Improving Synchronous Elastic Circuits: Token Cages and Half-Buffer Retiming

2010 IEEE Symposium on Asynchronous Circuits and Systems

Improving Synchronous Elastic Circuits: Token Cages and Half-Buffer Retiming

Mario R. Casu
Dipartimento di Elettronica, Politecnico di Torino, Italy

***Abstract*—Synchronous elastic circuits help synchronous designs tolerate computation or communication latencies, in a way similar to the asynchronous design style. The datapath is made elastic by turning registers into elastic buffers and adding a control layer that uses handshake signals and join/fork controllers. Join elements are the objective of two improvements discussed in this paper. The first one is an elegant implementation of input bypassable queues obtained by retiming one of the latches of the elastic buffer which follows the join controller. The second one enlarges the set of cases in which unneeded input tokens are discarded in join controllers with early evaluation. Their impact on throughput are discussed by means of examples representative of typical topologies and of a realistic processor datapath. Their area and power costs are evaluated on a 45 nm CMOS technology.**

I. Introduction

The designers of synchronous circuits in nanometer technologies are facing nowadays unprecedented challenges in closing their designs. They are more and more forced to take timing margins because of the high variability which affects process technology parameters – like threshold voltages and effective transistor channel lengths – and environmental parameters like temperature and on-chip supply voltage. They are then confronted with the dilemma of what to sacrifice, performance if they pursue a *worst case* approach, or design yield if instead they opt for a *typical* or even a *best case* target. Variability takes also other forms, like the rather unpredictable wiring delay, something that usually harasses designers in late design stages.

A recent wave of studies advocates the design of synchronous circuits optimized for the typical case and which tolerate a certain amount of variability through an error-correction approach [1][2][3] or variable-latency mechanisms [4][5][6]. These methods bring the computer architecture mantra "make the common case fast" – which ensues from Amdahl's law [7] – into the circuit and logic domain.

An answer to the variability problem exists since long time but many consider it too radical and consists in replacing the global clock with an asynchronous self-timed operation. Asynchronous circuits may be designed to be "elastic" against environmental changes. A discrete degree of elasticity, limited in resolution by the clock granularity, can be brought also in the synchronous domain. Synchronous elastic circuits have been proposed in the last decade in various shapes [8][9][10][11][12]. From the implementation viewpoint, they all rely on a handshake protocol based on *valid* and *stop* signals which travel in the same and in the opposite direction of data, respectively. Information about late completion of the datapath operations or late arrival of signals from long wires can be used by the elastic protocol so as to stretch computation in time (in a discrete way) and make it insensitive to excessive latency caused by variability.

If the *latency-insensitive design* approach described firstly in [8] seems more appropriate for system-on-chip design, the one proposed in [9] and then resumed and ameliorated in [10][12] is particularly interesting for circuit and micro-architectural design [13]. Our work stays in this last vein and backs two improvements aimed at a performance increase:

1) An elegant implementation of input bypassable queues which increase the throughput of systems with multi-input elastic controllers and which are obtained by retiming one of the latches of the elastic buffers that follow such controllers.
2) A circuit that we call "token cage" to be added to the multi-input controller of [12] and which improves its performance by discarding more of the data which are valid but useless for a given computation.

We start with a short review of synchronous elastic circuits in section II, we motivate this work in section III and illustrate the two main findings in section IV and V. A simple yet close to real-life example of application is presented in section VI. Section VII describes the results of logic synthesis and mapping experiments on a 45 nm CMOS technology. The conclusions are drawn in section VIII.

II. Elastic Circuits: A Short Review

We provide some basic grounds on elastic circuits and refer the reader to a recent published paper and to the references therein for any further details [14].

In a *synchronous elastic circuit* (SEC) computations are scheduled by a global clock. We will not discuss asynchronous elastic circuits and so will refer to SEC simply as Elastic Circuits (EC), without ambiguity. The constituent blocks of an EC tolerate (discrete) latencies of their input data and execute a computation when all data (or the minimum needed subset of them) are present. The reason why some of the input data to a block may arrive later than others is out of the paper's scope. In general, computation and communication latencies may arise which could not be predicted at design time, or scarcely so. Elasticity can be then also termed and formalized as *latency-insensitivity* [15].

