De-elastisation: From asynchronous dataflows to synchronous circuits

DOI:
10.7873/DATE.2015.0759

De-Elastisation: From Asynchronous Dataflows to Synchronous Circuits

Mahdi Jelodari Mamaghani, Jim Garside, Doug Edwards
School of Computer Science, The University of Manchester
Manchester, UK M13 9PL
Email: mamagham@cs.man.ac.uk

***Abstract*—Asynchronous VLSI programming provides a flexible abstract formalism for concurrent systems but the is an issue for industrial adoption. The asynchronous design paradigm provides 'elasticity' enabling the system to tolerate delays in communication and computation but can impose a prohibitive communication overhead when applied at a fine-grained level. This paper proposes 'De-elastisation' in a CAD flow for asynchronous dataflow networks to improve the circuits' performance and area. To preserve the architectural advantages of asynchronous design (e.g. short cycles), circuits are classified into blocking and non-blocking loops which the De-elastisation scheme relies upon. The technique is incorporated in the Teak CAD flow. Experimental results on substantial case studies show significant performance and area improvements. This work shows 3× improvement for the first category of circuits, suitable for iterative realisations and DSP-like architectures and 4× for the second category which are suitable for concurrent realisations.**

I. Introduction

Asynchronous systems are gaining attention because they facilitate fine-grained pipelines and, consequently, shorter cycles yielding higher performance [1] [2], particularly in streaming data applications. It can be more 'natural' to specify dataflows without regard to implementation details (such as clocks) giving greater designer productivity. Such 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. This 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 [3].

Several asynchronous circuits synthesis systems exist. Some, such as Haste/TiDE [4] or Balsa [5] are control-driven; others, e.g. Teak [6] or the work of Taylor et al. [7] are data-driven for increased performance in dataflow systems. Here we target eTeak [8] which is an extension to Teak. It incorporates synchronous elasticity [9] as a common timing discipline to achieve a deterministic (cycle-accurate) behaviour.

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. By selectively removing *some* elasticity performance may be improved, considerable chip area may be saved yet the flexibility of operation can largely be maintained.

The method employed here selectively removes elasticity from an asynchronous description, adopting a clocked protocol without handshaking in selected parts of the circuit without forcing synchronisation on the circuit as a whole. This results in a GALS (Globally Asynchronous Locally Synchronous [10]) system; regions may be run by different clocks or the intervening elastic buffers may use synchronous handshakes.

In contrast to De-Synchronisation [2] which 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 contributes to system flexibility and allows a designer freedom to explore.

To show the effect, two general purpose processors, SSEM [11] and Sparkler, specified in a CSP-like language (Balsa), are synthesised using Teak, eTeak and our flows. Results illustrate the overheads associated with elasticity in terms of performance and area.

A. Elasticity

Elasticity emerged as a solution to uncertainty in computation and communication delays. The idea was introduced by Carloni et al. [12] at system level and was formalised by Carmona et al. [9] 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 and its applications have appeared for GALS at system level [13][14] and network-on-chip [15]. Elasticity is able to deal with any sort of non-determinism including cache delays [16], speculation in processors [17], etc. 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. We keep elasticity at the boundaries of these sort of structures to address their non-deterministic behaviour.

'De-elastisation' should be considered with some circuit level restrictions. In this regard, we have classified circuits into two categories: blocking and non-blocking loops [18]. Blocking loops have data access from the environment through channels that push data into the circuit; non-blocking loops are where data is requested and no stall is involved.

978-3-9815370-4-8/DATE15/©2015 EDAA

B. Contributions

The contributions of this paper are as follows:

1) *De-elastisation* is a method proposed to partially substitute the elastic protocol in a dataflow circuit with rigid clocked timing, considering the architectural restrictions.

2) Classifying systems based on their behaviour and providing the designer with information about a 'proper' design style which contributes to designer's efficient decision making at higher abstraction level.

