Synchronous Elasticization at a Reduced Cost: Utilizing the Ultra Simple Fork and Controller Merging

Eliyah Kilada and Kenneth S. Stevens
University of Utah
eliyah.kilada@utah.edu, kstevens@ece.utah.edu

***Abstract*—Synchronous elasticization converts an ordinary clocked design into Latency-Insensitive (LI). It uses communication protocols such as SELF. Comparing to lazy implementations, eager SELF has no cycles and can provide performance advantage. Yet, it uses eager forks (EForks) consuming more area and power. This paper demonstrates that EForks can be redundant. A novel ultra simple fork (USFork) implementation is introduced. The conditions under which an EFork will behave exactly the same as a USFork (from the protocol perspective) are formally derived. The paper also investigates the conditions under which multiple SELF controllers can be merged to further decrease the area and power overhead (as long as the physical placement allows). The flow has been integrated in a fully automated tool, HGEN. HGEN uses 6thSense as an embedded verification engine. Comparing to the methodology used in published work on a MiniMIPS processor case study, HGEN shows up to 34.3% and 25.4% savings in area and power due to utilizing USForks. It also shows *at least* 32% saving in the number of EForks in s382 ISCAS benchmark. More reduction is possible if the physical placement allows for controller merging. Thanks to the advance in synchronous verification technology, HGEN runs within few minutes (for all this paper examples). This makes the proposed approach suitable for tight time-to-market constraints.**

I. Introduction

Latency insensitivity (LI) [1] allows designs to tolerate arbitrary latency variations in their computation units as well as communication channels. This could be specially important for interfaces where the actual latency can not be accurately estimated or required to be flexible. Examples of the former is systems with very long interconnects. Interconnect latency is affected by many factors that can not be accurately estimated before the final layout [2]. On the other hand, some applications require flexible interfaces that tolerate variable latencies. Examples can include interfaces to variable latency ALUs, memories or network on chip. It has been reported that applying flexible latency design to the critical block of one of Intel SOC (H.264 CABAC) can achieve 35% performance advantage [3].

Synchronous elasticization [4], [5], [6] is a technique of converting an ordinary clocked design into an LI. Unlike asynchronous circuits, synchronous elastic circuits can be easily designed with conventional design flows using STA [5], [7]. The Synchronous Elastic Flow (SELF) [4] is a communication protocol in synchronous elastic designs.

Eager implementation of the SELF protocol has been reported in [4]. Such implementation is combinational-cycle free and can provide performance advantages in some designs comparing to lazy implementations. However, the former is more expensive in terms of area and power consumption. The LI control network area and power consumption overheads may become prohibitive in some cases [5]. In fact, measurements of a MiniMIPS processor fabricated in a 0.5 μm node show that elasticization with an eager SELF implementation results in area and dynamic power penalties of 29% and 13%, respectively [8]. Therefore, minimizing these overheads is a primary concern. An algorithm that minimizes the total number of control steering units (i.e., joins and forks) in the LI control network (and, hence, its area and power overheads) has been proposed in [9].

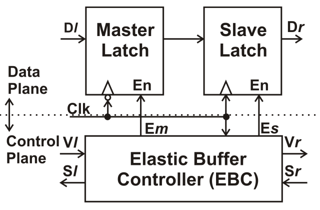


Fig. 1: An Elastic Buffer (EB) implementation.

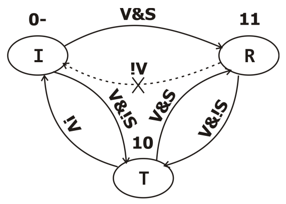


Fig. 2: SELF channel protocol.

Due to its lower area and power overheads, lazy SELF implementation can become an attractive solution. However, such implementation typically suffers from combinational cycles that can cause deadlock or oscillation [8]. Authors of [8] also showed that running the same testbench program on a MiniMIPS processor implemented with lazy SELF takes 32.7% and 58.8% longer runtime than an eager implementation in case of 1 and 3 bubbles in the register file path, respectively.

Section III introduces the Ultra Simple Fork (USFork). As the name implies, the USFork implementation has no logic gates - just wired connections. We study the EFork transition diagram and formally derive the conditions under which an EFork can be replaced by a USFork. The transformation guarantees that the USFork will schedule exactly the same state transitions as the EFork over all its channels, thus maintaining the same runtime. Furthermore, utilizing the USForks results in no combinational cycles or frequency penalties. In essence, our approach replaces the *redundant* EForks with USForks resulting in a hybrid control network where both EForks and USForks are used.

Section IV investigates the conditions under which multiple SELF controllers can be merged into one controller. The transformation reduces the control network area and power overhead and is limited only by the physical placement constraints. SELF controller clustering has previously been reported in [10]. However, their approach requires the control network model to be closed (i.e, an abstract for the environment must be available). They also require *static* latency values inside the control network. On the other hand, the approach proposed in this paper can handle situations where the environment abstract is not available or required to be flexible. It can also handle designs with variable latency units.

The above two transformations have been integrated in a fully automated tool, HGEN. Section VI gives an overview of the tool. HGEN takes a verilog description of a control network and returns a verilog netlist of the minimized version (using one or two of the above transformations).

Finally, the paper is concluded with sample results in Section VII.

II. Synchronous Elastic Architectures

An elastic system uses Elastic Buffers (EBs) as synchronization elements as counterparts to flip-flops in ordinary clocked systems. Fig. 1 shows a block diagram of an EB [4]. EB Controllers (EBCs) communicate through control channels. A control channel in the SELF protocol is composed of two signals. 'Valid' (V) in the forward direction, indicates the validity of the data coming from the transmitter. 'Stall'(S) in the backward direction, indicates the receiver is not ready to receive the incoming data on the channel. SELF identifies three different states on a communication channel (shown in Fig. 2): 1) *Transfer* (T): $V\&!S$. The transmitter provides valid data and the receiver can accept it. 2) *Idle* (I): $!V$. The transmitter does not provide valid data. 3) *Retry* (R): $V\&S$. The transmitter provides valid data, but the receiver can not accept it. The transmitter will sustain the valid data until the receiver is able to read it. Hence, SELF protocol prohibits a transition from *R* to *I* states.