Marked Graphs (MG), a subclass of Petri Nets [16], and some derivations like Dual Marked Graphs [12], are an

978-0-7695-4032-0/10 $26.00 © 2010 IEEE
DOI 10.1109/ASYNC.2010.16

appropriate model to evaluate behavior and also to calculate performance of Elastic Circuits. Vertexes and edges of an MG represent events and relations of causality between them. Edges can be unmarked or marked with *tokens* which represent the system's state. The initial marking represent system's initialization from which evolution begins. When *all* input edges of a vertex are marked an event *can be fired*, all input tokens are removed and a token is placed in each output edge. In our case events will be timed by the global clock ticks and correspond to synchronous elementary computations. Performance's measure is *throughput*, that is the average number of events per unit time, in other words the computations per clock cycle made by each block.

EC's require a handshake protocol to implement latency tolerance. Every data link connecting two blocks is complemented with a pair of control signals, *valid* and *stop*. The first one flows in the same direction of data and means *valid data*. In MG's vocabulary, it's a token. The second one flows in the opposite direction and is used to stop valid data that cannot be immediately consumed. Valid and stopped data are stored in *Elastic Buffers* (EB's). Their token capacity must be greater than or equal to the maximum forward-propagation latency of valid data plus the maximum back-propagation latency of stop signals, $C \geq L_f + L_b$. Usually, EB's are designed to hold exactly two items as a consequence of a unitary latency in both directions. A smart implementation consists of a pair of level-sensitive latches which may work as an edge-triggered register when there's no data to stop, or as two separately controlled memory elements when there's an output stop to absorb [9][10]. In the latter event one of the latches holds the stopped datum whereas the other one prevents the incoming one from being lost. An *elastic half-buffer (EHB) controller* defines latch transparency and memory conditions. An EHB controller and a latch form an EHB and two EHB's form an EB.

EB's initialized with void data – we will refer to them as *bubbles* – can be added without changing the circuit functionality. This is an effect of latency insensitivity and is crucial to improve throughput for particular topologies.

Blocks that have multiple inputs and/or multiple outputs need a *Join* and/or a *Fork* controller. The join element implements the *AND* firing rule described above, that is asserts a valid output only when *all inputs are simultaneously valid*, otherwise the valid ones get stopped. Fork elements send valid tokens along the various output directions, if the receivers are ready to get them. If instead one or more stop the data, the fork controller keeps data valid for them and invalidates the channels that were not stopped.

The AND firing rule is too restrictive for many practical cases in which computations take place with a subset of valid inputs, as it happens for a *2-way multiplexer* which reads one input at a time. The join controller can be then modified to handle *early evaluation* and so to fire as soon as the subset of needed inputs are valid, regardless the status of the unselected ones. This implies discarding the unneeded tokens which sooner or later arrive at the join inputs. A possible technique consists in sending back *negative* tokens, a.k.a. *antitokens*, which travel in the opposite direction of standard tokens. When a token meets an antitoken, both are canceled [12]. Or, they can be *accumulated* locally, waiting for the positive tokens to arrive [11][17]. These two techniques are thus nicely termed *active* and *passive* antitokens. A full-blown implementation of active antitokens requires doubling the protocol signals with negative valid and stop wires. The question whether a passive or active implementation performs better is still open, although the work in [18] indicates a slight preference for the passive one.

Figure 1 shows an example of elastic block partitioned in datapath and control. The cloud is a combinational logic the output of which is registered by an EB made of two latches each managed by its own EHB controller. L and H indicate active-low and active-high transparent latches. The join controller implements a strict AND firing rule [10].

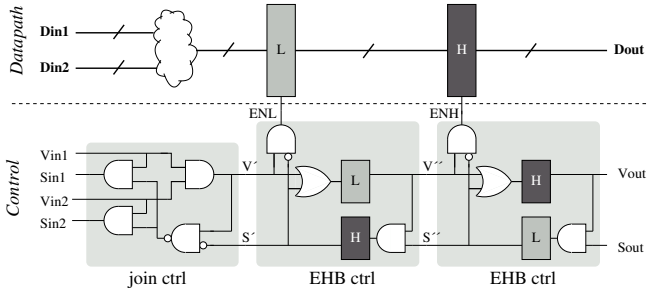

Figure 1. Elastic datapath and control: 2-input join and elastic half-buffer controllers [10].

In the example in Figure 2 the join controller implements early evaluation and supports antitoken generation and propagation [12]. The datapath, not reported, is identical to the one in Figure 1.

