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

Eliyah Kilada and Kenneth S. Stevens
University of Utah
ealiyah.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].

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 cannot 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 ⊙ and ⊗, 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'.

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 and hybrid SELF.

B. Eager Fork State Diagram

A 2-output-channel EFork has 3 terminal channels: namely, L (Left), R1 (Right1) and R2 (Right2). L channel consists of signals V1 and S1. Similarly, R1 consists of V1 and S1, and R2 of V2 and S2.

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 V1, S1, and S2 are part of the state vector (along with the flip-flop outputs, Q1 and Q2). To simplify the notations, the state vector takes the following format: <Q1, Q2, L, R1, R2>, where L, R1 and R2 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 <V1, S1, S2> ∈ {0, 1}^3. Subscript n is added to I and the 3 signals to denote the value at clock cycle n. We define S'1, to be an infinite sequence of input vectors ordered by the clock index. Hence, S'1[n] = I_n. We refer to the total input behavior, B'1, 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: <0, 0, 0>, <1, 1, 1>, ... >. The set of all sequences which are excluded for violating the SELF protocol will be denoted as E'1. Nonetheless, in this Section, some of the sequences will also be excluded due to other constraints. Under Constraint C1, the allowed input behavior, B'C1, is, thus, given by the following equation:

\[ B'C1 = B'1 - (E'1 \cup E'C1) \]  

TABLE I: EFork state table.
The USFork of Fig. 5 exactly implements these equations.

Please notice that the choice to connect $S_1$ to either $S_{11}$ or $S_{12}$ in Fig. 5 is irrelevant. The reason is, as will be shown in Theorem 2, under the input constraint specified in Theorem 1, $S_{11}$ and $S_{12}$ are always identical. They may differ only when $V_i$ is Zero, in which case the $L$ channel is in the idle ($I$) state whatever the value of $S_1$.

**Definition 2.** Equivalent Constraints Referring to Equation 1, two constraints $C_i$ and $C_j$ are said to be equivalent if $B_{i,j} = B_{j,i}$.

(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 $\forall i, j \in \{1, 2, \ldots, n\}$ 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 $V_i!(S_{1} \text{ xor } S_{2})$

3) ALWAYS $V_{1} \text{ xor } V_{2}$

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 $s_2$, $S_{11}$ never differs from $S_{12}$ while $V_i$ is One (C. 2), and $V_{1}$ is always the same as $V_{2}$ (C. 3).

**C. 2** States $s_0$ to $s_4$ are reset states. However, if the input behavior is constrained such that $S_{11}$ is always the same as $S_{12}$ while $V_i$ 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 $V_i$ to be always the same as $V_{2}$, are allowed, then the EFork will never move to any of the states $s_5$ to $s_8$ (where $V_i, s$ 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 s0 to s2 and still be protocol equivalent to the USFork. However, this is not the case. In states s5 to s8, Q2 and/or Q1 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 s3 or s4, it has no other legal transition but to move to one of the states s5 to s8 (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 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., VL, SL1, and SL2) 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 three terminal channels (i.e., L, R1, and R2). 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.

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 LE) 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;
```

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);
```

