Optimising Bundled-Data Balsa Circuits

Norman Kluge
Hasso-Plattner-Institut
University of Potsdam, Germany
norman.kluge@hpi.de

Ralf Wollowski
Hasso-Plattner-Institut
University of Potsdam, Germany
ralf.wollowski@hpi.de

***Abstract*—Balsa provides a design flow where asynchronous circuits are created from high-level specifications, but the syntax-driven translation often results in performance overhead. To improve this, we exploit the fact that bundled-data circuits can be divided into data and control path. Hence, tailored optimisation techniques can be applied to both paths separately. For control path optimisation, STG-based resynthesis has been introduced (applying logic minimisation). However, solid results are missing so far due to problems with state explosion and the reliable insertion of reset logic. To tackle this, we use an adjusted STG decomposition algorithm and started to develop a new logic synthesizer (based on ideas of petrify) with proper reset insertion. Adding the adapted data path, we are now able to get first promising post synthesis simulation results using an industrial technology library (with a performance improvement of up to 23%). First experiments show additional potential for performance improvements (of up to 56%) when standard tools for synchronous design are applied to the data path.**

I. Introduction

BALSA [1] provides a design flow, where asynchronous circuits are created from high-level specifications. The BALSA COMPILER converts a high-level BALSA program into a network of handshake (HS-)components. These basic building blocks form the circuit and communicate via asynchronous handshakes. However the syntax-driven translation used by the BALSA COMPILER often results in performance overhead. To improve the performance, one can utilise the fact that bundled data circuits can be divided into data and control path. Hence, tailored optimisation techniques can be applied to both paths separately. For control path optimisations, Signal Transition Graph (STG) based (re)synthesis has been introduced e.g. in [2]–[5], applying logic minimisation methods to (parts of) the control. In [2] a SpecC specification is transformed into tailored Balsa code, from which STGs are generated directly using the early acknowledge HS-protocol. The other approaches are working with a (Breeze) netlist of HS-components communicating via the original HS-protocol (resynthesis): In [3], only HS-components modelling pure control were resynthesised. In [4] and [5], mixed HS-components (consisting of control and data path) were also considered. In contrast to [5], in [4] the control of all components used by BALSA were resynthesised. So far, solid results about the achievable improvements with resynthesis are missing due to three main problems: First, the constructed STGs suffer from state space explosion for large, real world benchmarks. Second, the established logic minimisation tools fail at inserting reset logic reliably. Third, to get simulation results for the entire circuit, the data path has to be added. But this is not trivial due to the mixed HS-components because the control has to be separated from the data path. To overcome these problems we started to improve and complete the most comprehensive approach of [4]; we build an appropriate tool with the following main characteristics:

- To tackle state explosion, we use an STG decomposition algorithm which has been adjusted to the special properties of Balsa-STGs [6].
- We develop a new logic synthesiser (based on ideas of PETRIFY) featuring a proper reset insertion mechanism – by inserting the reset logic *before* technology mapping.
- For each mixed HS-component, we extracted its data path and added a suitable interface to its control part.

Although the tool is not yet complete (some local optimizations are not implemented and only a simple logic decomposition algorithm is used for now) it is already possible to combine the resynthesised control path with the adapted data path to a working circuit for many BALSA benchmarks, among them circuits which could not be resynthesized before because of their complexity. The post synthesis simulation results concerning the performance are promising. Furthermore, due to the separation of control and data path, opportunities for data path optimisations are revealed. Using standard synchronous optimisation tools, we achieve first promising results showing a significant further potential for performance improvement.

The paper is organised as follows: in the next section we summarise essential basic notions in the context of resynthesis. Section III describes the overall design flow and the therein integrated resynthesis flow as an overview. In section IV and V we discuss how control and data path are constructed, respectively. In Section VI we show how these two parts are working in combination. The post synthesis simulation results are presented in Section VII. We draw conclusion and future work in Section VIII.

II. Background

We assume the reader is familiar with Petri nets, STGs, their state graphs, and SI Implementability, see [7], [8] for details. Nevertheless, some selected basic notions as well as the concept of STG decomposition are provided here.

This research was partly supported by DFG-project 'Optacon' VO 615/10-1 and WO 814/3-1.

Signal Transition Graphs

A Signal Transition Graph (*STG*) is a Petri net that models the desired behaviour of an asynchronous circuit. An STG is a tuple $N = (P, T, W, l, M_N, In, Out, Int)$, where $(P, T, W, M_N, Sig^{\pm}, l)$ is a Petri net, In, Out and Int are disjoint sets of *input*, *output* and *internal signals*, and $Sig = In \cup Out \cup Int$ is the set of all signals; *signature* refers to this partition of the signal set. $Sig^{\pm} = Sig \times \{+, -\}$ is the set of *signal edges* or *signal transitions*; its elements are denoted as $s+$, $s-$ resp. instead of $(s, +)$, $(s, -)$ resp. A plus sign denotes that a signal value changes from *low* to *high*, and a minus sign denotes the opposite direction. l is the *labeling function* $l : T \mapsto Sig^{\pm} \cup \{\lambda\}$ associating each transition t with one of the signal edges or with the empty word λ. In the latter case, we call t a *dummy transition*; it does not correspond to any signal change. The graphical representation of the STG is as usual; transitions labelled with an input signal edge have thick borders. The idea of *parallel composition* $N_1||N_2$ is that the two STGs run in parallel synchronizing on common signals [9].