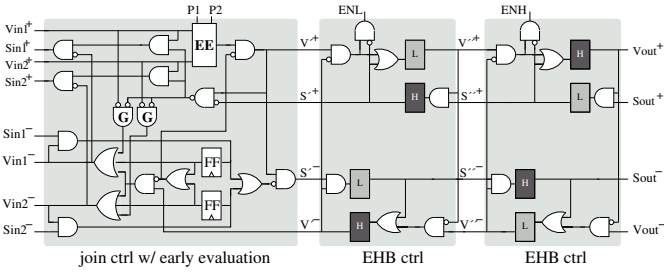

Figure 2. Elastic control: 2-input join with early evaluation and elastic half-buffer controllers with support for antitokens [12].

P_1 and P_2 in Figure 2 indicate which inputs are "processed". Accordingly, the early-evaluation logic represented by the block labeled EE will evaluate the following condition

$$EE = (\overline{P_1} \vee V_{in1}^{+}) \wedge (\overline{P_2} \vee V_{in2}^{+}). \quad (1)$$

EE is asserted even if a channel is not valid, provided it is not processed. In that case, an antitoken (V^-_{in1} or V^-_{in2}) can be generated by *G* labeled AND gates. The flip-flops are set when valid antitokens are stopped ($V^-_{in1} \wedge S^-_{in1}$, $V^-_{in2} \wedge S^-_{in2}$) and need to be kept for the next clock cycle. The EHB controllers are more complex than those in Figure 1 as they propagate antitokens and elaborate a more sophisticated enabling condition for the latches.

We did not report examples of fork structures as we will not touch them from now on. Our optimizations apply to structures which use both types of join controllers.

III. Motivations of this Work

This work aims to improve the performance of elastic systems by modifying the join element in two ways. The first modification applies to both the case of AND firing rule and of early evaluation. The second one only holds for the case of join with early evaluation. We briefly motivate the need for these improvements in the following two subsections.

A. The "bubble bounce" problem

A, B and C on top of Figure 3 are elastic computational blocks modeled as nodes of a simplified Marked Graph. Each block has two storage places for tokens – the two latches of an EB – but they have not been explicitly shown for simplicity. The EB of A is followed by a fork controller. The one of C is preceded by a join controller. The initial marking is given by the token configuration – that is the black circles – at T=0. The blocks that are enabled to fire are represented as gray rectangles. Block C receives a token (valid datum from A) and a bubble (not valid from B) at time T=0, thus it's not enabled. Immediately, that is combinationally, the bubble *bounces* against the join wall and turns into a stop for the valid datum. The immediate stop stalls block A which is not enabled to fire at T=0 and saves the incoming token in the L latch of its EB (token shown within the block at T=1).

The first consequence of this behavior emerges if blocks A and B are physically distant from the join block. The time it takes to propagate the bubble along the wire, the join logic delay and finally the time to back-propagate the stop along the valid channel must be less than a clock period and can be itself a clock period limiter if the wires are long. This was noticed first by C.-H. Li and others in [19].

The second consequence is a throughput reduction for topologies like reconverging branches with unequal latencies, like on top of Figure 3 where the outputs of A reconverge on C with different latencies. The stop repeats periodically and every block is enabled to fire once every two clock cycles. The throughput of the system in figure is then 1/2 and is said to be bubble limited [14]. As we previously said, adding bubbles does not change system functionality. So a way to improve the throughput could be