3) The technique also proposes a comprehensive solution to cover data-dependent loops, decoupling them from the rest of the design and preserving elasticity at the boundaries.

II. Related Work

Various techniques have been proposed to improve asynchronous systems. For performance, works have addressed the *slack matching* [19] 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. [20] have proposed transforming such circuit 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. In a more practical approach, Martin et al. [21] 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. [22] and Najibi et al. [23] 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 [24][25] through iteratively tracing the signals in the circuit.

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 [26]. In this context, Tarazona [27] and Dimou et al. [28] 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 [29] [30].

Our Method: this inherits from slack matching and the resynthesis 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 clock frequency.

Our simple CAD flow transforms the design from an elastic dataflow to a synchronous solid structure in which elasticity is preserved occasionally based on the architecture classification.

The authors believe this work is the first to optimise the synchronous dataflow networks by considering circuit level bottlenecks. This work also contributes to behavioural partitioning of the system by classifying the architecture.

III. De-Elastisation

De-elastisation maintains the dataflow concurrency at the network layer whilst proposing a rescheduling and resynthesis method to reduce the level of communication between the state holding entities regarding physical timing characteristics.

A. Loops in Teak dataflow networks

To safeguard functionality we categorise loops into *blocking* and *non-blocking*. 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 [7], Teak provides pull channels (triggered by a *want* of data) to read data which significantly simplifies its storage structures. Loops which receive data through push channels emerge as blocking (figure 1) and the loops that pull data on-demand become non-blocking (figure 2).

These figures show a range of standard Teak flow control components: **F**ork and **J**oin components define concurrent data flows whereas **S**elect and **M**erge allow alternative choices.

B. Algorithm

The algorithm is implemented in Haskell and integrated in the Teak 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. It has been shown that for ensuring deadlock freedom in the asynchronous domain three latches per cycle (or 2N+1 for N tokens) are required [27] whilst in the synchronous elastic domain a single buffer (double latch) (per token) is enough [14]. In step 2 a Depth-First-Search (DFS) is required to group the edges into fore-, back- and cross-edges.

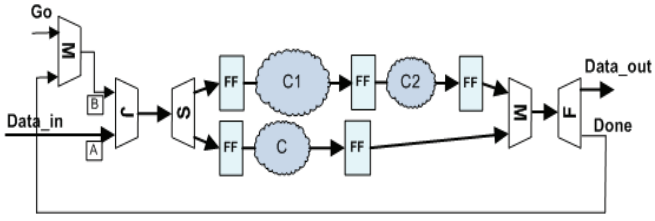

Fig. 1: Type 1: Elastic Join & Fork in a macro-module guards push channels (A) & (B). Join can receive data on either within $0 \leq t \leq n$; n denotes a bounded number of clock cycles as latency.

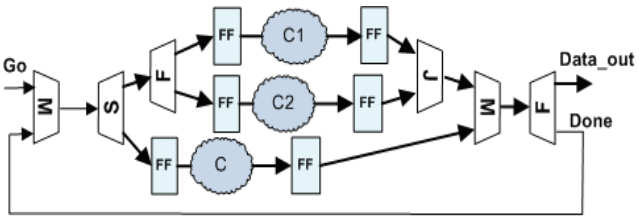

Fig. 2: Type 2: Join & Fork in a macro-module guard pull channels. Fork splits request & Join gathers data in a bounded number of cycles.

```
1: procedure SETREACHABILITYDEPTHS(network)
2:     let
3:     inpPorts                                   ▷ array of input ports
4:     inpComps          ▷ array of first adjacent comp. to each port
5:     visited ← ∅         ▷ (previously DSFed components, depth)
6:     stack ← ∅                                  ▷ stack for DFS
7:     finalList ← foldl (DFS stack) visited inpComps
8:
9:       DFS stack visited comp
10:        if comp is in stack return visited
11:        else push comp into stack and
12:           Update visited with comp
13:           foldl (DFS stack) visited nextComps
14:               nextComps ← AdjacentComps (comp)
15:
16:    return finalList▷ List of Joins and Merges with associated depths
17: end procedure
```

Fig. 3: First phase: assigns depth value to Joins and Merges in the network.