SI Implementability

A circuit is speed-independent if its fundamental behaviour does not depend on the delay of its gates [8, p. 83]. STGs are used for specifying the behaviour of speed-independent (SI) circuits. For an STG N, a *state vector* is a function $sv : Sig \mapsto \{0, 1\}$, assigning a boolean value to each signal. For an STG N a *state assignment* assigns a state vector sv_M to each marking M of its reachability graph RG_N, forming the state graph. If there is a state assignment, an STG N has the *complete state coding* property (*CSC*) if any two reachable markings M_1 and M_2 with the same state vector (i.e. $sv_{M_1} = sv_{M_2}$) enable the same output and internal signals. Otherwise, N has a *CSC conflict* and no circuit can be synthesised directly. For logic synthesis, the entire state space of the STG must be explored, leading to state space explosion [10]. To tackle this problem, STG decomposition was introduced.

STG Decomposition

For the STG decomposition algorithm of [7], a *partition* of the output signals of the given STG N is chosen, and the algorithm decomposes N into component STGs, one for each set in this partition. For synthesis, equations for the outputs of each component are derived from the respective state graph, instead of deriving the equations from the overall state graph of N.

Very often, the cumulated states of all component state graphs yields a number much smaller than the state count of N, in which case the decomposition enables to overcome the state space explosion problem. Actually, it might already be beneficial if each state graph is smaller than the one of N, in particular for reducing peak memory usage. The decomposition is always correct, i.e. the parallel composition of the components matches the behaviour of N – cf. [7].

In more detail, decomposition starts from a deterministic, consistent specification N without internal signals [11]. First, one chooses a *feasible partition*, i.e. a family $(In_i, Out_i)_{i \in I}$ for some set I such that the sets Out_i are a *partition* of Out, $In_i \subseteq Sig \setminus Out_i$ for each i – and some additional conditions, most importantly: If there are $t, t' \in T$ with $t' \in (t^{\bullet})^{\bullet}$ (t is called *syntactical trigger of* t'), then $l(t') \in Out_i$ implies $l(t) \in In_i \cup Out_i$. Observe: if we have a feasible partition, we can build another feasible one by adding additional input signals to one of the members.

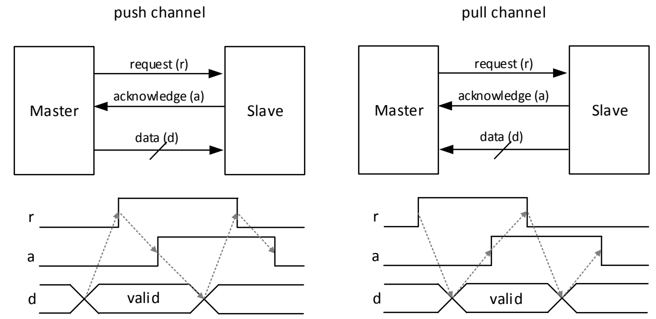

Fig. 1. 4-phase bundled data HS protocol

For each member (In_i, Out_i) of the partition, an *initial component* is generated: in a copy of the original STG N, every signal not in $In_i \cup Out_i$ is *lambdarised* and the signals in In_i are considered as inputs of this component – even if they are outputs of N. Then *reduction operations* are applied, most importantly the *contraction* of a λ-labelled transition (see e.g. [12] for an early reference). It removes a λ-labelled transition and in some way merges its preset with its postset; consequently, repeating this can contract a longer path into a place. So-called *secure* transition contractions preserve the essential behaviour, cf. [7]. Further reduction operations for components during decomposition are e.g. the deletion of redundant transitions and implicit places (see [4] for a more comprehensive discussion).

4-phase bundled data HS protocol

Bundled data describes a protocol where each bit of a data signal is encoded with one bit (in contrast to n-rail protocols) and is bundled with a request and an acknowledge signal. All three signals together form a *channel*. Depending on the flow direction of the data signals one can classify two types of channels: *push* (data flows from master to slave) and *pull* (other direction) channels. If there are no data signals the channel is called a *sync* channel. In Fig. 1 the proper signal transitions for a 4-phase protocol are shown.

III. Resynthesis overview

We propose a design system as shown in Fig. 2. Just like the original BALSA flow, we start with a Balsa program which is transformed into a Breeze netlist with the Balsa compiler (BALSA-C). In our case a Breeze netlist specifies a network of HS-components communicating via the 4-phase bundled data HS protocol. Afterwards, a Verilog netlist is generated from this Breeze file. In the original design flow this is done by BALSA-NETLIST – we introduce a new (resynthesis) tool

