Synchronous Elasticization: Considerations For Correct Implementation and MiniMIPS Case Study

Eliyah Kilada, Shomit Das, Kenneth Stevens
University of Utah
{Eliyah.Kilada, Shomit.Das}@utah.edu, kstevens@ece.utah.edu

Abstract—Latency insensitivity is a promising design paradigm in the nanometer era since it has potential benefits of increased modularity and robustness to variations. Synchronous elasticization is one approach (among others) of transforming an ordinary clocked circuit into a latency insensitive design. This paper presents practical considerations of elasticizing reconvergent fanouts. It also investigates the suitability of previously published as well as new join and fork implementations for usage in the elastic control network. We demonstrate that elasticization comes at a cost. Measurements of a MiniMIPS processor fabricated in a 0.5 μm node show that elasticization results in area and dynamic and leakage power penalties of 29%, 13% and 58.3%, respectively, without any loss in performance. These measurements do not exploit the capability of pipeline bubbles that occur if one needs to have unpredictable interface latency, or to insert extra bubbles into a pipeline due to wire delays. We finally show the architectural performance advantage of eager over lazy protocols in the presence of bubbles in the MiniMIPS.

Index Terms—SOC, Synchronous Elasticity, Latency Insensitive Design, MiniMIPS.

I. INTRODUCTION

Any system on chip (SOC) consists of a set of modules that not only perform individual functions, but also need to communicate with other modules. This communication may be in the form of either data or control signals. Sometimes, the communicating modules may be quite distant on the chip. The conventional synchronous circuit reads inputs and writes outputs at every clock cycle. The basic assumption is that computation and data transfer must be completed in exactly one clock cycle [1]. This places a very rigid timing constraint on the system, especially in the case where very long interconnects are involved. It is well known that metal wires do not scale as well as semiconductor gates. In fact, the resistance capacitance (RC) delay for metal wires per unit length is degrading with every new process generation [2], [3], [4]. This, in turn, increases the percentage of the clock cycle consumed in interconnect delays. Interconnect latency is affected by its dimensions as well as metal layers used, crosstalk (mainly determined by interconnect position in the chip), and power supply drop variations, among other layout specific factors [4]. This means that accurately evaluating the actual latency between modules is an issue that can only be resolved late in the design process. Moreover, over compensating for the delay due to wire effects can lead to sub-optimal designs [4].

Latency insensitive (LI) design tackles this issue by separating the computation and communication paradigms by creating a flexible protocol for communication between modules [5]. LI designs are able to tolerate interconnect latency variations without affecting the system functionality. This facilitates IP reuse and also ensures that system functionality depends only on modular correctness and not on the timing of the communication channels.

Synchronous elasticization is one approach (among others) of transforming an ordinary clocked circuit into an LI design [6], [7], [8]. This paper focuses on a clocked elastic protocol called the Synchronous Elastic Flow (SELF) protocol [6]. Synchronous elastic circuits are comparable to asynchronous designs in that they are robust to arbitrary channel latencies. Besides, they can be easily designed with conventional design flows using static timing analysis [7], [1].

A. Contribution

Practical considerations of elasticizing reconvergent fanouts are introduced. The suitability of different published, as well as new, join and fork implementations for usage in the elastic control networks are investigated. The cost of elasticizing a small microprocessor design is measured from fabricated chips, comparing area and power penalties. The performance of the elasticized MiniMIPS microprocessor is demonstrated when bubbles are inserted into the pipeline for different lazy and eager fork and join protocols.

II. SYNCHRONOUS ELASTIC ARCHITECTURES

An elastic system comprises elastic modules and elastic channels. Elastic modules are implemented using Elastic Buffers (EB) as counterparts to the flip-flops in conventional clocked systems. There are two parts to an EB implementation (Fig. 1), namely, data and control planes. EBs can be broken down into two Elastic Half Buffers (EHB) the same way a flip-flop can be replaced by a pair of master-slave latches [8].