1) Remove the algorithmic back-edge
2) Mark the back/forward/cross edges in the graph
3) Buffer the back-edges to ensure deadlock freedom
4) Run the first phase: *SetReachabilityDepths*
5) Run the second phase: *DeployBuffers*
6) Remove Elastic Joins, Eager Forks and elastic buffers except the Joins that are hooked to push channels

1) Modify DFS to find reachability depths for components: The first phase (figure 3) 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. Regarding the acyclic nature of the search, it avoids cycles in the graph. The algorithm complexity is O(n*|v|) where n is the number of input ports and v is the number of components in the search space. For multi-input components {Join, Merge, Variable} there may be multiple visits by different DFS runs resulting in different depth values for a single component. The function *Update* (line 12) 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.

2) DP to deploy buffers over the branches: Having the multiple-input components annotated with their reachability depths in the first phase, we consider balancing the depths associated with the input links by deploying buffers (figure 4).

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 figs. 1, 2) or closer to Joins as they are usually in critical paths.

C. Potential Improvements

The combinational nature of the Elastic Joins, especially those with high fan-in, contributes overhead to the critical paths. By removing Elastic Joins, Eager forks [9] become

```
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:
9:         where BufferLinks links List
10:          ▷ List is generated by DP, it contains the required number of
   buffers for balancing the incoming links
11:    return
12: end procedure
```

Fig. 4: Second phase: balance branches by deploying buffers.

redundant as they were inserted to avoid establishing combinational loops with the joins. The non-elastic circuit has reduced area and boosted clock frequency due to the removal of these, plus the resynthesised combinational components.

IV. Evaluation Results and Analysis

To evaluate the proposed De-elastisation technique two case studies are exercised: (a) SSEM, an iterative processor and (b) Sparkler, a 3-stage pipelined processor. Results are compared using three different methods: 1) Async [27] – a deadlock detection algorithm for Teak; 2) SCP [31] – a critical path aware technique for buffer insertion to improve performance; 3) SE [14] – a technique to improve area and performance by incorporating the synchronous elastic protocol in Teak. Our method (Sync) reaches a higher performance whilst minimising the buffer count for area (and performance).

A. SSEM Iterative Processor

The Small-Scale Experimental Machine (SSEM) [11] is implemented in Balsa. This machine has three stages wrapped in a single algorithmic loop. Here it runs a GCD program with 30 iterations. De-elastisation is applied to the Teak-generated circuit and the results are compared with their asynchronous and synchronous elastic counterparts [14] (table I). According to our classification in Section III-A, SSEM uses pull channels (Type-2) to read data from memory. The memory has a deterministic latency so we can de-elastisise the circuit completely.

B. Sparkler Pipelined Processor

Sparkler is a cut-down version of the SPARC v8 architecture [32] implemented in Balsa. It also comprises the three stages of Fetch-Decode-Execute and is pipelined, thus falling into the first category (Type-1). Sparkler's pipeline stages individually contain algorithmic loops. This, in contrast to the SSEM, results in shorter cycles which contributes to a higher performance. A Dhrystone benchmark is used for simulation.

The major difference between the first pairs (Async and SCP) and the second pairs (SE and Sync) in table I 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 inserted in non-critical paths. For instance, the execution time for an over-buffered asynchronous Sparkler is 2.01 μs whilst it is 7.2 μs for the synchronous elastic version (at 1.7 GHz) which is almost 3.5$\times$ worse.

