2014 17th Euromicro Conference on Digital System Design

Optimised Synthesis of Asynchronous Elastic Dataflows by Leveraging Clocked EDA

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

***Abstract*—A 'natural' way of describing an algorithm is as a data flow. When synthesizing hardware a lot of design effort can be expended on details of mapping this into clock cycles. However there are several good reasons – not least the maturity of Electronic Design Automation (EDA) tools – for implementing circuits synchronously. This paper describes: a) an approach to transform an asynchronous dataflow network into a synchronous elastic implementation whilst retaining the characteristic, relatively free, flow of data. b) work to translate a synchronous elastic dataflow into a synchronous circuit whose deterministic properties pave the road for further behavioural analysis of the system. The results exhibit considerable benefit in terms of area over an asynchronous dataflow realisation.**

I. Introduction

The forward-looking trend in VLSI is System-on-Chip (SoC). As SoCs expand one of the major problems encountered is designer productivity. To address this, 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 'tokens' in a given place at any instant. The clockless nature 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.

On the other hand, synchronous circuits are efficient, well understood and well supported by modern Electronic Design Automation (EDA) tools. It is unlikely that this paradigm will be supplanted for most applications in the foreseeable future.

One approach to this problem which is gaining popularity is BlueSpec [1] which abstracts the synchronous design flow more than languages such as Verilog and VHDL. It exploits an *Atomic Rules and Interfaces* model to realise parallelism at a higher abstraction level. This model, in contrast to sequential programming models (such as C++) which leverage an extension to model parallelism, is fundamentally parallel. BlueSpec Verilog (BSV) uses *Atomic Transactional Memories* to handle communication between modules/rules/processes which falls in the line of a *Latency-Tolerant Shared-Memory* model [2].

A similar approach, predating this, and necessarily neglecting clock cycles has been taken in the *asynchronous logic* community with languages such as Balsa [3], Haste/TiDE (formerly Tangram) [4] and CAST/CHP [5]. These languages exploit a *Communicating Sequential Processes (CSP)* model to implement concurrency. Here, processes communicate through message passing which has to be realised with handshaking. These languages have suffered from drawbacks in performance – typically from the need for handshake communications when transferring data rather than relying on evaluating within a prescribed cycle time – and area overheads due to the provision of the extra handshake circuitry. The community has attempted to address performance issues by improving the control flow [6], [7], [8]. Recently, Teak [9] and Click [10] have been proposed as *dataflow* synthesis frameworks to tackle the control overhead of Balsa and Haste, respectively.

A less radical approach to alleviating clocking problems and reducing the handshake overhead is Globally Asynchronous, Locally Synchronous (GALS) [11] design, where synchronous 'islands' of logic run without synchronisation between them. The main problems of GALS design are the lack of methodology and tool support for partitioning the system. Existing GALS methods are somewhat ad hoc; only rudimentary design automation has been proposed [12] where the top-level hierarchy determines the boundaries of the synchronous islands.

This paper describes a method to automate GALS-like implementation at a high-level of abstraction, independent of technology, protocol, data encoding or other details of circuit design. This starts from a high-level asynchronous specification but first introduces a common timing discipline by transforming it into synchronous elastic dataflows. These are then clustered into blocks automatically, inside which the fine grained elasticity can be eliminated to gain significant area savings. The 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 [13] with *go* and *done* activation signals although modules could be clocked internally. The ability to partition the elastic circuit automatically allows an exploration of many optimisation possibilities as the elasticity is reduced.

Paper Organisation: Section II introduces our contribution in this work. Section III overviews the current problems with GALS design. Section IV describes the properties of Teak and Section V looks at them from a GALS perspective. Section VI draws the advantages that synchronous elastic dataflow has over its asynchronous realisation as the first contribution. Section VII explains our second contribution. Finally, Section VIII depicts the results and Section IX concludes this work.

II. Our approach

Our approach exploits the fine-grained data-flow concurrency inherent from the asynchronous design rather than just preserving the latency insensitivity. We aim at raising the design abstraction level from RTL to algorithmic level to provide the designer with a flexible implementation of concurrent hardware. At this level, system functionality is specified by data flows, apart from timing constraints. In general, raising the level of abstraction could have three major benefits:

1) The designer is able to specify the hardware in the form of concurrent data flows rather than thinking in a sequential manner and squeezing the tasks in time boundaries.

2) It provides traditional designers with an interface to cover their unfamiliarity with asynchronous techniques, protocols or data-encoding in circuit implementation.

3) A higher level abstraction allows flexible exploration of the design space based on formal models, e.g. Petri nets [14] where it is possible to consider different analyses and measurements.

