms of performance. Therefore de-synchronisation [CKLS06] of such systems has to be done with careful considerations.

Fine-grained elasticity could be expensive and may result in significant (up to
7.3. CONCLUDING REMARKS

4×) area overhead. Also, as feature sizes shrink, sub-threshold leakage poses low-power challenges so fine-grained elasticity may be detrimental here, too. A less radical approach is to exploit De-elastisation [JMGE15] which borrows rigidity from the synchronous domain to lower the overhead of elasticity. This could be done through either re-synthesis (see Chapter 5) or re-timing (see Chapter 6), whilst inheriting the powerful features of asynchrony, such as better energy usage (almost 20%) and ability to handle system-level non-determinism with 3–4× less performance overhead against fine-grained elastic designs.

7.3 Concluding Remarks

Running designs through the De-elastisation process revealed 4 – 4.5× area saving versus the asynchronous counterpart whilst performance shows significant improvement of 3 – 4× in iterative and producer-consumer architectures. Against synchronous circuits the performance improvement is around 2× by leveraging circuit-level time borrowing with a moderate area overhead around 10%

Regarding power, since the proposed approach relies on a clocked protocol it has 12 – 30% more power usage compared to its asynchronous counterpart. However, its data-dependent clock gating scheme makes it around 15% more power efficient compared to its synchronous implementation.

Judging the ‘ease’ of expression of a design is much more difficult to quantify, but regarding the lines of code as a measure Balsa is productive as it allows the designer to express concurrency in a scalable fashion through the channels and meanwhile does not reflect hardware details, such as timing, to the designer.

7.4 Directions for Future Research

I believe eTeak can be used to pursue several question in the EDA community from modelling to circuits. This section gives an overview of possible research lines that may be derived from the results of this thesis.
7.4.1 Energy-centric Synthesis of GALS Systems

eTeak, as an automatic synthesis framework, offers a flexible, productive and customisable design methodology towards low-power and high-performance heterogeneous System on Chips (SoCs) where general-purpose CPUs are replaced with more energy-efficient processing units. It is claimed that raising abstraction to the system level enables designers to gain productivity with almost \(10\times\) higher chance of improving power closure by architectural exploration [ZCDC15].

One can exploit the eTeak framework for a new direction in architectural synthesis to propose a model to pursue an answer to the following question: with given energy and performance objectives, what is the optimal GALS configuration (the clocking policy of the synchronous islands) to overcome communication delays and meet application deadlines? To answer this question, advanced partitioning methods based on De-Elastisation [JMGE15] and De-synchronization [CKLS06] need to be developed. Unlike the work proposed by Foroozan Nejad et al. [FHM+14] I believe it is important to formulate the problem subject to both high-level patterns and physical constraints to determine the optimal clocking speed for the associated partitions. In the long term, the associated framework can enable designers to explore large-scale ideas towards IoT where the constraints are mainly communication-oriented. A proper formulation in this respect will enable designers to explore complex computation models and communication protocols in the future.

7.4.2 eTeak as a Backend for the “Go” Language

‘Go’ is an open source programming language, developed by Google, that makes it easy to build simple, reliable and efficient software. Its first stable version has been released in 2015. Go (like Erlang) exploits Tony Hoare’s model of computation, Communicating Sequential Processes (CSP) [HR78], to implement concurrency. It has been agreed that the concurrency offered by CSP is more effective than the parallelism available in imperative languages such as C++ and Java through threading. Nowadays CSP is being employed by companies such as Twitter for scaling software and Unix for efficient operating system implementation.

The results in Chapter 6 suggest a CSP-based model proposes outstanding energy
utilisation over RTL models. This motivates us to think of adopting the same trend in the hardware domain where concurrency is more effective than parallelism in software. To provide evidence for the reader, a brief experiment has been conducted where a prime number generator (aka sieve of Eratosthenes) is implemented both in software using the gccgo compiler and hardware using the proposed framework in this thesis. On average the hardware implementation runs 90-120× faster than its software counterpart while the processor clock speed is almost the same as the hardware clock speed (1.2 GHz).

7.4.3 Parameterisable Synthesis and Simulation

High-level synthesis flows have emerged to provide the designer with a flexible environment where the abstract specification of a system can be parameterised. The Chisel [BVR+12] framework developed by UC Berkeley has emerged as a hardware construction language and tool that enables advanced hardware design using highly parameterised generators. This powerful feature enables the designer to explore different architectural possibilities. Balsa’s powerful language features can be used for this purpose. Moreover, eTeak offers a graphical environment for concurrent cycle-accurate simulation which can be exploited towards fast system-level exploration.

7.4.4 Synchronous Dataflow Synthesis onto multiple FPGAs

Hardware/Software Co-design of large-scale systems permits realisation of big ideas. FPGAs are the most commonly used platforms for this purpose and there are vendors who support designers targeting this technology as long as the design fits in a single FPGA. However, for large-scale systems the designer has to consider decomposing the design to run on multiple FPGAs. This task is error prone and refining the timing would be time consuming. To avoid this the design has to be elastic in the first place. The distributed nature of the dataflow synthesis through eTeak allows realisation of synchronous localities communicating via channels within a global elastic ecosystem consisting several FPGAs. I believe the enhanced productivity provided by the proposed infrastructure can be leveraged for realising large-scale ideas in Big Data computing in near future.
7.4.5 Heterogeneous Synthesis of the Balsa Language

The dataflow model, also known as the actor model or ‘reactive programming’, is a convenient way of implementing various types of systems and architectures. These models have been widely used for realising and verifying data-streaming applications in the past decade. However, to leverage dataflows in the context of control-dominant architectures complex extensions, such as data tagging, are required. As an example, for modelling an out-of-order processor or a complex data structure using dataflows tagging data tokens for processing or storage is necessary to ensure correct functionality. To avoid complicating the dataflow models it is appropriate to employ control-driven models. This suggests a heterogeneous modelling and synthesis scheme suitable for both control and data driven systems. The proposed concept in Chapter 5 leverages this method to show how integration of elastic dataflows and De-synchronised control-driven entities could be exploited toward heterogeneous SoC synthesis from a unified high-level language, Balsa. As future work, eTeak could be extended to incorporate De-synchronised IPs which would benefit from the SELF-adopted SoC ecosystem of eTeak.
Bibliography


[FNT+99] Robert M Fuhrer, Steven M Nowick, Michael Theobald, Niraj K Jha,