Mismatches over the three channels 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 xor VR2_E;
```

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

To verify Theorem 1:

\[
\begin{align*}
\text{INVAR } & C_{-1}; \\
\text{PSLSPEC } & \text{never MISMATCH; } \quad \text{-- True}
\end{align*}
\]

Similarly, Theorem 2 Constraint 1 is verified as follows:

\[
\begin{align*}
\text{INVAR } & C_{-1}; \\
\text{PSLSPEC } & \text{never } C_{2}; \quad \text{-- True}
\end{align*}
\]

And Theorem 2 Constraint 2:

\[
\begin{align*}
\text{INVAR } & C_{2}; \\
\text{PSLSPEC } & \text{always } C_{3}; \quad \text{-- True}
\end{align*}
\]

And Constraint 3:

\[
\begin{align*}
\text{INVAR } & C_{3}; \\
\text{PSLSPEC } & \text{always } C_{1}; \quad \text{-- True}
\end{align*}
\]

**E. Multi-output-channel E Forks**

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

**Lemma 4.** n-output-channel E Fork is protocol equivalent to concatenated (n-1) 2-output-channel E Forks.

**Proof:** Proof is trivial and omitted for space limitations. □

**Lemma 5.** n-output-channel US Fork is protocol equivalent to concatenated (n-1) 2-output-channel US Forks.

**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 \((V_{i_{1}} \text{xnor} V_{j_{1}})\), then the hybrid fork (H Fork) of Fig. 9b is protocol equivalent to the eager fork (E Fork) of Fig. 9a.

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

**IV. ELASTIC BUFFER CONTROLLER Merging**

In a typical control network, some Elastic Buffer Controllers (EBCs) may activate their corresponding latches at similar schedules. This allows 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 straightforward, nonetheless.
An n-output EFork is as follows:

\[
\text{DEFINE } \text{Sl} := (\text{Sr1} \& q1) | (\text{Sr2} \& q2) | \ldots (\text{Srn} \& qn);
\]

\[
\text{DEFINE } \text{Vrn} := \text{Vl} \& qn;
\]

\[
\text{DEFINE } qn := \text{DFF_n.Q};
\]

\[
\text{DEFINE } d_n := (\text{Srn} \& qn) | \neg(\text{Vl} \& \text{Sl}) ;
\]

\[
\text{DEFINE } \text{Vr1} := \text{Vl} \& q1; \ldots
\]

\[
\text{DEFINE } q1 := \text{DFF_1.Q};
\]

\[
\text{DFF}_n: \text{DFF1(Clk,} d_n); \ldots
\]

\[
\text{DFF}_1: \text{DFF1(Clk,} d_1); \ldots
\]

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

\[
\text{module EForkn}(\text{Clk,} \text{Vl,} \text{Sr1,} \text{Sr2,} \ldots \text{Srn})
\]

VARIABLE

\[
\text{VAR}
\]

\[
\text{next (} \text{Clk} \text{)} := \neg \text{Clk};
\]

\[
\text{and for a D-FF (with a reset value of 1)}:
\]

\[
\text{module DFF1(} \text{Clk,} \text{D)}
\]

VARIABLE

\[
\text{VAR}
\]

\[
\text{q boolean;}
\]

\[
\text{module DFF1(} \text{Clk,} \text{D)}
\]

A. n-Input Join

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

\[
\text{module LJoinn}(\text{Vl1,} \text{Vl2,} \ldots \text{Vln,} \text{Sr})
\]

DEFINE

\[
\text{Vr := Vl1 & Vl2 & \ldots Vln;}
\]

\[
\text{DEFINE } \text{Sr1} := \neg(\text{Vr} \& \neg\text{Sr});
\]

\[
\text{ DEFINE } \text{Sr2} := \neg(\text{Vr} \& \neg\text{Sr});
\]

\[
\text{and in 6thSense (bil file) as:}
\]

\[
\text{PSLSPEC never EBC_i_j_MISMATCH;}
\]

\[
\text{DEFINE EBC_i_j_MISMATCH := (Vli xor Vlj) | (Sri xor Srj) ;}
\]

\[
\text{And in 6thSense (bil file) as:}
\]

\[
\text{[ constraint; InputChannel_i_Constraint ] <= not(Vi) or not(Si) or Vi_next;}
\]

\[
\text{Where, in both cases, } Vi \text{ is a one clock delayed version of } Vi_{\text{next}}.
\]

\[
\text{Vi_{next} is, then, considered as the virtual input that the verification engine exhaustively randomizes.}
\]

\[
\text{F. Variable Latency Unit}
\]

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