Contributions:

a) Regarding the advantages of Teak, we exploit a synchronous elastic flow (aka SELF protocol [15]) to introduce a common timing discipline to the circuit which transforms it into a clocked, latency insensitive system. A latency-insensitive system is able to tolerate dynamic changes in the computation and communication delays. This feature enables us 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 concentrating only on its functionality and postpone timing complexity to when synthesis takes place. Moreover it enables the tool to exploit synchronous EDA for logic synthesis whilst preserving the fine-grained concurrency and the network properties. The details are discussed in section IV. Figure 1 (a) corresponds to the SELF-adapted components integrated in Teak and (b) refers to buffers that are inserted to ensure deadlock freedom in the dataflow context.

b) A transformation mechanism was derived to partially convert Teak Dataflow Networks (TDNs) into Synchronous Sequential State Machines (SSMs) [16] which reduces the level of elasticity by partially removing the inter-component communications (figure 1 (d)). The algorithm is described in section VII. Following that, a heuristic is being developed to automatically partition the network based on its functional behaviour using 'GALSification' techniques [17] (figure 1 (c)).

III. Obstacles with traditional GALS design

Three decades ago the GALS concept was introduced [11]. Since then academia and industry have attempted to exploit this concept in SoC design which can, potentially, benefit from its multi-clocked behaviour. Broadly speaking, all the efforts to exploit GALS fall in the lines of ad hoc design where synchronous blocks are glued together using trivial interfacing logic and synchronisers [18]. The tremendous challenge with the multi-clock design is the implementation of stable communication between clock islands. To combat this issue many techniques are proposed in the literature including pausible

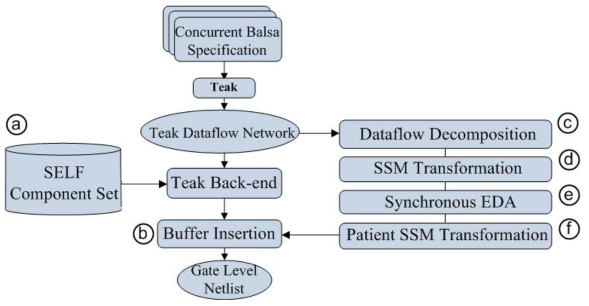


Fig. 1. Extended Teak synthesis flow

clocks, asynchronous and loosely synchronous interfacing etc. These techniques address the cross-timing issues through FIFO insertion which needs accurate consideration of timing details at circuit level and is survivable only by using the assemble-and-verify technique.

We believe that the time has arrived to automate the building of GALS systems without reflecting the extreme timing behaviour of the circuit to the designer. It is agreed that the synchronous approach with the associated accuracy complicates the design when it comes to SoC while the asynchronous approach simplifies the design aspects by separating timing from functionality [19]. Accordingly, we employ the asynchronous design approach along with advanced communication protocols to investigate the possibilities towards developing a new synthesis paradigm for GALS design. Section V explains our approach towards this objective.

IV. Teak: a data-flow synthesis system

We group the features of Teak 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 concurrency and synchronous message passing.

The networks are **slack elastic** [20] which means the communication channels are capable of storing a bounded number of tokens. This feature enables us to modify the level of pipelining over the channels without affecting the behaviour of the circuit.

From the computational perspective the network is built based on the **macro-module** style [13] with separate *go* and *done* activation signals. These modules are chained in sequence or parallel according to the source level directives. The macro-module architecture contributes to a distributed control mechanism where the datapath and the corresponding control are enclosed within a macro-module.

Accordingly, modules are controlled locally through handshaking so whenever data becomes available computation can start. This concept has already been introduced in **data-flow** systems [21]. Based on this concept data-dependent computation becomes possible which means that independent 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.

V. Teak towards *GALSification*

We explain how the features of Teak are exploitable towards automating the GALSification process and multi-clocked SoC design.

1) Point-to-Point Communication: Point-to-Point (PTP) communication enables a module to have independent rates of data streaming from different sources which contribute to a higher level of concurrency and accordingly effective throughput. Let's assume that module A with the input set {a,b,c,d} and the output set {x,y} is capable of performing two functions {f,g} which are not necessarily independent internally. The function f takes {a,b} as input and g takes {c,d}. Assume that input values are supplied with different bounded rates of a', b',c' and d' where a' is the slowest rate with one token/cycle and the rest with three tokens/cycle. Therefore g can operate and produce output independent from a' which results in higher throughout of module A. This technique is well-known in data-streaming and multimedia system design where it resolves the rate mismatch issue. The PTP communication is closely compatible with the data-flow computation style of the modules which is discussed later.

2) Slack Elasticity: A *Slack Elastic* system can be pipelined with any degree of storage on its communication channels. This behaviour was first formalised for the distributed computation systems which were described in a CSP-like language, CHP [20]. 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* benefit from elastic communication which is not available in the synchronous domain where rigid timing controls the communications. The elastic behaviour is preserved when the SELF protocol is employed. In section VIII we demonstrate a synchronous elastic processor which follows this concept.