When there are more than one transmitter and one receiver EBs, a control network is required to connect the different EBs. A control network is composed of control channels connected through control steering units, namely, join and fork components. A join element joins two or more incoming control channels into one output control channel. A fork element forks one incoming control channel into two or more output control channels. Fork and join components are represented in this paper by $\odot$ and $\otimes$, respectively. Finally, the SELF protocol used over the control channels can be implemented in an eager, lazy or hybrid flavors. For brevity, we will refer to an eager implementation of the SELF protocol, as simply 'eager SELF', and same for 'lazy and hybrid SELF'.

TABLE I: EFork state table.

Current State					Next State Inputs			Next State					
s_i	Q_1	Q_2	L	R_1	R_2	V_l	S_{r1}	S_{r2}	s_i	Q_1	Q_2	L	R_1	R_2
s_0	1	1	I	I	I	0	-	-	s_0	1	1	I	I	I
						1	0	0	s_1	1	1	T	T	T
						1	0	1	s_3	1	1	R	T	R
						1	1	0	s_4	1	1	R	R	T
						1	1	1	s_2	1	1	R	R	R
s_1	1	1	T	T	T	0	-	-	s_0	1	1	I	I	I
						1	0	0	s_1	1	1	T	T	T
						1	0	1	s_3	1	1	R	T	R
						1	1	0	s_4	1	1	R	R	T
						1	1	1	s_2	1	1	R	R	R
s_2	1	1	R	R	R	0	-	-	Illegal Transition					
						1	0	0	s_1	1	1	T	T	T
						1	0	1	s_3	1	1	R	T	R
						1	1	0	s_4	1	1	R	R	T
						1	1	1	s_2	1	1	R	R	R
s_3	1	1	R	T	R	0	-	-	Illegal Transition					
						1	0	0	s_5	0	1	T	I	T
						1	0	1	s_6	0	1	R	I	R
						1	1	0	s_5	0	1	T	I	T
						1	1	1	s_6	0	1	R	I	R
s_4	1	1	R	R	T	0	-	-	Illegal Transition					
						1	0	0	s_7	1	0	T	T	I
						1	0	1	s_7	1	0	T	T	I
						1	1	0	s_8	1	0	R	R	I
						1	1	1	s_8	1	0	R	R	I
s_5	0	1	T	I	T	0	-	-	s_0	1	1	I	I	I
						1	0	0	s_1	1	1	T	T	T
						1	0	1	s_3	1	1	R	T	R
						1	1	0	s_4	1	1	R	R	T
						1	1	1	s_2	1	1	R	R	R
s_6	0	1	R	I	R	0	-	-	Illegal Transition					
						1	0	0	s_5	0	1	T	I	T
						1	0	1	s_6	0	1	R	I	R
						1	1	0	s_5	0	1	T	I	T
						1	1	1	s_6	0	1	R	I	R
s_7	1	0	T	T	I	0	-	-	s_0	1	1	I	I	I
						1	0	0	s_1	1	1	T	T	T
						1	0	1	s_3	1	1	R	T	R
						1	1	0	s_4	1	1	R	R	T
						1	1	1	s_2	1	1	R	R	R
s_8	1	0	R	R	I	0	-	-	Illegal Transition					
						1	0	0	s_7	1	0	T	T	I
						1	0	1	s_7	1	0	T	T	I
						1	1	0	s_8	1	0	R	R	I
						1	1	1	s_8	1	0	R	R	I

III. Eager To Ultra Simple Fork Transformation

A. Eager SELF Protocol

An eager SELF implementation uses eager forks (EForks) and lazy joins. Study of lazy joins and forks are outside the scope of this paper. Fig. 4 shows a 2-output-channel EFork proposed in [4]. Once a (valid) data token is available at an EFork stem, it will immediately pass it to all its branches. Meanwhile, the EFork will stall until all its branches receive the data token. This gives an early start to the branches that are ready (i.e., their corresponding stall signal is Zero). Lazy forks, on the other hand, do not pass the data token from its stem to its branches until all branches are ready to receive. Hence, EFork can result in performance advantage over lazy forks in some systems. However, EFork incorporates one flip flop per branch that is triggered every clock cycle even if there is no activity in the control network. Moreover, eager forks have higher logic complexity comparing to lazy. All of that render the EFork expensive in terms of both area and power consumption.

B. Eager Fork State Diagram

A 2-output-channel EFork has 3 terminal channels: namely, L (Left), R_1 ($Right_1$) and R_2 ($Right_2$). L channel consists of signals V_l and S_l. Similarly, R_1 consists of V_{r1} and S_{r1}, and R_2 of V_{r2} and S_{r2}.

In order to compute the state diagram of an EFork, the behavior allowed by the SELF protocol over the fork 3 channels must be taken into account. Hence, the desired state diagram is obtained by composing the simple (2 flip-flop based) 4-state diagram of the EFork circuit of Fig. 4 with the SELF transition diagram of Fig. 2 (over the three terminal channels). The EFork state table and diagram are depicted in Table I and Fig. 3, respectively. In this diagram, the inputs V_l, S_{r1}, and S_{r2} are part of the state vector (along with the flip-flop outputs, Q_1 and Q_2). To simplify the notations, the state vector takes the following format: $<Q_1,Q_2,L,R_1,R_2>$, where L, R_1 and R_2 carry the corresponding channel status (i.e., I, T, or R). States with dot inside are reset states. Some of the transitions (and states) are not allowed (or reached) because of the SELF protocol constraints, and, hence, omitted from the diagram. Most of the transition labels were omitted from Fig. 3 for brevity.