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 \( I0 (Vi \& Si) \) to \( I1 (V & S) \) can not happen on any SELF channel. Hence, the following constraint is applied to the SELF output channel. In NuSMV:

\[
\text{DEFINE OutputChannel_i_Constraint := Vi | Si | Vi_{\text{next}};}
\]

\[
\text{INVAR OutputChannel_i_Constraint;}
\]

\[
\text{and in 6thSense as:}
\]

\[
\text{[ constraint; OutputChannel_i_Constraint ] <= not(Vi) or not(Si) or Vi_next;}
\]

\[
\text{Again, } Si \text{ is a one clock delayed version of the random input } Si_{\text{next}}.
\]

\[
\text{VI. HGEN TOOL}
\]

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
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 have been transformed into USFork, and the merged EB controllers. 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) can be described in Section V and more. Other component models (e.g., elastic half buffer and early evaluation components) 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 $M$ could be considered as combinational units [4] (as long as timing allows). Hence, adding separate EBCs for $R$ and $M$ 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.

<table>
<thead>
<tr>
<th>Design</th>
<th>#</th>
<th>#O</th>
<th>Total # EForks</th>
<th>Total # EBCs</th>
<th>Area (μ²)</th>
<th>Power (μW)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>P_{dyn}</td>
<td>P_{leak}</td>
<td># Replaced</td>
<td>P_{dyn}</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>/ P_{leakage}</td>
<td>/ P_{leakage}</td>
<td></td>
<td>Time(s)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MiniMIPS - 1</td>
<td>1</td>
<td>0</td>
<td>10</td>
<td>754.2</td>
<td>110.5</td>
<td>2.7</td>
</tr>
<tr>
<td>MiniMIPS - 2</td>
<td>1</td>
<td>0</td>
<td>10</td>
<td>754.2</td>
<td>110.5</td>
<td>2.7</td>
</tr>
<tr>
<td>MiniMIPS - 3</td>
<td>1</td>
<td>0</td>
<td>10</td>
<td>754.2</td>
<td>110.5</td>
<td>2.7</td>
</tr>
</tbody>
</table>

TABLE III: HGEN results for s382 benchmark.

<table>
<thead>
<tr>
<th>Design</th>
<th>#</th>
<th>#O</th>
<th>Total # EForks</th>
<th>Total # EBCs</th>
<th>Area (μ²)</th>
<th>Power (μW)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>P_{dyn}</td>
<td>P_{leak}</td>
<td># Replaced</td>
<td>P_{dyn}</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>/ P_{leakage}</td>
<td>/ P_{leakage}</td>
<td></td>
<td>Time(s)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s382 - 1</td>
<td>1</td>
<td>0</td>
<td>10</td>
<td>754.2</td>
<td>110.5</td>
<td>2.7</td>
</tr>
<tr>
<td>s382 - 2</td>
<td>1</td>
<td>0</td>
<td>10</td>
<td>754.2</td>
<td>110.5</td>
<td>2.7</td>
</tr>
<tr>
<td>s382 - 3</td>
<td>1</td>
<td>0</td>
<td>10</td>
<td>754.2</td>
<td>110.5</td>
<td>2.7</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Design</th>
<th>#</th>
<th>#O</th>
<th>Total # EForks</th>
<th>Total # EBCs</th>
<th>Area (μ²)</th>
<th>Power (μW)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>P_{dyn}</td>
<td>P_{leak}</td>
<td># Replaced</td>
<td>P_{dyn}</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>/ P_{leakage}</td>
<td>/ P_{leakage}</td>
<td></td>
<td>Time(s)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s27</td>
<td>7</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>s298</td>
<td>17</td>
<td>20</td>
<td>25</td>
<td>14</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>s344</td>
<td>9</td>
<td>11</td>
<td>32</td>
<td>15</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>s386</td>
<td>7</td>
<td>7</td>
<td>15</td>
<td>6</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>s447</td>
<td>8</td>
<td>19</td>
<td>32</td>
<td>6</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

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