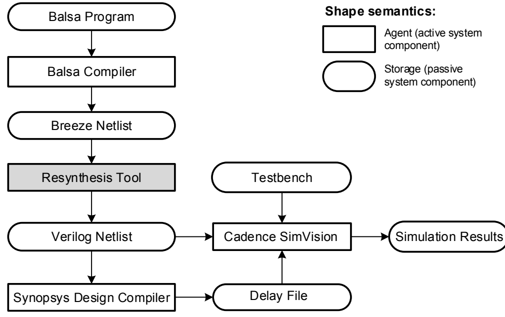

Fig. 2. Overall design system

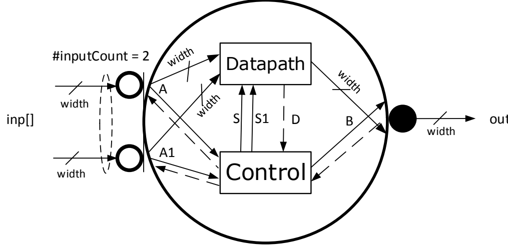

Fig. 3. Splitting of a *CallMux* with scale factor $n = 2$

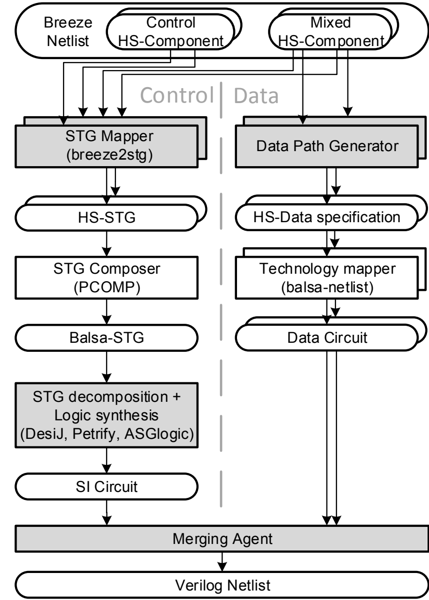

Fig. 4. Resynthesis tool (ASGRESYN) architecture

which will do this step in an optimised manner. The Verilog netlist, its derived delay file and a testbench for the design are given to a simulator to do performance tests (discussed in Sec. VII).

The main idea of our resynthesis approach is to separate the control part of the behaviour of the HS-components from the data path. This results in a new internal structure as shown in Fig. 3 for the *CallMux* component. The *CallMux* component has n writing channels on the left and multiplexes these onto the channel on the right hand side. In our approach, we split the component into a data and a control part. All incoming and outgoing request (solid line) and acknowledge signals (dashed line) are connected to the control, while all data signals (bold solid line with width information) are connected to the data path. For an easier processing later on, we introduced new names for channels. The $inp[]$ channel of the *CallMux* is a scaled channel, meaning there might be n (input) components connected to it, building an $n : 1$ multiplexer. Because this channel is scaled in the original definition, A the corresponding channel in our implementation, is also scaled; in the example $n = 2$, resulting in channels *A* and *A1*. For the communication between the control and data path, we have to introduce new signals. In the case of the *CallMux*, we need a scaled request-only channel S (for $n = 2$: *S* and *S1*) from control to data that states which channel was requested to multiplex and as answer an (unscaled) acknowledge-only channel D which states that the multiplexing (data) operation was done.

The general architecture of the resynthesis tool (ASGRESYN) is shown in Fig. 4. The HS-components of the Breeze netlist are divided into pure control HS-components and mixed HS-components. The latter contain both control and data path (like e.g. the *CallMux* component). HS-components like *Sequence* or *Concur* do not have any data lines attached, resulting in no need for a data path in our approach. However, the *Arbiter* component also doesn't have data lines, but needs a special part for making the decision (Mutex element) – this is done within our data path.

For the control part of every HS-component instance (pure control and mixed) a behaviour equivalent HS-STG is constructed. The HS-STGs of all components are merged with optimised parallel composition [9]. The resulting STG represents the overall control behaviour of the circuit and is called *Balsa-STG*. Due to complexity problems, STG decomposition may be applied and an SI circuit is generated with a logic synthesis tool. For every mixed HS-component a corresponding data path is generated. Both parts are merged and instantiated. These procedures are discussed more precisely in the following sections.

IV. Building the Control Path

A. HS-STG generation

The ideas of this subsection are discussed in more detail in [4]. To represent the control path of each HS-component we introduce behaviour-equivalent HS-STGs. These are generated from high-level expressions, derived from the original expressions presented in [13]. E.g. the expression for the *CallMux* looks as follows:

$$\begin{aligned} &active && B, S, D \\ &scaled && A, S \\ &f &=\;& \#(\#|(A : (rS; aD; B))) \end{aligned}$$

There are four channels A, B, D, and S (cf. Fig. 3). The channels B, D, and S are active channels, which means that the request signal is an outgoing and the acknowledge signal is an incoming signal. Thus A is a passive channel, resulting

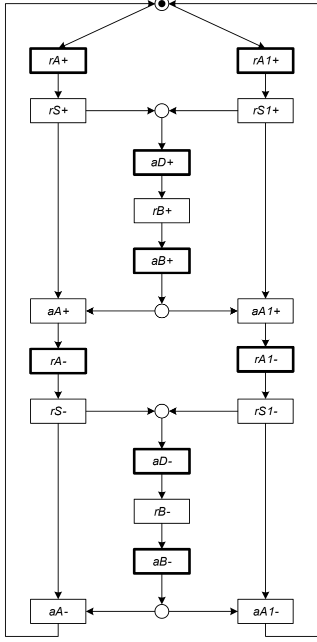

Fig. 5. CallMux-STG with scale-factor 2

in an reversed flow for the request and acknowledge signal. Note that there are no data signals attached to these (or in other words: all these channels are sync channels), because this only models the control part. The function of the *CallMux* begins with a loop construct (#), meaning after finishing the handshake in the corresponding braces, the handshake could start again. The $\#|$ forms a repeated choice, which repeats this section n times, where n is the scale factor. For every repetition, every scaled channel (in this case A and S) appears with a scaled name, e.g. for the scale factor 2 the expression is expanded into $A0 : (rS0; aD; B)$ and $A1 : (rS1; aD; B)$. Both expression are connected with a choice operator. The enclosure operator (:) expands into $rA0+; rS0+; aD+; rB+; aB+; aA0+; rA0-; rS0-; aD-;$ $rB-; aB-; aA-$, where ';' is the sequence operator. The resulting HS-STG for a scale factor of 2 is shown in Fig. 5. Note that in the implementation $rA0$ and $aA0$ are denoted as rA and aA respectively.