An elastic channel uses a pair of control wires, ‘valid’ in the forward direction and ‘stall’ in the backward direction. These wires implement handshaking between the source and destination entities similar to the request/acknowledge wires in asynchronous systems [9]. Valid is asserted by the source when it holds valid data. Stall is asserted by the destination when it is not able to receive the data. Hence, in the SELF protocol, the two control signals valid (V) and stall (S) determine three possible channel states [6]:

- **Transfer (V & IS):** The source sends valid data and the destination can receive it.
- **Idle (IV):** The source does not send valid data.
- **Retry (V & S):** The source sends valid data, but the destination can not receive it. The source sustains the valid data until the destination is able to store the data.
In this section, we present some existing, as well as, new fork and join designs and discuss the correctness of combining them during an elasticization flow.

**A. Lazy Fork**

The lazy fork does not propagate a (valid) data token from its stem to its ready branches until all its branches are ready to receive it. We...
say a branch is ready if its corresponding stall signal is zero. Besides, the stem stalls if any of the branches stalls.

The lazy fork can be implemented in different ways. Fig. 4a shows an implementation of \textit{LFork} similar to the one reported in [6], [8]. We propose another implementation of the lazy fork in Fig. 4b. We refer to it as \textit{LKFork}. The valid signal on channel \( i \), \( V_{ri} \), of the 1-to-n LKFork is given by the following equation:

\[
V_{ri} = V_i \cdot \prod_{j=1, j \neq i}^{n} S_{rj} \tag{1}
\]

while the stall on the source channel, \( S_i \), is defined as follows:

\[
S_i = \sum_{i=1}^{n} S_{ri} \tag{2}
\]

Unlike the LFork, there is no internal path inside the LKFork that connects the stall signal of a branch to its corresponding valid signal. This advantage can substantially reduce the number of combinational cycles formed when lazy forks and joins are connected together. Combinational cycles are discussed in Section IV-D.

\subsection*{B. Eager Fork}

The eager fork (\textit{EFork}) will propagate valid tokens independently on each output channel, while stalling the input channel, until all output channels have accepted the data. Thus, when a (valid) data token is available at EFork stem, it will immediately pass the token to all its branches that are ready (i.e., not stalled), giving them an early start. Fig. 5 shows an \( n \) output extension of the EFork in [6]. Because of the early start provided by EFork, it may result in some performance advantage on the architecture level. However, this comes at the expense of more area and power consumption.

\subsection*{C. Lazy Join}

The lazy join must wait for all its input channels to carry valid data until it asserts its output channel valid signal. Fig. 6a shows an \( n \)-input lazy join (we refer to it as \textit{LJoin}) proposed in [8]. Note that LJoin doesn’t propagate a stall backward on any of its input channels.

Fig. 3. Control channels optimization example.

Fig. 4. Two lazy fork implementations.

Fig. 5. A 1-to-n eager fork \textit{EFork}.

Fig. 6. Two lazy join implementations.
shown in Fig. 9. It can be easily shown that if VA circuit implementation of this connection using LFork and LKJoin is suitable for use in the SELF protocol.

synchronous elastic design. Hence, this combination is not generally the combination can cause a deadlock in the control network of a

oscillation inside the loop as well as on the loop (shown in dotted lines in Fig. 8). The loop is logically unstable since it has an odd number of inverting elements. This results in an logically unstable loop can be formed. This occurs when a logical

The circuit implementation of this connection using LFork and LJoin is shown in Fig. 8. Assume that elastic buffer C holds a bubble (i.e., its output valid signal is zero), while A holds data. Assume also that SA2 is zero (B is not stalled). This connection will form a loop (shown in dotted lines in Fig. 8). The loop is logically unstable since it has an odd number of inverting elements. This results in an oscillation inside the loop as well as on the SA wire.

2) LFork-LKJoin: If an LFork branch feeds an LJoin input, the combination can cause a deadlock in the control network of a synchronous elastic design. Hence, this combination is not generally suitable for use in the SELF protocol.