3) Macromodule Style: The *Macromodule logic* was introduced to enable designers to implement complex circuits using simple data processing building blocks [13]. Later, this concept was used to simplify the asynchronous control design [22]. Teak employs this technique to perform control interactions locally instead of using a separate central control unit which has significant performance implications. We exploit distributed control behaviour to perform functional composition and decomposition of Macromodules which results in defining new boundaries within the network. Moreover, the Macromodules allow us to optimise one module without affecting the behaviour of the network.

4) 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 [21]. In Teak networks dataflow architecture joins with PTP communication, realises concurrency and eases the modules' decomposition process. Decomposing the modules towards *GALSification* based on their functionality, rather than structural properties, is our main objective.

Teak extracts parallel entities from the high-level Balsa code, produces a Control-Data Flow Graph (CDFG) and then maps it onto Macromodules with local control handshaking through the go-done channels. Therefore the resulting circuit benefits from a distributed control scheme. This feature allows us to explore 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.

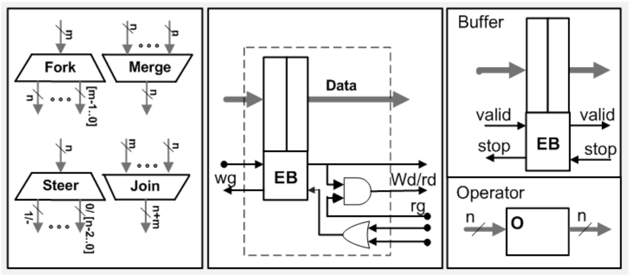


Fig. 2. SELF adapted Teak component set

VI. Synchronous Elastic Teak Dataflow Networks

From the above mentioned features of Teak, it was considered as a desirable framework to investigate high-performance synchronous elasticity from a high-level perspective. To incorporate synchrony in Teak, the existing component library was 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 [23]. An asynchronous buffer stage comprises a transparent latch with its associated controller whereas a synchronous buffer will be an edge triggered register. The component set is depicted in Figure 2 and the associated functionality of each is described in section VII. Since the new component set does not imply any combinational feedback loops within the network, conventional EDA can be used for optimisation purposes. Moreover, SELF is beneficial for the computation blocks as it simplifies the deadlock issue with loops according to its simple interlocking behaviour.

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 comparatively deterministic. The following are the advantages these machines have over their asynchronous counterparts. The corresponding results are proposed in section VIII for a case study.

1) *Simplified loop structures regarding correctness:* In an asynchronous loop each cycle must have always enough buffering for a lead token to move forward and leave space for the following token. Consequently, at least three latches are required in a cycle to ensure correctness. In a SELF adapted loop one master-slave register is enough as tokens' movements are synchronised with the clock. Figure 3 depicts a simple loop structure in Teak to realise iterative operations.

2) *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, time-decoupling

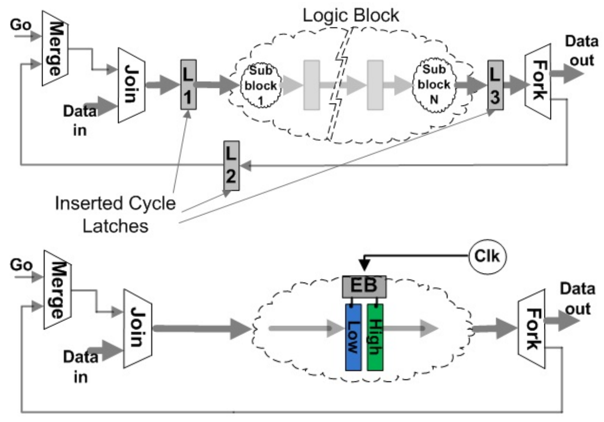


Fig. 3. Top Structure: an An asynchronous loop (ring) needs at least 3 latches to ensure correctness. Bottom Structure: a SELF adapted loop needs only one Elastic Block with a pair of latches to work properly

controllers in SELF employ clock-gating which reduces the switching activity of the storage elements in the datapath.

3) *Datapath improvement using a synchronous EDA library:* Data manipulation components generated by Teak use QDI encoding for computation which is necessary as the environment uses the same encoding for communication. By adapting SELF, exploiting synchrony in the computation units becomes possible as the communication fabric follows the same timing behaviour. This concept triggers the idea proposed in section VII.

4) *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 [22] which assume bounded delay for signal transfer. SELF employs bundled-data encoding for the clocked circuits which removes the necessity for considering additional timing assumptions. Therefore, our SELF adapted dataflows are four times smaller than Teak circuits.