Such an STG is generated for every instance of a HS-component in a Breeze netlist and called HC-STG. The signal names are replaced with the actual channel IDs or a component ID for the new control-data communication channels. All those STGs are joined together with advanced parallel composition [9] forming the Balsa-STG. Afterwards most internal signals (handshake signals between two HS-components) are dummified and reduction operations are applied. This STG may be synthesised with a logic synthesis tool, like PETRIFY [14].

B. State space explosion problem

However, for large benchmarks like most of the *SAMIPS* [15] components, the STGs suffer from state space

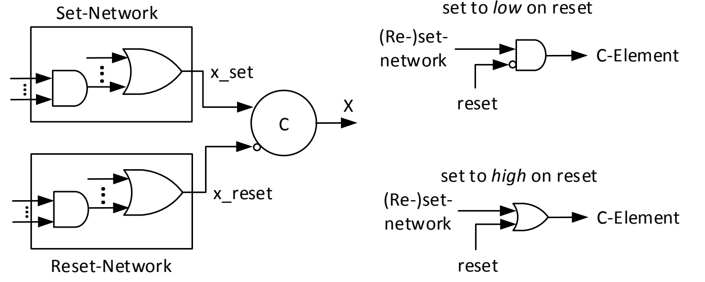

Fig. 6. Reset insertion

explosion. To tackle this, STG decomposition was introduced e.g. in [7]. Unfortunately, STG decomposition was not successful in all cases to avoid *irreducible* CSC conflicts[1].

With our current construction, all Balsa-STGs have a specific structure. In particular, no HS-STG has input transitions connected sequentially (avoiding so-called self-triggers [16] – the most frequent reason for irreducible CSC conflicts). The parallel composition of such STGs preserves this property, and thus it is also preserved in the Balsa-STG. The idea is to decompose the Balsa-STG with DESIJ [17] into components that preserve this quality. For this, the so-called *common cause* partition puts two output signals in the same part (component) of the output partition if they have a common trigger signal (either input or output). With this approach we could resolve *all* CSC conflicts for *every* BALSA benchmark available to us [6].

C. Reset problem

The second main problem during first evaluation tests was, that the simulation sometimes didn't finish properly. As it turned out the reset logic inserted by PETRIFY contained hazards. We searched for alternative tools to tackle the reset problem with PETRIFY. We found two publications which also identified this problem: [18] and [19]. Unfortunately, we didn't gain access to the latter one. PETRESET [18] didn't work with our 130nm library, but delivered partly working solutions with 250nm. However, there was no tool which could generate a working reset logic in every case. All three tools (PETRESET, the tool from [19] and PETRIFY) insert the reset logic *after* synthesis and technology mapping. At this point of the flow, it is hard to insert the reset in a hazard-free manner, since no new gates can be added but existing ones must be used [19]. Therefore, we propose a solution which inserts the reset logic right after synthesis and *before* technology mapping and started to implement a new logic synthesis tool which realises this approach. The implementation of the state graph construction and equation derivation is based on the ideas from [8].

The derived functions for the set- and reset-network for the C-Element (cf. Fig. 6 on the left hand side) are constructed as sum-of-products satisfying the monotonic cover conditions [8,

[1]A CSC conflic is called *irreducible*, if CSC cannot be achieved by signal insertion.

p. 134]. The reset is now inserted in front of the C-Element right after the OR gate level (In Fig. 6 at the position of signals x_set and x_reset). Depending on which level the signal (which is produced by the C-Element) should have at the initial state, one network must be *low* and the other one *high*. If the signal should be *high* (*low*), the set network should be *high* (*low*) and the reset-network must be *low* (*high*) to work properly. For an *active high* reset input, at the end of to-be *high*-network, an "OR reset" gate has to be added; to get a network to *low*, an "AND NOT reset" gate must be inserted (see Fig. 6 on the right hand side). If the reset signal is *low*, these gates have logically no impact (behaves like a buffer). In this case, this reset logic gates do not introduce hazards. Because they are placed at the end of their assigned network (and therefore it is the last switching signal in this network) and the reset input is constant (*low*), they do not introduce a critical race at the input of the C-Element. Because there may occur critical races between the two networks even without reset, the reset does not change this fact. However, this is compensated by the C-Element. Avoiding forks between the end of a network at its reset logic we can easily argument that, assuming arbitrary gate delays (SI assumption), the delay of the reset gate (with constant *low*) can be counted towards the OR gates delay. But, if the reset signal is *high*, there might be glitches on the output, because reset may occur at any time – even if the circuit is computing the next state. Even worse, on the outputs one cannot observe if the circuit has settled. Therefore reset must be kept on *high* for an upper bound of the longest path between reset input and all other inputs of the reset logic gates. Switching reset from *high* to *low*, assuming all inputs are on their initial value, the circuit starts to behave as specified by the STG.