We use the same block diagram of Fig. 7 to illustrate the point. The circuit implementation of this connection using LFork and LKJoin is shown in Fig. 9. It can be easily shown that if VA is zero, VA1 must also be zero and VAC be zero. This will force SA1 to be one, SA to be one and VA1 to be zero. Apparently, the loop shown in dotted lines forms a latch, since all its wires can simultaneously carry controlling values to all the gates in the loop. Hence, after a zero on VA, the system will deadlock. VA2, VAC, SC and SA will be stuck at zero, zero, one and one, respectively.

3) LKFork-LKJoin: Through similar argument to Section IV-D2, we can show that LKFork-LKJoin combination can result in a deadlock. Consider, for example, the network structure of Fig. 10. Assume that we use LKFork and LJoin to implement all the forks and joins in the figure. The boxes above the channel line represent valid signal values, and below are stall values. Once VA is zero, VA1, VA2, VAB, VAC, VAB2, VAC1 and VABC will also be zero. This, in turn, will cause SAB2 and SAC1 to be one (through join JABC) and SAB, SAC, SA1, SA2 and SA to be asserted. Again, the dotted loop will stick at these values whatever the control network inputs are. Hence, this combination is not generally suitable for use in the SELF protocol.

4) LKFork-LJoin: In all the network structures we examined, the loops formed by LKFork-LJoin combination are logically stable (i.e., have even number of inverting elements). However, since there are no state holding elements (e.g., flip-flops) in these loops, any glitches can dangerously oscillate back and forth in a complex control network. Consider for example the LJoin in Fig. 6a. If Sl does a zero to one transition while all the other input valids are one. Without appropriate delay matching inside the join, a short positive glitch will propagate on Sl. Besides, if zero and one values are simultaneously injected at different points in a loop, the two values can race around that loop causing oscillation. We also noticed that deadlock can occur under certain values of the input arrival times and gate delays in some structures. Research is in progress to determine the sufficient timing constraints and device sizing that can eliminate such oscillations and possible deadlocks.

5) EFork-LJoin/LKJoin: Eager forks inherently cut the cycles through the flip-flops used at each of its outputs. Hence, they do not suffer from any of the loop problems mentioned above. An EFork-LJoin can show slight area and power advantage over an EFork-LJoin (due to the removal of one AND gate from each join input channel).

V. CASE STUDY: MINIMIPS

MIPS (Microprocessor without Interlocked Pipeline Stages) is a 32-bit architecture, first designed by Hennessy [10]. The MiniMIPS is an 8-bit subset of MIPS, fully described in [11]. The MiniMIPS uses an 8-bit datapath. Only 8 registers are implemented and the program counter is also 8-bit long.

The MiniMIPS is used for a case study of elasticization. Fig. 11 shows a block diagram of the ordinary clocked MiniMIPS. The MiniMIPS has a total of 12 synchronization points (i.e., registers), shown as rectangles in Fig. 11: P (program counter), C (controller),
TABLE I
CLOCKED AND ELASTIC MINIMIPS CHIPS RESULTS. MEASUREMENTS ARE DONE AT 5V AND 30°C

<table>
<thead>
<tr>
<th></th>
<th>Clocked MiniMIPS</th>
<th>Elastic MiniMIPS</th>
<th>Penalty</th>
</tr>
</thead>
<tbody>
<tr>
<td>Area (µm X µm)</td>
<td>1246.765 X 615.91</td>
<td>1284.1 X 771.54</td>
<td>29%</td>
</tr>
<tr>
<td>$P_{\text{dyn}}$ @80 MHz (mW)</td>
<td>330</td>
<td>373</td>
<td>13%</td>
</tr>
<tr>
<td>$P_{\text{package}}$ (µW)</td>
<td>16.5</td>
<td>25.8</td>
<td>58.3%</td>
</tr>
<tr>
<td>$f_{\text{max}}$ (MHz)</td>
<td>91.7</td>
<td>92.2</td>
<td>-0.5%</td>
</tr>
</tbody>
</table>

most signals. RFWrite, in Fig. 12, is the register-file-write control channel. RFWrite_valid must be active if data is going to be written in the register file. Therefore, RFWrite_valid has been ANDed with RegWrite inside the register file.