VII. Transforming TDN to RTL interfacing with synchronous EDA

Introducing synchronous locality to the network for removing the unnecessary communication overhead is the motivation for this section. Figure 4 shows the cycle-accurate switching activity of the links in a SELF-adapted processor. It is obvious that the communication overhead of the system due to fine-grained handshaking could be prohibitive in terms of power and performance. The triangle symbols in the figure belong to the *Retry* state of SELF in which the associated link is stopped and is not allowed to transfer data. In an ideal system, for the sake of performance, Retry should not happen at all. To tackle this issue we need to insert several buffers to balance the pipelines in the system and avoid Retry-ing. Moreover, due to the fine-grained combinational communication between components, several glitches trigger in each link before they stabilise and clear for the next cycle. Although the same as any synchronous system, the glitches get resolved as the next clock edge comes up, they would be potentially influential on dynamic power. Regarding these problems, we believe that moving towards coarser granularity will be beneficial for synchronous elastic TDNs.

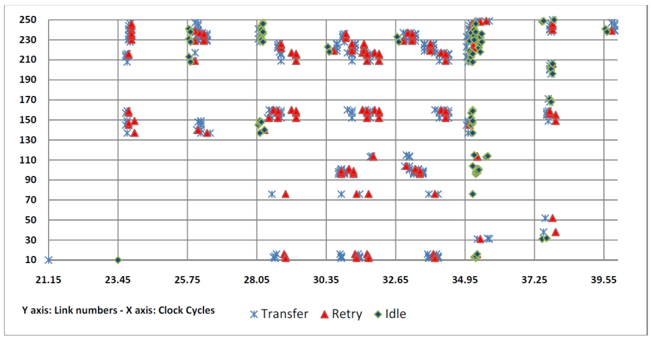


Fig. 4. The communication states associated with each link in a synchronous elastic dataflow processor is monitored. Due to the fine-grained combinational nature of the network, several glitches occur within each clock cycle.

To alleviate the handshake overhead, we introduce synchrony to the network to let synchronous EDA re-synthesise the system locally to reduce the fine-grained communication overhead whilst the environment enjoys the inherent properties of asynchrony. In this work we propose an RTL transformer which acts as an interface with synchronous EDA. The following explains the transformation algorithm and section VII-D discusses the details of binding a synthesised synchronous machine into the Teak dataflow network.

A. The Algorithm

Our approach performs state-space exploration of the network which is essentially the same as executing the graph. We extract all possible data manipulation scenarios from the graph by traversing the control flow. Our technique takes out the operations and squeezes them into clocked states where dataflow is implicitly governed by clock and data assignments, explicitly done by evaluating the expressions and moving the value into the variables.

The algorithm, shown in Figure 5, starts traversing the 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, *addToStateGraph()* does processing and inserting them into the *state graph*. After traversing the network recursively by *DFS()*, lines 9-17, the *runFSM()* function in line 2 outputs the detected states with associated data expressions and variable assignments considering the order of the states.

Definition 1. *[State Graph] is a directed graph denoted by a triple SG = (S,C,L), 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 holds the number of the states in the graph and L is the label given to each graph associated with the procedure label in the Teak network.*

Definition 2. *[Execution Scenario] is a sequence of components that the search function extracts from a source to a sink in a macromodule. A source/sink can be either a component or an input/output port.*

```
 1: procedure ExtractFSM($part$)
 2:     runFSM . DFS ($visited, comp$)
 3:     where
 4:     $inputComps$ = GetInputComps($part$)
 5:     $outputComps$ = GetOutputComps($part$)
 6:     $visited \leftarrow \emptyset$
 7:     $comp \leftarrow inputComps$
 8:
 9:     DFS ($visited, comp$)
10:     if CheckVisited($visited, comp$) then
11:         return $visited$
12:     else
13:         $nextComps$ = GetNextComps($part, comp$)
14:         addToStateGraph . visitComps ($nextComps$)
15:         $visited'$ =foldl markJoin($part$) $visited\ comps$
16:         foldl DFS $visited'\ comps$
17:     end if
18: end procedure
```

Fig. 5. Teak Dataflow Network to RTL Transformation algorithm

A linear system (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. Due to presence of Steer (Choice) and Fork in the network the control flow is not linear. Therefore, the extracted scenarios for a non-linear network might encounter state explosion. To avoid this problem we choose target sub-networks with a limited number of components for the transformation (figure 6).

We next explain the RTL implementation of some Teak primitives. It should be noticed that the RTL transformer does not preserve the handshake property of the primitives. In this regard, to ensure the correct functionality additional buffering would be required.

1) 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.

2) Fork: introduces concurrency to the circuit which: a) activates two or more macro-modules at the same time or b) supplies them with data. Therefore it is a parametrisable component capable of carrying any number of bits from input to outputs. In case (a) as long as macro-modules are independent they will function in parallel. The example in Figure 7 shows blocks A and B activated by a control Fork. However, due to data dependency of B on A, the control Fork is redundant and our method will consider them in sequence. The *markJoin* function in figure 5 at line 15 checks for this sort of structure in the design. In case (b), Fork gets translated to data assignments from input link to the output links.