C. Input Behavior Constraints

For a 2-output-channel EFork, the input vector, I, is a 3-tuple of signals $< V_l, S_{r1}, S_{r2} >\in \{0,1\}^3$. Subscript n is added to I and the 3 signals to denote the value at clock cycle n. We define S^I, to be an infinite sequence of input vectors ordered by the clock index. Hence, $S^I[n] = I_n$. We refer to the total input behavior, B^I_T, as the set of all input sequences. Some of the input sequences are not allowed by the SELF protocol. For example, the following sequence will cause an R to I transition on the L channel: $<< 1,0,0 >,< 1,1,1 >,< 0,1,1 >,.. >$. The set of all sequences which are excluded for violating the SELF protocol will be denoted as E^I_P. Nonetheless, in this Section, some of the sequences will also be excluded due to other constraints. Under Constraint C_i, the allowed input behavior, B^I_{Ci}, is, thus, given by the following equation:

$$B^I_{Ci} = B^I_T - (E^I_P \cup E^I_{Ci}) \tag{1}$$

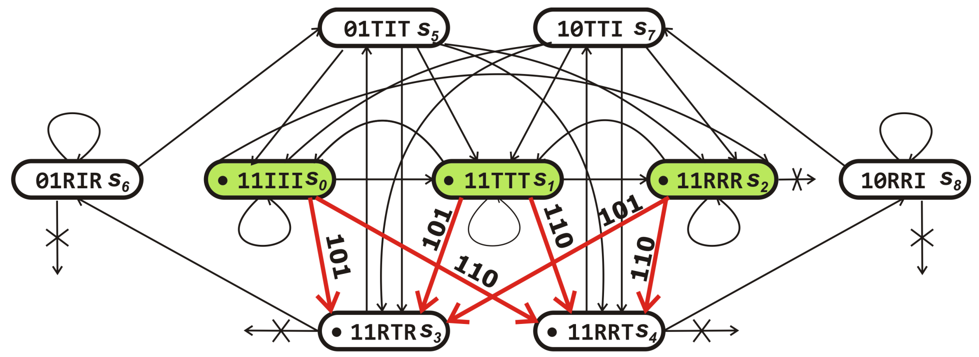


Fig. 3: EFork state diagram.

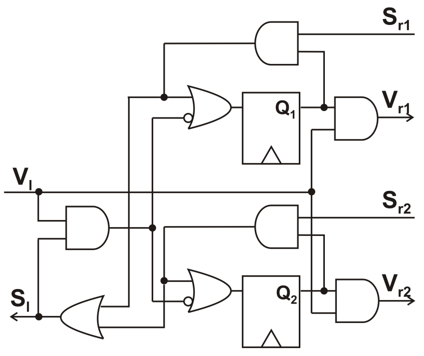


Fig. 4: An EFork.

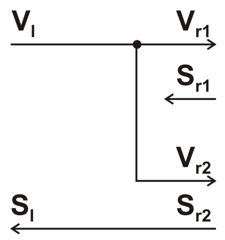


Fig. 5: A USFork.

V_l \ $S_{r1}S_{r2}$	00	01	11	10
0	0	0	0	0
1	1	-	1	-

Fig. 6: V_{r1} (same for V_{r2}) in states s_0 to s_2.

V_l \ $S_{r1}S_{r2}$	00	01	11	10
0	-	-	-	-
1	0	-	1	-

Fig. 7: S_l in states s_0 to s_2.

Where E^I_{Ci} is the set of sequences excluded from the input behavior for violating constraint C_i. The words property and constraint will be used interchangeably as long as the context is clear. In our notation, *constraint x* constrains the input behavior such that *property x* holds. Properties (and constraints) will be specified using PSL syntax unless mentioned otherwise.

Definition 1. Protocol Equivalence *Two forks are said to be SELF protocol equivalent (or, for short, just protocol equivalent), if, given the same input sequences, their terminal channels go through the same SELF state transitions.*

Theorem 1. *The EFork of Fig. 4 is protocol equivalent to the USFork of Fig. 5 if the fork input behavior is constrained such that the following property is TRUE in the former: ALWAYS $s_0|s_1|s_2$. Where s_i is 1 if the EFork is in state s_i for $\forall i \in \{0,1,2\}$ (Refer to Fig. 3).*

Proof: Figures 6 and 7 show the Karnaugh maps of V_{r1} (or V_{r2}) and S_l, respectively, in states s_0 - s_2. By using simple logic optimization, the following equations can be obtained:

$$V_{r1} = V_l,\ V_{r2} = V_l,\ S_l = S_{r1} \text{ or } S_l = S_{r2} \qquad (2)$$

The USFork of Fig. 5 exactly implements these equations. ■

Please notice that the choice to connect S_l to either S_{r1} or S_{r2} in Fig. 5 is irrelevant. The reason is, as will be shown in Theorem 2, under the input constraint specified in Theorem 1, S_{r1} and S_{r2} are always identical. They may differ only when V_l is Zero, in which case the L channel is in the idle (I) state whatever the value of S_l.

Definition 2. Equivalent Constraints *Referring to Equation 1, two constraints C_i and C_j are said to be equivalent if $B^I_{Ci} = B^I_{Cj}$. (i.e., the allowed input behavior under constraint i is the same as the allowed input behavior under constraint j.)*

In other words, two properties i and j (also referred to as constraints) are equivalent if, constraining the input behavior such that property i holds, will cause property j to hold, and vice versa.

Similarly, n properties (also referred to as constraints) are equivalent if $\forall i,j \in \{1,2,..,n\}$ property i and property j are equivalent.

Theorem 2. *The following three properties (also referred to as constraints) are equivalent:*

1) *ALWAYS $s_0|s_1|s_2$. Where s_i is 1 if the EFork is in state s_i $\forall i \in \{0,1,2\}$.*
2) *NEVER $V_l\&(S_{r1} xor S_{r2})$*
3) *ALWAYS $V_{r1} xnor V_{r2}$*

Proof: We will prove that constraining the input behavior such that one property holds will cause the other two to hold as well, and vice versa.

C. 1 If the input behavior is constrained such that EFork operates in s_0 to s_2 only, then properties 2 and 3 are satisfied as well. As shown in Table I, in s_0 to s_2, S_{r1} never differs from S_{r2} while V_l is One (C. 2), and V_{r1} is always the same as V_{r2} (C. 3).

C. 2 States s_0 to s_4 are reset states. However, if the input behavior is constrained such that S_{r1} is always the same as S_{r2} while V_l is one, then the EFork can reset only in any of the states s_0 to s_2, exclusively. Besides, it will stay in these states since all the red transitions in Fig. 3 will not fire. Hence, C. 1 will be satisfied, and subsequently, C. 3 will be satisfied as well.