TABLE I: De-Elastisation algorithm applied over a set of general propose processors

Case Study	No. of Buffers	Clock rate (MHz)	Area (k $* \mu m^2$)	Exec. time (100 $* \mu$s)
SSEM-Async [27]	65	NA	56.183	46.5
SSEM-SCP [31]	195	NA	66.231	39.3
SSEM-SE [14]	6	485	12.563	62.4
SSEM-Sync	**15**	**1087**	**10.723**	**16.44**
SPARKLER-Async [27]	564	NA	472.270	234.5
SPARKLER-SCP [31]	1221	NA	574.288	194.1
SPARKLER-SE [14]	247	690	134.067	302.6
SPARKLER-Sync	**582**	**1540**	**212.550**	**65.64**

For the sake of consistency, all the designs use UMC 130 nm technology. To measure area, Verilog netlists are technology mapped using the Teak back-end, *Synopsys DesignCompiler* and the *DesignWare* library to synthesise the synchronous functional units (operations). To measure performance *Modelsim SE 6.3a* from *Mentor Graphics* is used. Our algorithm runs almost 10× faster than the SCP method [31] due to its light weight nature.

V. Conclusions and Future Work

In this work we presented De-elastisation as a novel approach for system design. It incorporates synchronous subcircuits in an asynchronous elastic flow to achieve higher performance while minimizing the area. Two processor examples with different implementation styles, iterative and pipelined, are presented and their associated structures are classified as a guideline for the designers in a dataflow context. Automatic partitioning the system into structural loops with different critical paths opens up a new synthesis scheme for High-level GALS synthesis. As a future work, we will focus on running de-elastisised parts with different clock frequencies in an elastic ecosystem.

VI. Acknowledgments

We like to thank Will Toms for his useful remarks. This work was supported by EPSRC Grant "Globally Asynchronous Elastic Logic Synthesis (GAELS)" (EP/I038306/1) and a full research scholarship from the School of Computer Science, The University of Manchester. eTeak is available at http://apt.cs.manchester.ac.uk/projects/tools/eTeak.

References

[1] W. Lee *et al.*, "Design of low energy, high performance synchronous and asynchronous 64-point FFT," in *Design, Automation Test in Europe Conference Exhibition, DATE*, 2013.

[2] J. Cortadella *et al.*, "Desynchronization: Synthesis of asynchronous circuits from synchronous specifications," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 25, no. 10, p. 2006, 2006.

[3] N. Jamadagni *et al.*, "An asynchronous divider implementation," in *Asynchronous Circuits and Systems (ASYNC), 2012 18th IEEE International Symposium on*, May 2012, pp. 97–104.

[4] S. F. Nielsen *et al.*, "A behavioral synthesis frontend to the haste/tide design flow," in *Asynchronous Circuits and Systems, ASYNC. 15th IEEE Symposium on*, 2009.

[5] D. A. Edwards *et al.*, "Balsa: An asynchronous hardware synthesis language," *The Computer Journal*, vol. 45, 2002.

[6] A. Bardsley *et al.*, "Teak: A token-flow implementation for the balsa language," in *Application of Concurrency to System Design, ACSD. Ninth International Conference on*, 2009.

[7] S. Taylor *et al.*, "Asynchronous data-driven circuit synthesis," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 18, no. 7, pp. 1093–1106, July 2010.

[8] M. Jelodari Mamaghani *et al.*, "eTeak: A data-driven synchronous elastic synthesiser," in *13th International Conference on Application of Concurrency to System Design, PhD Forum*, 2013.

[9] J. Carmona *et al.*, "Elastic circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2009.

[10] D. M. Chapiro, "Globally-asynchronous locally-synchronous systems," Ph.D. dissertation, Stanford University, 1984.

[11] S. Lavington, "A History of Manchester Computers (2nd ed.), Swindon: The British Computer Society," 1998.

[12] L. Carloni *et al.*, "Theory of latency-insensitive design," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Trans. on*, 2001.

[13] S. Das *et al.*, "A formal framework for interfacing mixed-timing systems," *Integration, the VLSI Journal*, vol. 46, pp. 255–264, 2013.