This approach can be optimised by checking if a network is self-resetting, i.e. if the logic itself produces the correct level if all inputs are at their initial value. This leads to a higher upper bound for the needed *high*-phase of the reset signal and an additional check for circular dependencies must be done (similar to [19]).

For technology mapping, we implemented methods presented in [20]. Implementing the set- and reset function under the monotonic cover conditions, it is trivial to decompose OR gates, but hard to decompose AND gates. For AND gate decomposition all traces in the specification have to be considered. This algorithm works well for smaller Balsa benchmarks (cf. section VII), but sometimes fails for larger ones. Thus we have to improve this algorithm in the future. After decomposition, the smallest subset of gates that covers all logic blocks is chosen to implement the circuit.

V. Building the Data Path

The BALSA-NETLIST tool is capable of implementing Balsa circuits with different handshake protocols (named *styles*), e.g. dual-rail or four-phase bundled data. We introduce a new style, which specifies the data path for our resynthesised circuit in an abstract manner. It is based on the four-phase bundled data style. For every HS-component, we had to check which controlling logic is still needed or could be removed. In some cases we had to add logic, depending on the assumptions made while designing the control expressions. The data processing parts kept untouched in all cases. The interface of the component has to be altered, so it is compatible with our control path.

Listing 1. CallMux datapath

```
module BrzCallMux_3_2 (
  inp_0r, inp_0a, inp_0d,
  inp_1r, inp_1a, inp_1d,
  out_0r, out_0a, out_0d,
  initialise
);
  input inp_0r, inp_1r, out_0a, initialise;
  output inp_0a, inp_1a, out_0r;
  input [2:0] inp_0d;
  input [2:0] inp_1d;
  output [2:0] out_0d;
  wire select_0n, nselect_0n;
  MUX2 I0 (out_0d[0], inp_0d[0], inp_1d[0], select_0n);
  MUX2 I1 (out_0d[1], inp_0d[1], inp_1d[1], select_0n);
  MUX2 I2 (out_0d[2], inp_0d[2], inp_1d[2], select_0n);
  MUX2 I3 (out_0r, inp_0r, inp_1r, select_0n);
  AN2 I4 (inp_0a, nselect_0n, out_0a);
  AN2 I5 (inp_1a, select_0n, out_0a);
  srff I6 (inp_1r, inp_0r, select_0n, nselect_0n,
   initialise);
  assign select_0n = inp_1r; //added
endmodule
```

In listing 1, a Verilog example for a 3-bit 2:1 *CallMux* component is given. The listing shows the original BALSA implementation. The removed parts in our implementation are marked with a strike trough. The first three multiplexers (I0, I1, I2) must kept because these are the actual data processing gates. The multiplexer I3 delays the request signal (from either inp_0 or inp_1) for the same duration it takes to process the data[2]. The two AND gates I4 and I5 are not needed, because they generated acknowledge signals which are obsolete. The only acknowledge signal which is still needed (to signal the control that the multiplexers have switched) is out_0r. Also, the SR-latch is no longer required, because the only reason to store the information which channel has been selected to acknowledge the proper channel later on. This is now done by the control part, too. Instead of the SR-latch, inp_1r is used as selector for the multiplexers.

VI. Example

Listing 2. Example program

```
procedure muxtest (input i1 : byte;
  input i2 : byte; output o : byte) is
begin
 i1 -> o;
 i2 -> o
end
```

Until this point, we have only discussed how the mapping of single component is done. In this section we want to show the differences between the original BALSA and the resynthesis implementation of an entire (simple) Breeze netlist. In listing 2 a simple Balsa program is given: there are two inputs, which are sequentially pushed onto the output. Fig. 7 shows the Balsa implementation of this program. This module is connected to

[2]In this model we assume that there are no wire delays and that gates have symmetric edges.