C. 3 If the input behavior is constrained such that only those input sequences that cause V_{r1} to be always the same as V_{r2}, are allowed, then the EFork will never move to any of the states s_5 to s_8 (where V_{ri}s differ). Moreover, EFork will not reset in states s_3 or s_4 since all the input sequences that go through them must also go through states s_5 to s_8 (no other transition is available). And the latter sequences are excluded by the constraint. Hence, forcing C. 3 will cause the EFork to reset and operate in states s_0 to s_2 only. Therefore, both C. 1 and C. 2 will be satisfied.

■

Definition 3. Equivalence Constraint. *We call a constraint on the input behavior that causes EFork to be protocol equivalent to USFork an equivalence constraint.*

When the context is clear, an equivalence constraint will also be referred to as an equivalence *condition*.

Definition 4. Minimal Equivalence Constraint. *An equivalence constraint is minimal if it allows for maximum behavior of the inputs beyond which an EFork will fail to be protocol equivalent to a USFork.*

Theorem 3. *Each of the three constraints of Theorem 2 is minimal.*

Proof: If C. 1 is not minimal, then EFork is allowed to operate in other states beside s_0 to s_2 and still be protocol equivalent to the USFork. However, this is not the case. In states s_5 to s_8, Q_2 and/or Q_1 change from their reset values (which is "11"). Hence, if the EFork is allowed to operate in these states, then the USFork would need flip-flops to "memorize" their values, which is not the case. Similarly, if the EFork operates in states s_3 or s_4, it has no other legal transition but to move to one of the states s_5 to s_8 (which we argued break the protocol equivalence). Hence, C. 1 is a minimal constraint.

Since, from Theorem 2 the three constraints are equivalent. Therefore, they constrain the input behavior similarly. It follows that, since C. 1 is minimal, C. 2 and C. 3 are minimal as well.

■

To check for EFork replacements, the EFork can be checked against *any* of the three properties listed in this section. However, without loss of generality, only property 3 will be used, hereafter. Would two branches of an EFork satisfy property 3, the EFork can be correctly replaced by a USFork. Being a minimal *condition* for equivalence (as proven in Theorem 3), it maximizes the chance of finding candidate EForks for replacement.

Replacing an EFork with a USFork can not create combinational cycles, since there are no internal paths inside the USFork that connects valid to stall ports (or vice versa). This is an advantage over lazy forks where such internal paths do exist [8]. Besides, since (under the mentioned *conditions*) the USFork is protocol equivalent to the EFork, they both schedule the same protocol state transitions over their terminal channels. Hence, they will both have the same runtime. Finally, replacing an EFork with a USFork can never degrade the control network maximum frequency. It may actually boost it in some cases since the USFork cuts from the EFork internal path delays.

D. Verification

To verify Theorems 1 and 2, we use the setup of Fig. 8. The whole structure is modeled and passed to a symbolic model checker, NuSMV [11]. The inputs of both the EFork and USFork (i.e., V_l, S_{r1}, and S_{r2} are driven simultaneously from the same protocol terminal (PT). A PT can simply be an EB controller initialized in a random state. It can also be implemented as a SELF channel with protocol constraints forced on its valid and stall signals. In this Section the first approach is used, the other will be used later in the paper. The outputs of the EFork and USFork have suffixes of $\_E$ and $\_US$, respectively. They are ORed together to form the corresponding signals over the the three terminal channels (i.e., L, R_1, and R_2). Valid and stall signals on channel L will be denoted as VL and SL, respectively. Same for the other channels. For example, $VR1$ is the ORing of $VR1\_E$ and $VR1\_US$.

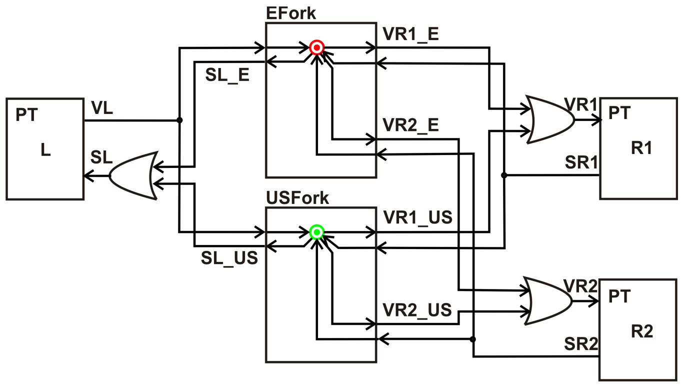


Fig. 8: Equivalence verification setup.

The shown blocks as well as a clock generator are all connected synchronously in NuSMV. The clock changes phase with every verification cycle. The I, T, and R states of the EFork L channel (denoted as L_E) are defined as follows:

```
DEFINE L_E_I := !VL_E;
DEFINE L_E_T := VL_E & !SL_E;
DEFINE L_E_R := VL_E & SL_E;
```

And on the USFork:

```
DEFINE L_US_I := !VL_US;
DEFINE L_US_T := VL_US & !SL_US;
DEFINE L_US_R := VL_US & SL_US;
```

The other states of the other 2 channels are defined similarly for both EFork and USFork. The EFork states of operation are also defined as follows:

```
-- s0 = 11III
DEFINE S0_E := EFork.q1 & EFork.q2 & L_E_I & R1_E_I &
R2_E_I;
-- s1 = 11TTT
DEFINE S1_E := EFork.q1 & EFork.q2 & L_E_T & R1_E_T &
R2_E_T;
-- s2 = 11RRR
DEFINE S2_E := EFork.q1 & EFork.q2 & L_E_R & R1_E_R &
R2_E_R;
-- s3 = 11RTR
DEFINE S3_E := EFork.q1 & EFork.q2 & L_E_R & R1_E_T &
R2_E_R;
-- s4 = 11RRT
DEFINE S4_E := EFork.q1 & EFork.q2 & L_E_R & R1_E_R &
R2_E_T;
```

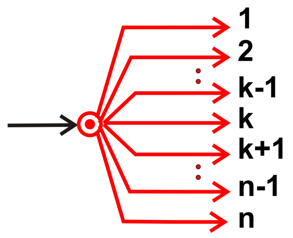
Mismatches over the three channels are defined as follows:

```
DEFINE L_MISMATCH := (L_E_I xor L_US_I) | (L_E_T xor L_US_T)
| (L_E_R xor L_US_R);
DEFINE R1_MISMATCH := (R1_E_I xor R1_US_I) | (R1_E_T xor
R1_US_T) | (R1_E_R xor R1_US_R);
DEFINE R2_MISMATCH := (R2_E_I xor R2_US_I) | (R2_E_T xor
R2_US_T) | (R2_E_R xor R2_US_R);
DEFINE MISMATCH := L_MISMATCH | R1_MISMATCH | R2_MISMATCH;
```