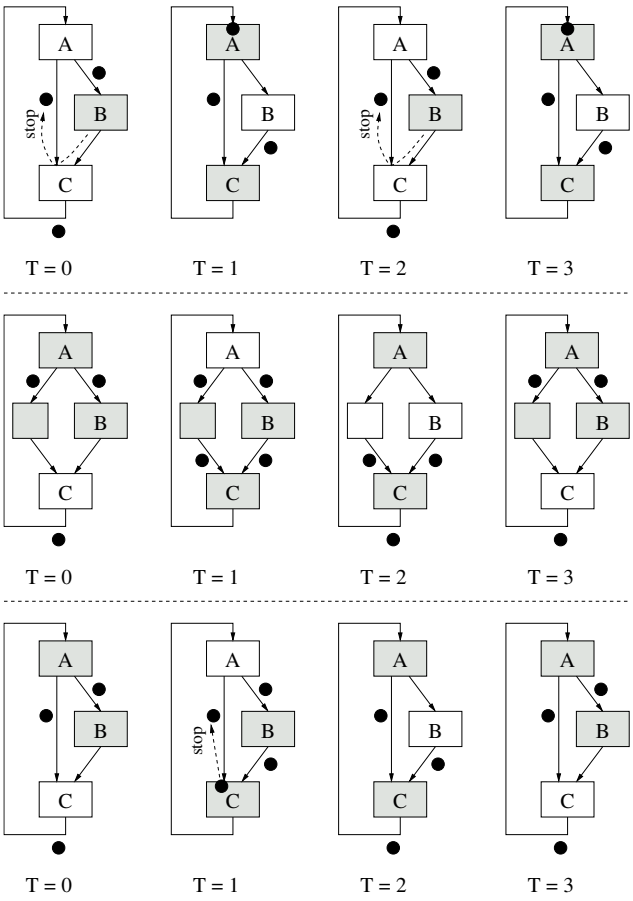

Figure 3. Top of figure: the bubble bounce problem. Middle: buffer insertion. Bottom: buffer sizing.

that of adding a bubble in the "fast" channel so as to balance the channel latencies. This is illustrated in the middle of Figure 3. The time evolution shows that there is no need to stop any data because all channels have the same latency. The throughput is 2/3 because all blocks fire twice every three cycles (marking at T=3 is the same of T=0).

In more complex systems than our simple example, adding bubbles can solve a local problem but may degrade the overall system throughput by creating new critical cycles [20]. Another option without negative consequences is *buffer sizing* which consists in increasing the capacity of buffers in short branches, possibly without changing the forward latency. In the problem at stake, we need an input buffer added to the fast channel with zero forward-latency and one-cycle backward latency. In practice, a bypassable FIFO queue which stores the data in case of stop created by the bubble bounce and which *delays* the stop itself. The combinational path of the stop signal will be split in two clock cycles, making the timing constraint less burdensome. The bottom part of Figure 3 shows what happens when such a queue was added on the left input of C. The early valid from A gets stored (token within block C at T=1).

When the late token arrives from B the queue is read and the new datum from A gets stopped. However this stop has no effect as A was already stalled because its input was not valid. The throughput increases again from 1/2 to 2/3.

We show later on in section IV an elegant solution for this queue which basically consists in retiming the elastic half-buffer which follows the join element.

B. The "useless (yet stopped) token" problem

Suppose now that the join controller of block C in the top-most part of Figure 4 implements an early evaluation firing rule. Letter "P" indicates the right input as the needed one, that is the "processed" input, for the present computation. Unfortunately, the processed channel is devoid of tokens whereas the left input contains a "useless" token. The join controller of C cannot cancel that token with an antitoken because this would require an output positive token to be generated at the same time (token preservation [12]).

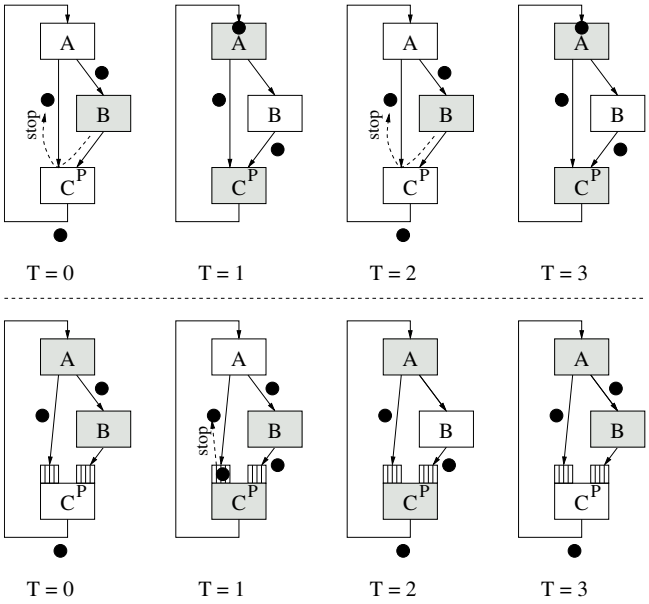

Figure 4. Top: the useless token problem. Bottom: effect of token cage.