3) Merge: multiplex 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 into account 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.

4) Join: synchronises data of arriving inputs. A two-way join of n and 0 bits can be used as 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 at least a link is a 0 bit signal, it implies control dependency which means that other data links should wait for a task to complete. *ExtractFSM()* considers this join after making sure that the associated scenario with the 0 bit signal is already extracted. *markJoin()* function takes care of it. This kind of Join acts as a sequencer [3] in Balsa. Later, buffering will be required to ensure that data arrive to the join at the same time.

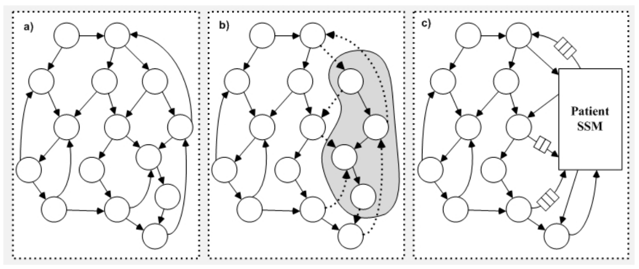


Fig. 6. a) A typical TDN graph b) The grey area is a random cut consisting of an arbitrary number of SELF adapted components c) The selected part is transformed into a SSM with additional buffering at its input and output arcs to resolve hazards

5) 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 allows us to distinguish 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 wire. Similarly, a write to a variable is translated as assigning the current value of the input wire to the register.

A Pseudo-code of our approach is shown in Figure 5.

B. An example: Sparkler Shifter Unit

To test our method several small-scale designs have been exercised. The shifter example is a pipeline-like structure that allows us explain the flow clearly. It is a two-stage 32-bit shifter unit from Sparkler processor which is a cut-off version of SPARC v8 architecture [24] implemented in Balsa. The shifter is capable of shifting at most two bits per iteration. For the 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 Figure 8 and the Teak

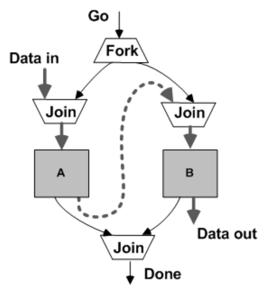
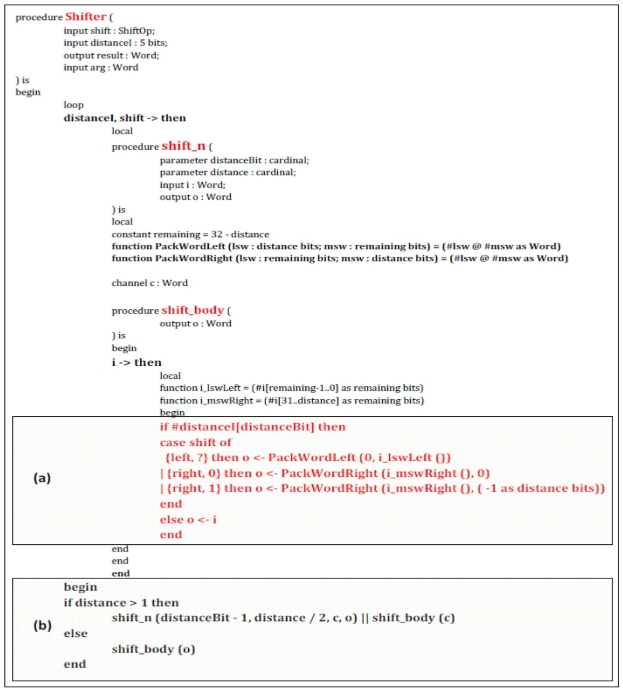


Fig. 7. A macromodule consist of two blocks. The control Fork initiates both to realise parallelism which is inherent in specification. The dotted link represents data dependency between A and B.



```
procedure Shifter (
          input shift : ShiftOp;
          input distanceI : 5 bits;
          output result : Word;
          input arg : Word
) is
begin
          loop
          distanceI, 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
                    i -> then
                              local
                              function i_lswLeft = (#i[remaining-1..0] as remaining bits)
                              function i_mswRight = (#i[31..distance] as remaining bits)
                              begin
                              if #distanceI[distanceBit] then
                              case shift of
                               {left, ?} then o <- PackWordLeft (0, i_lswLeft ())
 (a)                          | {right, 0} then o <- PackWordRight (i_mswRight (), 0)
                              | {right, 1} then o <- PackWordRight (i_mswRight (), ( -1 as distance bits))
                              end
                              else o <- i
                              end
                    end
                    end
                    end
          begin
          if distance > 1 then
                    shift_n (distanceBit - 1, distance / 2, c, o) || shift_body (c)
 (b)      else
                    shift_body (o)
          end
```

Fig. 8. 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

implementation of this unit using Teak primitives is shown in Figure 9. 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 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 them and creates a separate stage for each in the form of macromodules, MM1 and MM2. The *distance-shift* variable captures the input values from distance and shift input ports upon which the number of bits for shifting the data at each stage gets decided.