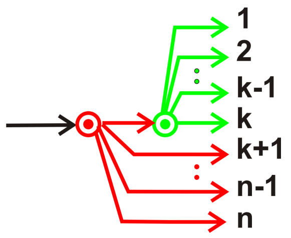
Finally, the three constraints (or properties) are defined as follows (without temporal qualifier):

```
DEFINE C_1 := S0_E | S1_E | S2_E;
DEFINE C_2 := VL & (SR1 xor SR2);
DEFINE C_3 := VR1_E xnor VR2_E;
```

A constraint is forced through the NuSMV INVAR reserved word, and a property is verified using PSLSPEC. In the following code,



(a) Eager fork (EFork).



(b) Hybrid fork (HFork).

Fig. 9: Eager to hybrid fork transformation.

only one constraint is forced at a time.

To verify Theorem 1:

```
INVAR C_1;
PSLSPEC never MISMATCH; -- True
```

Similarly, Theorem 2 Constraint. 1 is verified as follows:

```
INVAR C_1;
PSLSPEC never C_2; -- True
PSLSPEC always C_3; -- True
```

And Theorem 2 Constraint. 2:

```
INVAR !C_2;
PSLSPEC always C_1; -- True
PSLSPEC always C_3; -- True
```

And Constraint. 3:

```
INVAR C_3;
PSLSPEC always C_1; -- True
PSLSPEC never C_2; -- True
```

E. Multi-output-channel EForks

Theorem 6 extends the results of the previous theorems to multi-output-channel EForks.

Lemma 4. *n-output-channel EFork is protocol equivalent to concatenated (n-1) 2-output-channel EForks.*

Proof: Proof is trivial and omitted for space limitations. ∎

Lemma 5. *n-output-channel USFork is protocol equivalent to concatenated (n-1) 2-output-channel USForks.*

Proof: Proof is trivial and omitted for space limitations. ∎

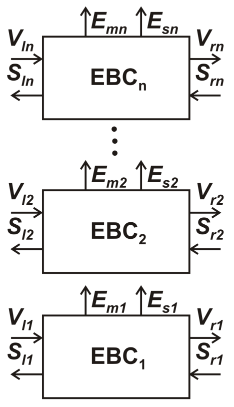
Theorem 6. *If, in Fig. 9, $\forall i, j \in \{1, 2, .., k\}$ the following property holds: ALWAYS $(V_{ri} xnor V_{rj})$, then the hybrid fork (HFork) of Fig. 9b is protocol equivalent to the eager fork (EFork) of Fig. 9a.*

Proof: The proof follows from Lemmas 4 and 5 and Theorem 2, and was omitted due to space limitations. ∎

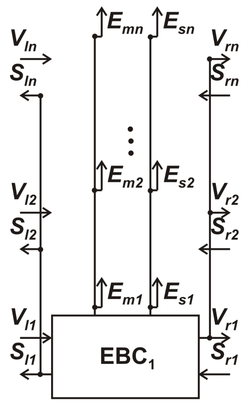
Red forks in Fig. 9 are EForks and green are USForks.

IV. Elastic Buffer Controller Merging

In a typical control network, some Elastic Buffer Controllers (EBCs) may activate their corresponding latches at similar schedules. This can allow for possible merging of these controllers into one controller that feeds them all (as much as the physical placement permits). Similar observation has also been noted by the authors of [10]. However, their algorithm requires the control network model to be closed and of static latency. This Section provides a framework for merging such controllers in any control network. That includes open networks (i.e., when the environment abstract is not available or required to be flexible) as well as networks incorporating variable latency units. The approach is straight forward, nonetheless.



(a) Before



(b) After

Fig. 10: EBC merging.

Definition 5. Functional Equivalence *Two structures are said to be functionally equivalent, if, given the same input sequences, they produce the same output sequences.*

Theorem 7. *If the n EBCs of Fig. 10a are initialized in the same state and the environment behavior is constrained such that the following two properties (also referred to as constraints) are true $\forall i, j \in \{1, 2, ..n\}, i \neq j$:*

1) *ALWAYS $(V_{li}$ xnor $V_{lj})$*
2) *ALWAYS $(S_{ri}$ xnor $S_{rj})$*

Then, the structure of Fig. 10b is functionally equivalent to the one in Fig. 10a.

Proof: Trivial. It is easy to show under the conditions of the theorem, that the following properties will also hold: ALWAYS $(V_{ri}$ xnor $V_{rj})$, ALWAYS $(S_{li}$ xnor $S_{lj})$, ALWAYS $(E_{mi}$ xnor $E_{mj})$, and ALWAYS $(E_{si}$ xnor $E_{sj})$ ∎

EBC merging is limited only by the physical placement constraints. A technique in which a maximum diameter per cluster of merged EBCs (*schedulers* in their case) was proposed in [10]. The same technique can be readily integrated in this approach.

V. Verification Models Of Different Control Network Components

An elastic control network needs to be verified as a whole to check if the required conditions for merging EBCs or using USForks are met. Two frameworks were particularly useful for us: namely, 6thSense [12] and NuSMV. The Section will try to cover both frameworks as space allows.

6thSense uses a standard VHDL to model a circuit and is particularly designed for synchronous circuit verification. Most of the control network models will be omitted since they are intuitive.