To figure out the situation it is expedient looking at the implementation of the controller logic in Figure 2 and at logic equation (1). Suppose that input 1 is processed and not valid while input 2 is not processed and valid:

$$(P_1 \wedge \overline{V_{in1}^{+}}) \wedge (\overline{P_2} \wedge V_{in2}^{+}).$$

The consequent false value of EE in (1) and so of V'^{+} in Figure 2 sets to false the two negative valid signals V_{in1}^{-} and V_{in2}^{-} (we assume a false value of V_{out}^{-} and of the two flip-flops). In turn the false value of V_{in2}^{-} asserts S_{in2}^{+} which stops the valid input. This is graphically represented by the stop arrow at T=0 in Figure 4, top part.

The back-pressure exerted on the valid and useless channel results in the end in a smaller throughput, if that channel belongs to a critical part of the system. The time evolution on top of Figure 4 shows that a throughput of 1/2 is obtained, exactly like it happens in the case without early evaluation depicted in the top part of Figure 3.

A reasonable solution consists in discarding the useless datum and "remembering" that it was canceled out. If, after such elimination, another token pops up in that channel before the needed token on the processed channel arrives, the new one will be stopped, as it is impossible to know if it will be needed in the future (the processing configuration may switch). We need thus a sort of "queue" of unit capacity, just for the valid signal, not for the data, but since we will not use the queue content anymore we call it "token death cage", or "cage" for short. The bottom part of Figure 4 shows the time evolution when the useless token is caged – and not stopped anymore – at time T=1. The resulting throughput is 2/3, the maximum possible for this case, obtained without any buffer sizing. As shown in figure, we need to stop at T=1 the new datum on the non-processed input: We don't know if it will be processed next. However, there's no throughput penalty here because the stop occurs when A is not enabled.

We show later on in section V a possible implementation of the cage.

IV. Half-Buffer Retiming

Figure 5 represents a typical situation in which two channels from two EB's on stage *i* of a hypothetical pipeline join on a single EB on stage *i+1*. The figure shows both data and control paths. The clouds represent combinational logic and/or wires annotated with delays, T_{d1}, T_{d2} and T_{dj}.

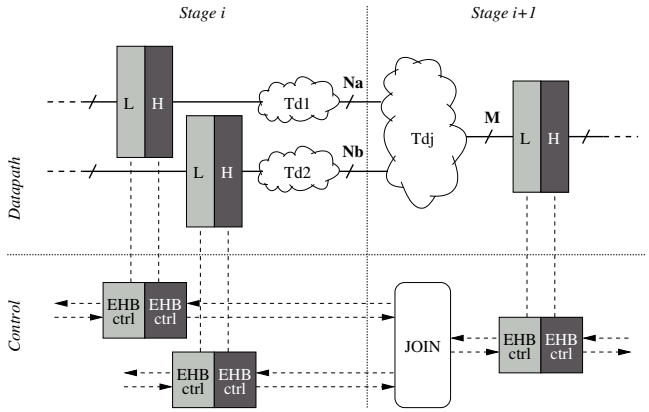

Figure 5. Datapath: Two data inputs merge on the same output data. Control: The join controller merges/forks valid/stop signals.

We create two bypassable queues on the inputs of stage *i+1* in two steps whose final result is shown in Figure 6:
1. Retime the negative level-sensitive latch (L) of the output EB moving it backward across the cloud.
2. Retime and duplicate the EHB controller and let the two

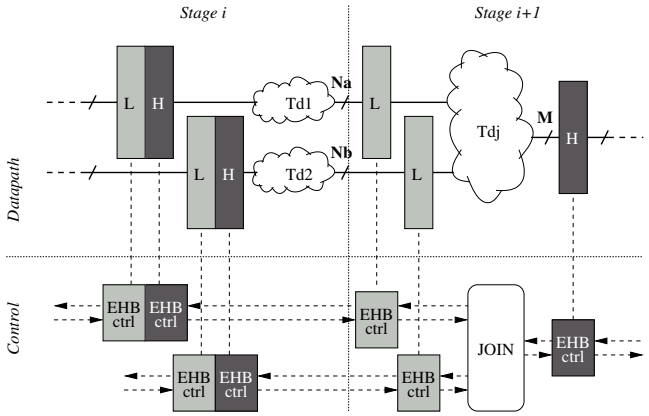

Figure 6. Datapath: Two negative-level sensitive latches have been retimed. Control: Two elastic half-buffers controlled have been retimed.

copies operate independently on channels A and B.