C. RTL for the shifter example

The RTL code generated for the shifter example is depicted in Figure 10. The code is in the form of an FSM in which the state flow is extracted from the CDFG of the shifter unit. By considering the graph as a macromodule with separate go and done (implicit) signals, it is possible to analyse the behaviour in terms of the rates associated with reads/writes from/to output/input FIFOs.

The grey route in Figure 9 shows a possible scenario in the dataflow whose associated states 1-19 are depicted in Figure 10. The wide coloured links are for data and narrow black ones are for control tokens. After receiving a *go*, the data provided by shifter,distanceI input ports flows into the network and gets captured by the *distanceI-shift* variable (C3). Thereafter, a read-after-write (RAW) link allows the *arg* port

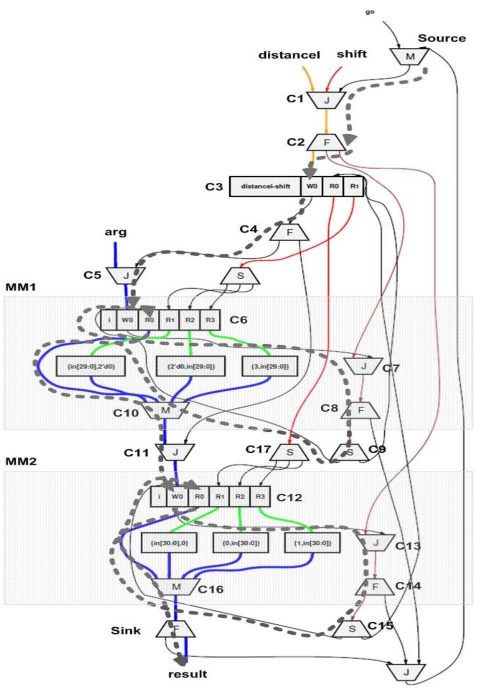


Fig. 9. Sparkler shifter graph with two stages of data manipulation which is simplified to depict the control and the dataflow separately.

inject its data into the circuit which gets stored in *i* variable (C6). Then, the associated RAW link becomes active which informs the control flow (C7) that the data is available and processing data can start.

In this example, the RAW link from the *distanceI-shift* variable (C3) gets distributed by a control Fork (C4) which means that more than one macromodule (MM1 and MM2) is allowed to start computation by reading data from this variable which implies concurrency. Due to a data dependency between MM1 and MM2 it is not possible to implement them as concurrent FSMs. Therefore the algorithm ignores (C4) and puts the corresponding states with each macromdule into the state graph in sequential order. In Figure 10 states 7-11 are for MM1 and 12-19 are connected with MM2.

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).

The algorithm described in this paper is implemented in Haskell and incorporated in the Teak flow. The software is able to partially transform TDN to FSM. For this example it extracts 16 execution scenarios with 56 states. Figure 10 shows the output for this process.

D. Synchronous Sequential Machine (SSM) Transformation

So far we explained a method to transform macromodules in a TDN into an intermediate HDL representation which is in FSM format extracted from CDFG with enclosed expressions within the states to realise the datapath components.

```
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

always @ (posedge clk) begin : FSM_SEQ

   if (reset == 1'b1) begin
     state <=  #1  1;
   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: next_state = 5;
 5: L72 = {arg}; next_state = 6;
 6: i1 = L72; next_state = 7;
 7: L187 = {L151}; next_state = 8;
 8: L188 = L187[0:0]; next_state = 9;
 9: case (L188): 0: next_state = 10; 1: next_state = 20;
 10: L78 = i1; next_state = 11;
 11: L87 = L78; next_state = 12;
 12: L35 = {L87}; next_state = 13;
 13: i2 = L35; next_state = 14;
 14: L184 = {L147}; next_state = 15;
 15: L185 = L184[0:0]; next_state = 16;
 16: case (L185): 0: next_state = 17; 1: next_state = 34;
 17: L105 = i2; next_state = 18;
 18: L112 = L105; next_state = 19;
 19: result = L112[31:0]; next_state = 1;
 20: L174 = distanceI-shift; next_state = 21;
 21: case (L174): 3: next_state = 22; 0: next_state = 26; 1: next_state = 30;
 22: L138 = i2[30:0];
 23: temp = 1; L111 = {L138[30:0],temp[0:0]}; next_state = 24;
 24: L112 = L111; next_state = 25;
 25: result = L112[31:0]; next_state = 1;
 26: L136 = i2[30:0];
 27: temp = 0; L107 = {temp[0:0],L136[30:0]}; next_state = 28;
 28: L112 = L107; next_state = 29;
 29: result = L112[31:0]; next_state = 1;
 30: L137 = i2[30:0];
 31: temp = 0; L109 = {L137[30:0],temp[0:0]}; next_state = 32;
 32: L112 = L109; next_state = 33;
 33: result = L112[31:0]; next_state = 1;
 //...
 default : $display("ERROR in FSM_COMB");
endcase
end
endmodule
```

Fig. 10. The RTL output for the shifter example

Considering this method, it is possible to use synchronous EDA to synthesise and optimise TDNs partially. EDA takes the extracted FSM and synthesises it into an SSM.