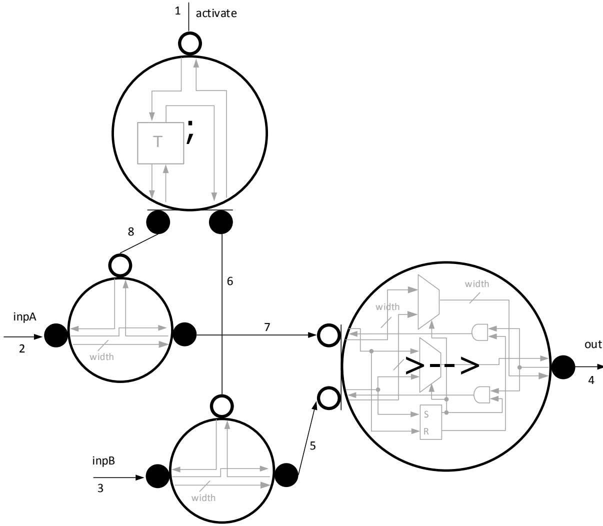

Fig. 7. Example: Balsa implementation

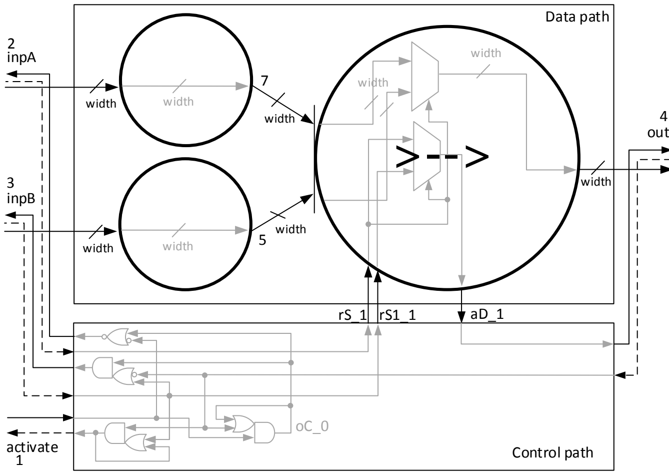

Fig. 8. Example: Resynthesis implementation

one active and three passive *ports*. The environment activates this module by starting a handshake on the activate port (channel 1). Afterwards, the module requests data on inpA and pushes these into port out, followed by the same procedure for port inpB. The *Sequence* component (;) is activated by channel 1 and then activates the channels 8 and 6 sequentially. It contains a T-Element which in our case is build with 10 logic gates. The two *Fetch* components are just transporting the data from their input channels (2, 3) to their output channels (7, 5) after they were activated (8, 6). They just contain wires. The *CallMux* ($> \text{---} >$) is constructed as shown in listing 1 (including the crossed out lines), containing a width-bit multiplexer, an 1-bit multiplexer, an rs-Latch and two AND-gates.

In Fig. 8 the corresponding resynthesised implementation is shown. Here the ports are split into their data lines (if present), which are connected to the data path, and their control wires (request and acknowledge) which are connected to the control

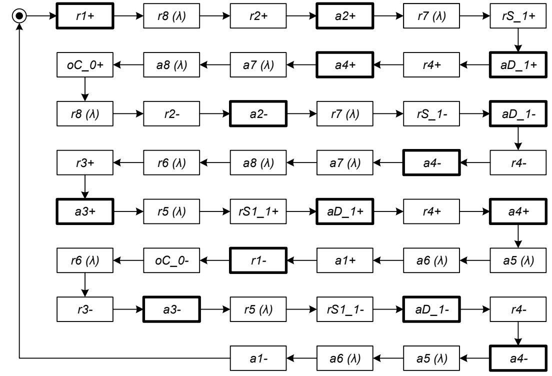

Fig. 9. Example: Balsa-STG for the control part

path. Because *Sequence* is a pure control component, there is no corresponding component in our data path; its behaviour is completely implemented by the control path. The two *Fetch* components are still just wires, only containing the data lines. Note that no communication between the data and control path is needed in this component because it is currently assumed that there are no wire delays (SI assumption). (This communication protocol may change if wire delays come into play.) The *CallMux* is constructed as shown in listing 1 (without the crossed out lines), containing a width-bit multiplexer and an 1-bit multiplexer.

After constructing the Balsa-STG for the control path (Fig. 9)[3], we can dummify all eight inter-component communication signals: r5, a5, r6, a6, r7, a7, r8, and a8, because they are internal [4]. Thus, no logic has to be constructed to generate these signals. However, we have to introduce new signals during our workflow: rS_1, rS1_1, aD_1 as communication signals between control and data path of *CallMux* and oC_0 as a preventive CSC solving signal. For this example we achieve a performance improvement up to 31%, depending on different synthesis options (discussed in the following section).

VII. Experimental results

Using the proposed flow in Fig. 4, we ran some benchmarks to measure the performance improvement. In this case we define performance as the latency time for a specific calculation example for a benchmark. All simulations are post-synthesis simulations, thus currently wire delays are not considered. As technology we have used a standard 130nm technology from the IHP GmbH. This library does not contain any asynchronous-specific cells like C-Elements[4] or Mutexes.

A. Simple benchmarks

In Table I we used simple well-known benchmarks to compare the original Balsa implementation with ours. The resynthesis section is divided into direct synthesis, which uses the entire Balsa-STG constructed from the HS-components,

[3] Note that in this trivial example the STG is purely sequential, but Balsa-STGs in general specify concurrent behaviour.

[4] Instead of original C-Elements, we use set- and reset-dominant C-Elements/RS-Latches.

TABLE I
RESULTS: SIMULATION TIME

	Balsa	Resynthesis							
		Without STG decomposition (direct)				With STG decomposition			
		Petrify		ASGlogic		Petrify		ASGlogic	
gcd(127,1)	949,261 ps	Simulation failed (1)	N/A	792,076 ps	-16,6%	764,535 ps	-19,5%	822,120 ps	-13,4%
counter{10}	42,562 ps	Synthesis failed	N/A	34,362 ps	-19,3%	Synthesis failed	N/A	41,232 ps	-3,1%
counter{10} alt.	50,168 ps	35,745 ps	-28,7%	38,616 ps	-23,0%	44,645 ps	-11,0%	48,743 ps	-2,8%
shifter(3)	15,551 ps	13,494 ps(2)	-13,2%	14,933 ps	-4,0%	14,106 ps	-9,3%	14,936 ps	-4,0%
buffer{10}	8,426 ps	8,120 ps	-3,6%	8,043 ps	-4,5%	8,113 ps	-3,7%	8,043 ps	-4,5%