NuSMV model checker has its own input language and supports both synchronous and asynchronous circuit verification. To mimic a synchronous behavior in NuSMV, the network components (e.g., joins and forks), including a clock generator, are connected synchronously. All combinatorial logic are modeled with zero delay (using DEFINE reserved word), and the clock generator changes phase with every verification cycle. An NuSMV model for a clock generator is as follows:

```
MODULE ClkGenerator
VAR Clk:boolean;
ASSIGN
init(Clk) := 0;
```

```
next (Clk) := !Clk;
```

and for a D-FF (with a reset value of 1):

```
MODULE DFF1(Clk,D)
VAR Q:boolean;
ASSIGN
init(Q):= 1;
next(Q):= case
(Clk=0) & (next(Clk))=1: D;
1: Q;
esac;
```

A. *n-Input Join*

An NuSMV model for the n-input join structure used in [4] is as follows:

```
MODULE LJoinn(Vl1,Vl2,..Vln,Sr)
DEFINE Vr:= Vl1 & Vl2 & ... Vln;
DEFINE Sl1:= !(Vr & !Sr); ...
DEFINE Sln:= !(Vr & !Sr);
```

B. *n-Output Fork*

An NuSMV model for an n-output EFork is as follows:

```
MODULE EForkn(Clk,Vl,Sr1,Sr2,...Srn)
VAR
DFF_1: DFF1(Clk,d1); ...
DFF_n: DFF1(Clk,dn);
DEFINE d1 := (Sr1 & q1) | !(Vl & Sl) ;
DEFINE q1 := DFF_1.Q;
DEFINE Vr1 := Vl & q1; ...
DEFINE dn := (Srn & qn) | !(Vl & Sl) ;
DEFINE qn := DFF_n.Q;
DEFINE Vrn := Vl & qn;
DEFINE Sl := (Sr1 & q1) | (Sr2 & q2) | .. (Srn & qn);
```

USFork transformation Condition 3 of Theorem 2 is verified for each two branches in the EFork to determine if they can be replaced by a USFork. Hence, in an n-output EFork F and $\forall i, j \in \{1, 2, .., n\}, i \neq j$, the following properties are specified. In NuSMV:

```
DEFINE F_i_j_MISMATCH := Vri xor Vrj ;
PSLSPEC never F_i_j_MISMATCH;
```

And, in 6thSense (bil file):

```
[ fail; F_i_j; "F_i_j" ] <= Vri xor Vrj ;
```

C. *Elastic Buffer Controller*

Similarly, the EBC model immediately follows the FSM or the circuit implementation of [4]. The EBC merging condition of Theorem 7 is verified for each two EBCs in the network to determine if they can be merged. Hence, for a control network with n EBCs and $\forall i, j \in \{1, 2, .., n\}, i \neq j$, the following properties are specified. In NuSMV:

```
DEFINE EBC_i_j_MISMATCH := (Vli xor Vlj) | (Sri xor Srj) ;
PSLSPEC never EBC_i_j_MISMATCH;
```

And in 6thSense (bil file) as:

```
[ fail; EBC_i_j; "EBC_i_j" ] <= (Vli xor Vlj) or (Sri xor
Srj) ;
```

D. *SELF Input Channel*

A SELF input channel is the control channel corresponding to a data input (or group of data inputs) to the design. The valid signal of this channel Vi is an input to the design and the stall (Si) is an output. Vi will be defined as a *random* input with the SELF protocol constraints applied. In particular, SELF prohibits a transition from R to I state on any channel. This constraint on the input behavior is expressed in NuSMV as:

```
DEFINE InputChannel_i_Constraint := !(Vi) | !(Si) | Vi_next;
INVAR InputChannel_i_Constraint;
```

and in 6thSense (bil file) as:

```
[ constraint; InputChannel_i_Constraint ] <= not(Vi) or
not(Si) or Vi_next;
```

Where, in both cases, Vi is a one clock delayed version of Vi_next. Vi_next is, then, considered as the *virtual* input that the verification engine exhaustively randomizes.

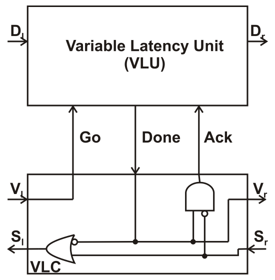


Fig. 11: A variable latency unit and a controller [4].

E. *SELF Output Channel*

Similarly, a SELF output channel is the control channel corresponding to a data output (or group of data outputs) from the design. The valid signal of this channel Vi is an output from the design and the stall (Si) is an input. The SELF protocol does not explicitly set constraints on the possible sequence of values over the input stall signal. However, it can be easily inferred from the EB specifications in [4] or the EHB (elastic half buffer) in [13] that a transition from $I0$ ($!V\&!S$) to $I1$ ($!V\&S$) can not happen on any SELF channel. Hence, the following constraint is applied to the SELF output channel. In NuSMV:

```
DEFINE OutputChannel_i_Constraint := Vi | Si | !(Si_next);
INVAR OutputChannel_i_Constraint;
```

and in 6thSense as:

```
[ constraint; OutputChannel_i_Constraint ] <= Vi or Si or
not(Si_next);
```

Again, Si is a one clock delayed version of the random input Si_next.

F. *Variable Latency Unit*

Fig. 11 shows a block diagram of a variable latency unit (VLU) and a variable latency controller (VLC) [4]. The VLC model follows the figure directly and omitted for brevity. The VLU model would depend on the actual block design. Nonetheless, to be able to verify the control network, it suffices to know the minimum and maximum latency values of that block (whatever its functionality is). Hence, to model the VLU, we used a model that randomly picks the next latency value from a range of values [min,max] specified by the designer for each block.

VI. HGEN TOOL

To automate the transformations described in this paper, we developed HGEN. HGEN (Hybrid network GENerator) is a fully automated tool that takes a verilog description of a control network and returns a verilog description of the minimized version. The tool currently uses 6thSense as the verification engine. Support for NuSMV is left for future versions. HGEN models the input verilog control network into VHDL. It adds the proper constraints for the

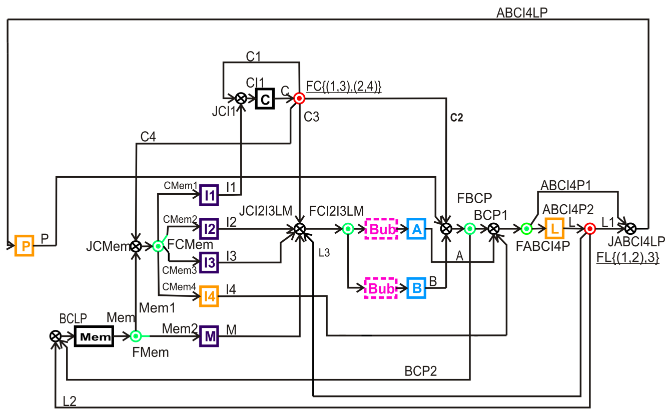


Fig. 12: Control network of the elastic clocked MiniMIPS with Register File bubbles. Image adapted from [9].