[14] M. Jelodari Mamaghani *et al.*, "Optimised synthesis of asynchronous elastic dataflows by leveraging clocked eda," in *Digital System Design (DSD), 2014 17th Euromicro Conference on*, Aug 2014, pp. 607–614.

[15] G. Michelogiannakis *et al.*, "Elastic buffer flow control for on-chip networks," *Computers, IEEE Trans. on*, vol. 62, pp. 295–309, 2013.

[16] G. Hoover *et al.*, "Synthesizing synchronous elastic flow networks," in *Design, Automation and Test in Europe (DATE)*, 2008, pp. 306–311.

[17] M. Galceran-Oms *et al.*, "Speculation in elastic systems," in *Design Automation Conference (DAC). 46th ACM/IEEE*, 2009, pp. 292–295.

[18] E. A. Lee *et al.*, "System design, modeling, and simulation using Ptolemy II," C. Ptolemaeus, Ed. Ptolemy.org, 2014.

[19] P. A. Beerel *et al.*, "Slack matching asynchronous designs," in *Asynchronous Circuits and Systems, 12th IEEE International Symposium on*, 2006, pp. 11–pp.

[20] P. Beerel *et al.*, "Performance analysis of asynchronous circuits using markov chains," ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2002, vol. 2549, pp. 313–343.

[21] A. Martin *et al.*, "The lutonium: a sub-nanojoule asynchronous 8051 microcontroller," in *Asynchronous Circuits and Systems, 2003. Proceedings. Ninth International Symposium on*, May 2003, pp. 14–23.

[22] G. Gill *et al.*, "Performance estimation and slack matching for pipelined asynchronous architectures with choice," in *2008 International Conference on Computer-Aided Design (ICCAD'08), November 10-13, 2008, San Jose, CA, USA*, 2008, pp. 449–456.

[23] M. Najibi *et al.*, "Slack matching mode-based asynchronous circuits for average-case performance," in *Proc. of the International Conference on Computer-Aided Design*. IEEE Press, 2013, pp. 219–225.

[24] E. Yahya *et al.*, "Statistical static timing analysis of conditional asynchronous circuits using model-based simulation," in *Asynchronous Circuits and Systems (ASYNC), 2013 IEEE 19th International Symposium on*, May 2013, pp. 67–74.

[25] G. Venkataramani *et al.*, "Leveraging protocol knowledge in slack matching," in *Computer-Aided Design, 2006. ICCAD '06. IEEE/ACM International Conference on*, Nov 2006, pp. 724–729.

[26] A. Alekseyev *et al.*, "Improved parallel composition of labelled petri nets," in *Application of Concurrency to System Design (ACSD), 2011 11th International Conference on*. IEEE, 2011, pp. 131–140.

[27] L. T. Duarte, "Performance-oriented syntax-directed synthesis of asynchronous circuits," Ph.D. Thesis, The University of Manchester, 2010.

[28] G. Dimou *et al.*, "Performance-driven clustering of asynchronous circuits," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 33, no. 2, pp. 197–209, Feb 2014.

[29] T. Chelcea *et al.*, "Resynthesis and peephole transformations for the optimization of large-scale asynchronous systems," in *In DAC*. ACM Press, 2002, pp. 405–410.

[30] L. Plana *et al.*, "Attacking control overhead to improve synthesised asynchronous circuit performance," in *Computer Design: VLSI in Computers and Processors, ICCD. Proceedings. IEEE International Conference on*, 2005.

[31] H. Ren *et al.*, "Critical path analysis in data-driven asynchronous pipelines," in *Computer Communication and Informatics (ICCCI), 2012 International Conference on*. IEEE, 2012, pp. 1–9.

[32] "Sparc v8 architecture," Website. [Online]. Available: http://www.gaisler.com/doc/sparcv8.pdf