(1) Due to not properly working reset logic

(2) In 6 of 10 synthesis runs, the corresponding simulation failed due to not properly working reset logic

and the STG decomposition-based approach. For logic synthesis itself, PETRIFY and our tool (ASGLOGIC) are used (and compared) to generate the netlists. Because PETRIFY could never synthesise the original counter implementation (due to *Loop* component and its resulting STG), we also use an alternative implementation of *counter*, where an infinite loop is build with a *While* component with an expression which is always true. PETRIFYS solutions are in most cases faster than these of our tool. Despite that PETRIFY could synthesise every benchmark (except the original counter), sometimes the simulation did not finish, thus the circuits did not behave properly. We have analysed the waveforms and detected that for these benchmarks the reset logic inserted by PETRIFY was not hazard-free. For the shifter-benchmark, we even observed that the reset occasionally worked in different synthesis runs. Therefore, we developed our new logic synthesis tool as discussed in section IV-C. However, while using STG decomposition, we did not observe this reset problem with PETRIFY. Currently it is not really clear why the problem did not occur or if it is maybe resolved by the new STG decomposition algorithm with the common-cause heuristics. A second observation on the decomposition solutions is that they are in most cases slower than their direct-synthesis counterpart. This is expected, because more signals are kept unlambdarised during decomposition, resulting in more logic which has to generate these. We also observed the need for more signals to solve CSC. However, for large benchmarks the Balsa-STGs are too large to synthesise them entirely, which is resolved by STG decomposition (see below). Surprisingly, for the buffer benchmark we observed that the decomposition result with PETRIFY is faster than the direct-synthesis counterpart. In this case the overall reset logic for all decomposition components is simpler than the reset logic for the entire STG.

B. SAMIPS

In Table II we have tried to resynthesise all components of the *SAMIPS* processor [15]. The table entries state if the synthesis was successful (yes) or not (no). Cells with decomposition (right hand side) have the number of failed (decomposition) components in parenthesis. One can see that without decomposition, there are just a few benchmarks which were successfully resynthesised. Considering the number of their components (#Comp.) when resynthesised with STG decomposition, we can assume that they are really small benchmarks. After applying STG decomposition a half of the benchmarks could be resynthesised successfully. If we combine the results of the two logic synthesis tools it is nearly two third. Thus, decomposition is essential for the success of our resynthesis approach. The most failures during component synthesis are resulting from non-decomposable AND gates, which we try to fix in the future. Unfortunately, until now we don't have any testbenches for the SAMIPS' components and couldn't do any performance analysis yet.

TABLE II
EXPERIMENT: SUCCESS OF RESYNTHESIS OF SAMIPS

	Resynthesis w/o STG decomposition (direct)		Resynthesis with STG decomposition		
	Petrify	ASGlogic	#Comp.	Petrify	ASGlogic
SAMIPS	no	no	35	**yes**	**yes**
CP0	no	no	210	no (3)	**yes**
EX	no	no	390	no (20)	no (29)
FWunit	no	no	22	**yes**	no (1)
shift	no	no	53	**yes**	no (1)
ID	**yes**	**yes**	1	**yes**	**yes**
DeCode	no	no	213	no (11)	no (24)
RegBank	no	no	343	no (14)	no (12)
IF	**yes**	**yes**	5	**yes**	**yes**
AAU	no	no	70	no (3)	no (4)
ADD4	**yes**	no	7	**yes**	**yes**
Arb1	no	no	9	**yes**	**yes**
Arb2	no	no	40	no (2)	no (2)
PC	no	no	10	no (3)	**yes**
MEM	no	no	92	no (2)	no (6)
WB	no	no	22	**yes**	**yes**

C. Data path experiments

As stated before, the splitting of control and data path reveals the opportunity to optimise both independently. Since control resynthesis was quite successful, we started to experiment with data path optimisations. Our first idea was to use

TABLE III
EXPERIMENT: DATA PATH OPTIMISATION

	Balsa	Resynthesis with STG decomposition			
		Petrify		ASGlogic	
gcd(127,1)	949,3 ns	415,6 ns	-56,2%	417,8 ns	-56,0%
counter{10}	42,6 ns	Syn failed	N/A	30,1 ns	-29,3%
counter{10} alt.	50,2 ns	36,1 ns	-28,1%	37,3 ns	-25,7%
shifter(3)	15,6 ns	11,1 ns	-28,4%	11,3 ns	-27,2%
buffer{10}	8,4 ns	7,5 ns	-10,9%	7,4 ns	-12,3%

standard (synchronous) optimisers to do so. We commanded this tool to keep the control parts untouched, due to its lack of knowledge about SI circuits. The results are shown in Table III. This approach worked surprisingly well even without explicitly performing delay matching[5]. All of our simple benchmarks did run properly (except for the original counter with PETRIFY, as discussed before). However this approach will not work in all cases. There are currently no delay constraints between data and control path, therefore it is possible for a control signal to arrive earlier than the corresponding data signal. Thus, we have to introduce suitable delay constrains in the future.

VIII. CONCLUSION AND FUTURE WORK

In this paper a fully automated resynthesis flow for BALSA was presented. We achieve a performance improvement of up to 23% by just optimising the control path. To tackle state explosion, we use an STG decomposition algorithm which has been adjusted to the special properties of Balsa-STGs (using the common-cause partition). We started to develop a new logic synthesiser featuring a proper reset insertion mechanism – by inserting the reset logic before technology mapping. For each mixed HS-component, we extracted its data path and added a suitable interface to its control part. All tools are open source an can be downloaded on GitHub: https://github.com/hpiasg.

However, currently not all Balsa benchmarks can be resynthesised. Our logic synthesis tool is yet not capable of decomposing all AND gates, thus not all logic functions can be technology mapped properly.