SELF channels. The EFork to USFork transformation conditions are verified for each 2 branches in the network. Similarly, the EB controller merging conditions are checked for each two EB controllers. HGEN automatically generates the suitable models for the variable latency units (based on the min and max latencies provided by the user in a configuration file). It generates a report with the EFork branches that has been transformed into USFork, and the merged EB controllers. *-nm* (*no merge*) option can be used to prevent HGEN from merging equivalent EB controllers (i.e., to only check for and do EFork to USFork transformation). The option is useful for doing the EBC merge after having some insight over the place and route information. HGEN currently supports all the network components described in Section V and more. Other component models (e.g., elastic half buffer and early evaluation components [14]) can be readily integrated.

VII. Results

All the examples of this paper have been automatically run by HGEN. Tables II and III show the number of checked properties per design as well as the tool runtime. In all the examples the runtime is within few minutes. The machine used has AMD Athlon(TM) 64 X2 Dual Core 3.2GHz Processor. Area and power are synthesis numbers. Design Compiler Ultra technology and IBM 65 nm library were used.

A. The MiniMIPS Processor

For the sake of comparison, we use the case study elasticized by the authors of [9] and [8], MiniMIPS. The MiniMIPS is an 8-bit subset of the 32-bit MIPS (Microprocessor without Interlocked Pipeline Stages) [15], [16]. A simple block diagram of the architecture can be found in [8]. In summary, MiniMIPS has 10 EBCs (shown in solid rectangles in Fig. 12, excluding Mem): P to control the 8-bit program counter, C to control the 4-bit (data) controller, $I1 - I4$ to control the 8-bit 4 instruction registers, respectively, A and B to control the ALU two 8-bit input registers, L to control the ALU 8-bit output register, and M to control the 8-bit memory data register. From the control point of view, the register file R and the memory Mem could be considered as combinational units [4] (as long as timing allows). Hence, adding separate EBCs for R and Mem are optional.

To illustrate the capability of the proposed approach, we like to study the MiniMIPS in three different settings:

1) Register File Bubbles: In this setting the control network is closed. We add one bubble stage at the two outputs of the register file (shown in dotted rectangles in Fig. 12). In practice this can be done to accommodate a high latency register file or because of long wires. The resultant control network verilog is passed to HGEN. Row one of Table II shows the results. In this setting, 10 out of the 12 EForks can be replaced by USForks (Fig. 12 shows EForks in red and USForks in green). This achieves 34.3% area reduction, and 25.4% and 32% dynamic and leakage power savings, respectively.

The testbench program of [16] was run on the two versions of the elastic MiniMIPS: the one with all forks implemented as EForks, and the other after USFork replacements. As expected, both versions finished the program after the same number of clock cycles (147 cycles in this case).

Furthermore, if the chip physical placement allows, 7 out of the 12 EBCs (10 + 2 bubbles) can be omitted (i.e., merged with other EBCs). This can achieve 63.0% area reduction, and 53.4% and 54.8% dynamic and leakage power, respectively. Since merging some equivalent EBCs may not be feasible because of the design timing constraints, the actual saving of area and power will be in between the above two limits (i.e., depending on how many EBCs can actually be merged).

2) Variable Latency ALU: In this setting, the control network is closed, and there are no bubbles at the register file outputs. The ALU is modeled with a variable latency unit that finishes an operation in one or two clk cycles. Row two of Table II shows the results. In this setting, 9 out of the 12 EForks can be replaced by USForks. This achieves 32.3% area reduction, and 30.5% and 25.9% dynamic and leakage power savings, respectively. Similarly, the table also shows the area and power savings in case the physical placement allows for merging 7 out of the 10 EBCs.

3) Off-Chip Memory With Unknown Latency: In this setting, the control network is open at the memory interface. The memory interface is modeled in HGEN by one input and output SELF channels. In practice this can be done if the actual latency of the memory is unknown or required to be flexible. Row three of Table II shows the results. In this setting, 7 out of the 12 EForks can be replaced by USForks. This achieves 25.6% area reduction, and 22.8% and 22.2% dynamic and leakage power savings, respectively. Similarly, the table also shows the area and power savings in case the physical placement allows for merging 5 out of the 10 EBCs.

B. s382

S382 is one of the ISCAS benchmarks. It has 3 input channels: F, T, and C, and 6 output channels: Y2, Y1, R2, R1, G2, and G1, and 21 EBCs. Table III shows the results of running HGEN over s382 in 3 different *incremental* settings:

1) All the 9 input/output channels are left open.
2) Y2 is connected to F, and Y1 is connected to T. The other 5 input/output channels are left open.
3) Y2 is connected to F, and Y1 is connected to T. R2 and R1 and G2 are connected to C through a 3-input join followed by a bubble. Output channel G1 is left open.

Intuitively, the input behavior of setting 3 is a subset of 2, which, in turn, is a subset of 1. Hence, the number of EForks that can be replaced by USFork is the same or increases as we go from setting 1 to setting 3. Though the proposed approach handles open and closed control networks, however, this example shows that the chance of finding candidate EForks for replacement increases as we know more about the environment. In s382, the reduction in the number of EForks is 32%, 36%, and 72% in settings 1, 2, and 3, respectively.

Finally, Table IV shows HGEN results for other ISCAS benchmarks - verified in *totally open* control network settings (i.e., no abstract for the environment is provided). The results emphasize the

TABLE II: HGEN results for the MiniMIPS processor elastic control network. Power is computed at 4 ns clock period.