VI. VERIFICATION, FABRICATION AND PERFORMANCE EVALUATION

Both the clocked and elastic (without bubbles) MiniMIPS have been synthesized, placed, routed and fabricated in a 0.5 µm technology. The functionality of the fabricated processors have been verified on Verigy’s V93000 SOC tester using the testbench in [11]. The EFork and LJoin circuits (of Fig. 5 and 6a) have been used in the elastic MiniMIPS. Table I summarizes chip measurements. It shows that elasticizing the MiniMIPS has area and dynamic and leakage power penalties of 29%, 13% and 58.3%, respectively. For accurate leakage power comparison, both designs have been set to the same state (through a test vector) before measuring the average supply current.

On the other hand, there is no performance loss due to elasticization. Both MiniMIPS have been fabricated without the memory block. Memory values have been programmed inside the tester. Hence, we had to make an assumption about the memory access time, and the assumptions affect the maximum operating frequency of both MiniMIPS in the same way. Therefore, the actual value of memory access time would minimally affect the performance comparison. Hence, we chose an arbitrary value of zero for memory access time for both designs. Shmoo plots for both clocked and elastic MiniMIPS are shown in Fig. 13.

Part of the reason for the noticeable area and power overheads is that the MiniMIPS is, relatively, a small design (8-bit datapath). However, part of it too is the usage of eager forks. EFork has one flip-flop per each branch that consumes power every cycle. Add to this, its gate complexity. This motivated the research toward less complex implementations of eager forks or switching to lazy protocols.

Note that these measurements do not take advantage of bubbles problems that occur if one needs to have flexible interface latencies or extra pipeline stages inserted.

To measure the runtime advantage of eager forks over lazy, we inserted a different number of bubbles at the register file outputs (i.e., before registers A and B, simultaneously). Table II compares the number of cycles required to run simulations of lazy and eager protocol implementations using the testbench in [11]. For the lazy protocol, we used LKFork-LJoin combination. The behavioral simulations used some timing constraints enforced to avoid possible oscillations, see Sec. IV-D4. Table II shows that, in this case, there is an advantage for using eager forks, specially with a large number of bubbles in the system. The table also shows that there is no runtime penalties due to elasticization in the absence of bubbles.

The runtime advantage of the eager versus lazy designs (see Sec. IV-B) is illustrated in the following example. Fig. 14 shows a simplified part of the MiniMIPS control network. We added one bubble before the A register, and another one before the B register,
labeled b1 and b2 respectively. Consider the clock cycle when V A and V B go low. SC1 will go high through join JABC14P. In FC (assuming SC2 is low), VC is high, SC1 is high. A lazy FC will invalidate the data at C2 (i.e., deasserts VC2) until SC1 goes low again. Hence, no new data token can be written at register b1 or b2 until the stall condition on C1 is removed (i.e., SC1 goes low again). On the other hand, an eager FC will validate the data on C2 (i.e., asserts VC2) for the first clock cycle giving C2 branch an early start. Hence, new data tokens can be written immediately in registers b1 and b2 in the following cycle.

VII. CONCLUSION

Synchronous elasticization is an approach of converting ordinary clocked designs into latency insensitive implementations. Being latency insensitive, the synchronous elastic designs are able to tolerate interconnect, interface, and internal latency variations without affecting the system correctness. This allows for increased modularity and facilitates IP reuse. We presented some practical considerations of elasticizing reconvergent fanouts. We also investigated the suitability of different published, as well as new, join and fork implementations for usage in the elastic control network. We also demonstrated that elasticization can be expensive. Measurements of two MiniMIPS processor chips in a 0.5 μm node showed that elasticization results in area and dynamic and leakage power penalties of 29%, 13% and 58.3%, respectively, without any performance loss. These measurements, nonetheless, do not take advantage of bubbles that occur if one needs to have unpredictable interface latencies or extra pipeline stages inserted. We finally demonstrated the possible architectural performance advantage of eager over lazy protocols in the presence of bubbles.

ACKNOWLEDGMENTS

This work was supported by the National Science Foundation under grant CCF-0810408.

REFERENCES