To also improve the data path, as a first attempt we used a standard (synchronous) optimiser and achieved an overall performance improvement of up to 56% for simple benchmarks. To do further experiments (e.g. comparison with synchronous solutions [21]), we have to introduce proper matched delays between control and data path; this mechanism is also needed to place & route an SI circuit properly.

ACKNOWLEDGEMENTS

We thank the IHP GmbH (Frankfurt/Oder) for providing the design environment and the design kits, in particular Milos Krstic and Steffen Zeidler for their helpful advice. Best thanks to Walter Vogler and Stanislavs Golubcovs for fruitful cooperation making this work possible.

[5]Note that Balsa either inserts delay chains or completion detection logic.

REFERENCES

[1] A. Bardsley and D. A. Edwards, "The Balsa asynchronous circuit synthesis system," in *Forum on Design Languages*, Sep. 2000.

[2] T. Yoneda, A. Matsumoto, M. Kato, and C. Myers, "High level synthesis of timed asynchronous circuits," in *Asynchronous Circuits and Systems, 2005. ASYNC 2005. Proceedings. 11th IEEE International Symposium on*, March 2005, pp. 178–189.

[3] F. Fernández-Nogueira and J. Carmona, "Integrated circuit and system design. power and timing modeling, optimization and simulation," L. Svensson and J. Monteiro, Eds., 2009, ch. Logic Synthesis of Handshake Components Using Structural Clustering Techniques, pp. 188–198.

[4] S. Golubcovs, W. Vogler, and N. Kluge, "STG-based resynthesis for balsa circuits," in *Proceedings of the 2013 13th International Conference on Application of Concurrency to System Design*, ser. ACSD '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 140–149.

[5] A. Alekseyev, I. Poliakov, V. Khomenko, and A. Yakovlev, "Optimisation of balsa control path using stg resynthesis," 21st UK Asynchronous Forum, 2009.

[6] S. Golubcovs and W. Vogler, "Decomposing balsa-stgs (working notes)," Institute of Computer Science, University of Augsburg, Tech. Rep., 2014.

[7] W. Vogler and B. Kangsah, "Improved decomposition of signal transition graphs," *Fundamenta Informaticae*, vol. 76, pp. 161–197, 2006.

[8] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev, *Logic synthesis of asynchronous controllers and interfaces*, ser. Advanced Microelectronics. Springer-Verlag, 2002.

[9] A. Alekseyev, V. Khomenko, A. Mokhov, D. Wist, and A. Yakovlev, "Improved parallel composition of labelled Petri nets," in *Proceedings of the 2011 Eleventh International Conference on Application of Concurrency to System Design*, ser. ACSD '11, 2011, pp. 131–140.

[10] A. Valmari, "The state explosion problem," in *Lectures on Petri Nets I: Basic Models, Advances in Petri Nets, the Volumes Are Based on the Advanced Course on Petri Nets*, 1998, pp. 429–528.

[11] M. Schaefer and W. Vogler, "Component refinement and CSC solving for STG decomposition," *Theoretical Computer Science*, vol. 388, no. 1–3, pp. 243–266, 2007.

[12] C. André, "Structural transformations giving b-equivalent PT-nets," in *Selected Papers from the 3rd European Workshop on Applications and Theory of Petri Nets*. London, UK: Springer-Verlag, 1983, pp. 14–28.

[13] D. Edwards, A. Bardsley, L. Janin, L. Plana, and W. Toms, "Balsa: A tutorial guide." The Advanced Processors Technologies Research Group, The University of Manchester, Tech. Rep., 2006.

[14] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev, "Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers," *IEICE Transactions on Information and Systems*, vol. E80-D, no. 3, pp. 315–325, 1997.

[15] Q. Zhang and G. Theodoropoulos, "Modelling samips: A synthesisable asynchronous mips processor," in *Proceedings of the 37th Annual Symposium on Simulation*, ser. ANSS '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 205–212.

[16] D. Wist, M. Schafer, W. Vogler, and R. Wollowski, "STG decomposition: Internal communication for SI implementability," in *Application of Concurrency to System Design (ACSD), 2010 10th International Conference on*, June 2010, pp. 13–23.

[17] M. Schaefer, D. Wist, and R. Wollowski, "Desij–enabling decomposition-based synthesis of complex asynchronous controllers," in *Application of Concurrency to System Design, 2009. ACSD '09. Ninth International Conference on*, July 2009, pp. 186–190.

[18] I. David, L. Cohen, and R. Dobkin, "Petreset," 2010, unpublished.

[19] V. S. Vij and K. S. Stevens, "Automatic addition of reset in asynchronous sequential control circuits." in *VLSI-SoC*, M. Margala, R. A. da Luz Reis, A. Orailoglu, L. Carro, L. M. Silveira, and H. F. Ugurdag, Eds. IEEE, 2013, pp. 374–379.

[20] P. Siegel and G. De Micheli, "Decomposition methods for library binding of speed-independent asynchronous designs," in *Proceedings of the 1994 IEEE/ACM International Conference on Computer-aided Design*, ser. ICCAD '94, 1994, pp. 558–565.

[21] S. Zeidler, M. Goderbauer, and M. Krstic, "Design of a low-power asynchronous elliptic curve cryptography coprocessor," in *Electronics, Circuits, and Systems (ICECS), 2013 IEEE 20th International Conference on*. IEEE, 2013, pp. 569–572.