			The Original Control Network				HGEN Step 1				HGEN Step 2			
Design	#I	#O	Total # EForks	Total # EBCs	Area (μ^2)	Power (μW) $\frac{P_{dyn}}{P_{leakage}}$	# Replaced EForks	Area (μ^2)	Power (μW) $\frac{P_{dyn}}{P_{leakage}}$	$\frac{\#Properties}{Time(s)}$	# Merged EBCs	Area (μ^2)	Power (μW) $\frac{P_{dyn}}{P_{leakage}}$	$\frac{\#Properties}{Time(s)}$
MiniMIPS - 1	0	0	12	12	834.0	$\frac{159.2}{3.1}$	10	547.8	$\frac{118.7}{2.1}$	$\frac{19}{0.52}$	7	308.4	$\frac{74.2}{1.4}$	$\frac{65}{0.64}$
MiniMIPS - 2	0	0	12	10	754.2	$\frac{108.3}{2.7}$	9	510.6	$\frac{75.3}{2.0}$	$\frac{19}{0.7}$	7	278.4	$\frac{40.1}{1.2}$	$\frac{64}{0.85}$
MiniMIPS - 3	1	1	12	10	754.2	$\frac{110.5}{2.7}$	7	561.0	$\frac{85.3}{2.1}$	$\frac{19}{20.56}$	5	394.8	$\frac{60.8}{1.6}$	$\frac{64}{10.46}$

TABLE III: HGEN results for s382 benchmark.

Design	#I	#O	Total # EForks	Total # EBCs	# Repl. EForks	# Merg. EBCs	$\frac{\#Propert.}{Time(s)}$
s382 - 1	3	6	25	21	8	7	$\frac{255}{20.1}$
s382 - 2	1	4	25	21	9	8	$\frac{255}{375.22}$
s382 - 3	0	1	25	22	18	17	$\frac{255}{152.29}$

TABLE IV: HGEN results ror other ISCAS benchmarks - in *open* network settings.

Design	#I	#O	Total # EForks	Total # EBCs	# Repl. EForks	# Merg. EBCs	$\frac{\#Propert.}{Time(s)}$
s27	7	4	3	3	1	1	$\frac{7}{0.79}$
s298	17	20	25	14	2	2	$\frac{131}{5.37}$
s344	9	11	32	15	2	2	$\frac{177}{89.81}$
s386	7	7	15	6	2	2	$\frac{40}{1.49}$
s1488	8	19	32	6	5	5	$\frac{102}{4.56}$

speed of the tool. Further savings in the number of EForks and EBCs can be achieved with more knowledge of the environment model.

VIII. Conclusion

The paper demonstrated that eager forks (EForks) can be redundant in the control network of synchronous elastic circuits. We introduced an ultra simple fork (USFork) implementation. Conditions for replacing an EFork with a USFork were formally derived. The replacement does not cause any combinational cycles, does not degrade runtime nor the network maximum operational frequency. Rather, it has the potential of saving substantial amount of area and power consumption. The paper also investigated the conditions under which multiple SELF controllers can be merged to further decrease the area and power overhead (as long as the physical placement allows). The flow was integrated in a fully automated tool, HGEN. HGEN showed superior results in MiniMIPS case study as well as some ISCAS benchmarks. We showed that the chance of finding candidate EForks for replacement increases as we know more about the environment model. Nonetheless, the proposed approach handles control networks that are open or closed, with static or variable latencies. Finally, for all the paper examples, HGEN runs within few minutes, making it suitable for industrial use.

References

[1] L. Carloni, K. Mcmillan, and A. L. Sangiovanni-VincentelliR, “Theory of latency insensitive design,” in *IEEE Transactions on CAD of Integrated Circuits and Systems*, vol. 20, no. 9, Sep 2001, pp. 1059–1076.

[2] L. Carloni and A. Sangiovanni-Vincentelli, “Coping with latency in soc design,” *Micro, IEEE*, vol. 22, no. 5, pp. 24–35, Sep/Oct 2002.

[3] A. Gotmanov, M. Kishinevsky, and M. Galceran-Oms, “Evaluation of flexible latencies: designing synchronous elastic h.264 cabac decoder.” in *The Problems in design of micro- and nano-electronic systems*, 2010.

[4] J. Cortadella, M. Kishinevsky, and B. Grundmann, “Synthesis of synchronous elastic architectures,” in *ACM/IEEE Design Automation Conference*, July 2006, pp. 657–662.

[5] J. Carmona, J. Cortadella, M. Kishinevsky, and A. Taubin, “Elastic circuits,” *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 28, no. 10, pp. 1437–1455, Oct. 2009.

[6] H. M. Jacobson, P. N. Kudva, P. Bose, P. W. Cook, S. E. Schuster, E. G. Mercer, and C. J. Myers, “Synchronous interlocked pipelines,” in *8th International Symposium on Asynchronous Circuits and Systems*, Apr. 2002, pp. 3–12.

[7] S. Krstic, J. Cortadella, M. Kishinevsky, and J. O’Leary, “Synchronous elastic networks,” in *Formal Methods in Computer Aided Design, 2006. FMCAD ’06*, Nov. 2006, pp. 19–30.

[8] E. Kilada, S. Das, and K. Stevens, “Synchronous elasticization: Considerations for correct implementation and minimips case study,” in *VLSI System on Chip Conference (VLSI-SoC), 2010 18th IEEE/IFIP*, sept. 2010, pp. 7 –12.

[9] E. Kilada and K. Stevens, “Control network generator for latency insensitive designs,” in *Design, Automation Test in Europe Conference Exhibition (DATE), 2010*, 8-12 2010, pp. 1773 –1778.

[10] J. Carmona, J. Julvez, J. Cortadella, and M. Kishinevsky, “Scheduling synchronous elastic designs,” in *Application of Concurrency to System Design, 2009. ACSD ’09. Ninth International Conference on*, july 2009, pp. 52 –59.

[11] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella, “Nusmv 2: An opensource tool for symbolic model checking.” in *Proc. of 14th Conf. on Computer Aided Verification (CAV 2002)*, vol. 2404, July 2002.

[12] 6thSense. Http://www.research.ibm.com/sixthsense/.

[13] G. Hoover and F. Brewer, “Synthesizing synchronous elastic flow networks,” in *Design, Automation and Test in Europe, 2008. DATE ’08*, 10-14 2008, pp. 306 –311.

[14] J. Julvez, J. Cortadella, and M. Kishinevsky, “Performance analysis of concurrent systems with early evaluation,” in *Computer-Aided Design, 2006. ICCAD ’06. IEEE/ACM International Conference on*, nov. 2006, pp. 448 –455.

[15] J. H. et al., “The MIPS Machine,” in *COMPCON*, 1982, pp. 2–7.

[16] N. Weste and D. Harris, *CMOS VLSI design: a circuit and systems perspective*, 2004.