Definition 3. *An 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 [16]. Synchronous EDA is able to synthesise SSMs from behavioural or structural HDL specifications.*

After generating the SSM for the corresponding partition, the machine is transformed to a patient system [16] whose clock is controlled by a synchronous elastic block [15].

Definition 4. A patient *SSM is a latency-insensitive machine [25] 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 [16].*

According to the slack elastic property of TDN, buffering the input and output ports for any degree does not affect the functionality. We have used Synopsis' Design Compiler to synthesise the datapath elements of our case study [26] and transform them into the patient systems. The corresponding results are shown in the next section. There is no reason why this method cannot be applied to more complicated synchronous machines. Figure 6(c) shows an abstract view of the hierarchical transformation where a random cut of a TDN is implemented as a SSM and inserted into the graph.

VIII. Results

This section shows the results for the benefits achieved through the extended Teak flow, discussed in VI.

A. Synchronous Elastic Sparkler Shifter

Basically, the shifter example discussed in section VII-B consists of two pipeline stages which are wrapped in a single feedback loop (aka algorithmic loop). To ensure deadlock freedom (see section VI), the async. version of this 32-bit shifter needs a 3-stage latch whilst the sync. version running at 550MHz works with only one elastic buffer whose overall latency 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 async. protocol in the context of the pipelined architectures.

B. Synchronous Elastic SSEM Processor

As another case study the Manchester Small-Scale Experimental Machine (SSEM) is exercised [26]. The high level specification of this computer is developed in Balsa and has been synthesised onto hardware using the Balsa synthesis system [27]. This machine comprises three 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 11 the area cost associated with truly asynchronous and synchronous elastic design styles is depicted. The results demonstrate that our flow achieves a substantial impact on area: up to 4.5x improvement.

This experiment also confirmed that SELF preserves slack elasticity which is the key property for our further investigations. The first pair of columns in Figure 12 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 in this experiment. According to the results, although the fully buffered synchronous SSEM has shorter critical path delay, its throughput is 3.7x 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 ($T_{request}$ + $T_{acknowledge}$), but this buffering policy can be extremely prohibitive for throughput in the synchronous domain as each buffer consumes one clock cycle. Especially in a processor architecture where data and control dependencies prevent efficient pipelining.

As discussed in section VI, SELF simplifies the loop structures in the dataflow network and allows the use of synchronous CAD tools to optimise the circuit, particularly computation-heavy data manipulation units and detects the combinational loops for the sake of deadlock freedom. The second pair of columns in figure 12 demonstrate results with a necessary amount of buffering to ensure deadlock freedom. In this experiment the asynchronous SSEM has 65 buffers based on an algorithm for deadlock detection [28], whilst the synchronous elastic version has only 6 buffers.

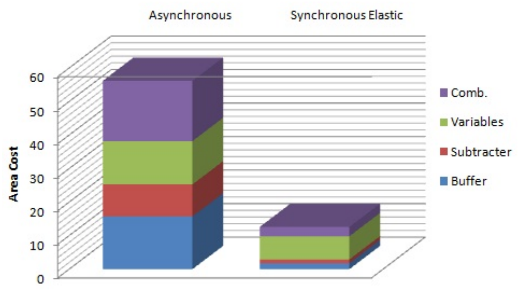
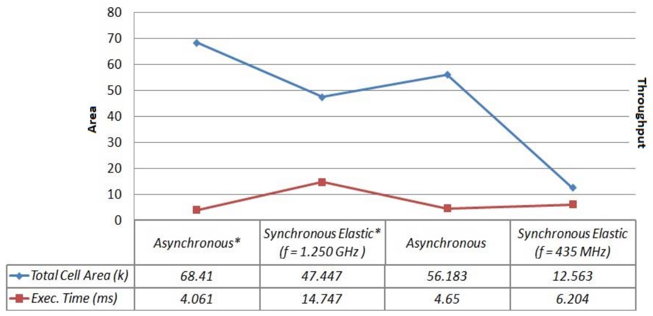


Fig. 11. The asynchronous dual-rail SSEM vs. its synchronous elastic counterpart in terms of cell area (UMC 130nm technology). Each column is fragmented based on the existing entities in the circuit. Buffers are used to remove deadlock.



	Asynchronous*	Synchronous Elastic* (f = 1.250 GHz)	Asynchronous	Synchronous Elastic (f = 435 MHz)
Total Cell Area (k)	68.41	47.447	56.183	12.563
Exec. Time (ms)	4.061	14.747	4.65	6.204

Fig. 12. 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 4.5x area improvement.

Although our approach results in a significant reduction in area, there are costs associated with this improvement. In terms of throughput, the synchronous elastic SSEM degrades by a factor of 1.3x. To tackle this, further slack matching [20] is required to balance pipelines towards performance improvement which is under development.

IX. Conclusion and Future work

This work successfully proposes an extension to Teak flow enabling it to synthesise synchronous elastic dataflow from fully asynchronous concurrent specification. This approach not only preserves the properties of asynchronous dataflows, but also it allows exploitations to be performed using mature CAD tools for further transformations in the synchronous domain. Based on the analysis, elasticity at component level suffers from prohibitive costs in terms of performance as communication overhead dominates computation. A reasonable alternative is to replace these with Synchronous Sequential Machines (SSMs) in which rigid timing removes elasticity. Accordingly, this work also presents an approach towards *de-elasticising* handshake based macromodules using synchronous EDA. As future work, we will focus on composing/grouping the macromodules based on their timing behaviour to form clocked islands.

X. Acknowledgment

This work is supported by EPSRC Grant "Globally Asynchronous Elastic Logic Synthesis (GAELS)" (EP/I038306/1).

References

[1] R. S. Nikhil, "BlueSpec: A general-purpose approach to high-level synthesis based on parallel atomic transactions," in *High-Level Synthesis*. Springer, 2008, pp. 129–146.

[2] J. D. Kubiatowicz, "Integrated shared-memory and message-passing communication in the alewife multiprocessor," Ph.D. dissertation, Massachusetts Institute of Technology, 1997.

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

[4] Handshake solutions, a philips subsidiary. [Online]. Available: http://www.handshakesolutions.com/

[5] A. J. Martin *et al.*, "CAST: Caltech asynchronous synthesis tools," in *Asynchronous Circuit Design Working Group Workshop, Turku, Finland*, 2004.

[6] T. Yoneda *et al.*, "High level synthesis of timed asynchronous circuits," in *Asynchronous Circuits and Systems, ASYNC. Proceedings. 11th IEEE International Symposium on*, 2005.

[7] 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.

[8] T. Chelcea *et al.*, "Balsa-cube: an optimising back-end for the balsa synthesis system," in *14th UK Asynchronous Forum*, 2003.

[9] 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.

[10] A. Peeters *et al.*, "Click elements: An implementation style for data-driven compilation," in *Asynchronous Circuits and Systems (ASYNC), 2010 IEEE Symposium on*, 2010.

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

[12] A. Hemani *et al.*, "Lowering power consumption in clock by using globally asynchronous locally synchronous design style," in *Design Automation Conference, 1999. Proceedings. 36th*, 1999.

[13] M. J. Stucki *et al.*, "Logical design of macromodules," in *Proceedings of the Joint Computer Conference*. ACM, 1967.

[14] C. A. Petri, "Kommunikation mit automaten," Ph.D. dissertation, Hamburg University, 1962.

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

[16] M. Vijayaraghavan *et al.*, "Bounded dataflow networks and latency-insensitive circuits," in *Proceedings of the 7th IEEE/ACM International Conference on Formal Methods and Models for Codesign*, ser. MEMOCODE, 2009.

[17] D. Sokolov *et al.*, "GALS partitioning by behavioural decoupling expressed in Petri nets," in *Asynchronous Circuits and Systems (ASYNC),IEEE Symposium on*, 2014.

[18] P. Teehan *et al.*, "A survey and taxonomy of GALS design styles," *Design Test of Computers, IEEE*, 2007.

[19] K. S. Stevens *et al.*, "The future of formal methods and GALS design," *Electronic Notes in Theoretical Computer Science*, 2009.

[20] R. Manohar *et al.*, "Slack elasticity in concurrent computing," in *Proceedings of the Fourth International Conference on the Mathematics of Program Construction*. Springer-Verlag, 1998.

[21] Arvind *et al.*, "Annual review of computer science." Annual Reviews Inc., 1986, ch. Dataflow Architectures.

[22] J. Cortadella *et al.*, *Logic Synthesis for Asynchronous Controllers and Interfaces*. Springer, 2002.

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

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

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

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

[27] A. Bardsley, "Balsa: An asynchronous circuit synthesis system," Master's thesis, The University of Manchester, 1998.

[28] L. T. Duarte, "Performance-oriented Syntax-directed Synthesis Of Asynchronous Circuits," 2010.