Moving back and doubling the latch does not necessarily mean doubling the area occupied. It may be the case that the input and output size of the combinational logic are such that $N_a + N_b \leq M$ (take for example a 16x16 multiplier with 32 bit output).

EHB retiming applies to join controllers without and with early evaluation. Since the latter contain the edge-triggered flip-flops shown in Figure 2, after having retimed the active-low latch we must change the trigger polarity of the flip-flops from positive to negative.

Two issues arise, though, when we retime latches and EHB controllers. The first one regards timing constraints. The second one concerns the way antitokens produced by an early evaluating controller move across the latches. The following two subsections face these two matters.

A. Timing issues with latch retiming

Moving the two latches ahead may require a time borrow to complete the computation of the "cloud" in Figure 6 which takes T_{dj} time units. Suppose we use a single-phase clock scheme with symmetric duration of low and high phases, 1/2 clock cycle each. Suppose that prior to latch retiming there was no slack left in the path which crosses the two datapath stages, i-th and i+1-th (critical path). Then the clock-output delay t_{cq} of the upstream EB plus the propagation delay of the clouds and the setup time t_{su} of the downstream EB sum up to a clock period T_{ck}. In formulas,

$$t_{cq} + max(T_{d1}, T_{d2}) + T_{dj} + t_{su} = T_{ck}. \qquad (2)$$

This condition is graphically exemplified in Figure 7, top waveforms labeled "before retiming". The input of the active-low latch of stage i+1 (Lin) arrives just t_{su} before clock's edge (null slack) and gets stored. The output of the active-high latch (Hout) arrives t_{cq} after the second positive clock edge.

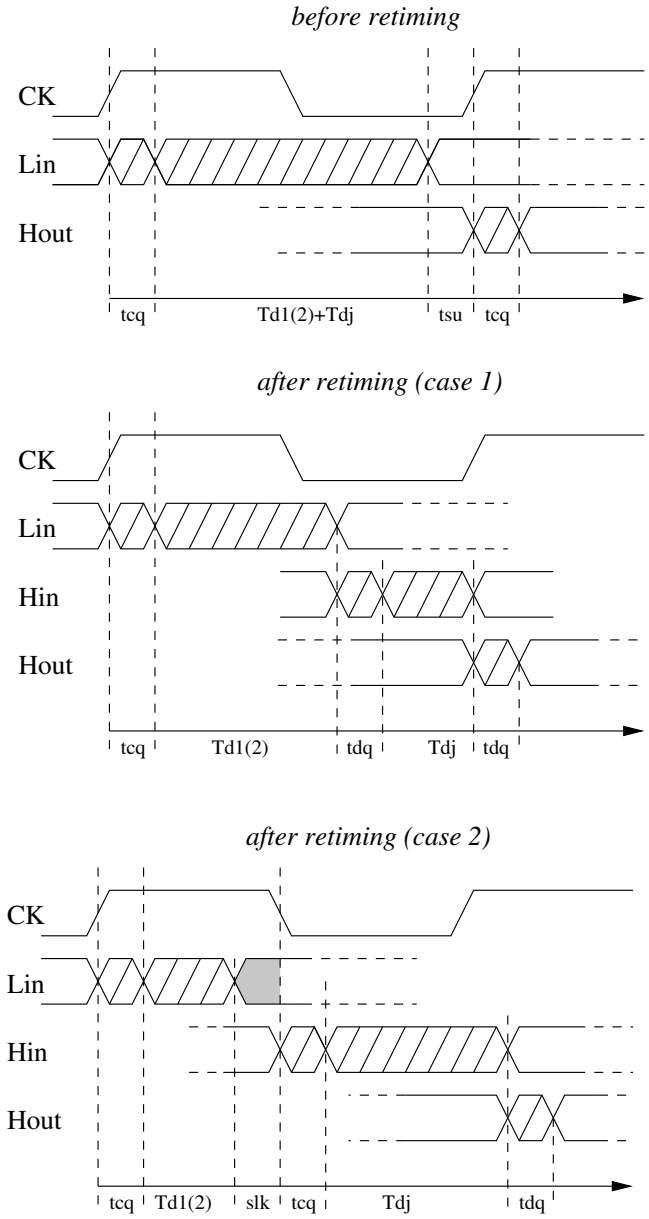

Figure 7. Timing waveforms before and after retiming.

After retiming, two possible situations occur depending on the arrival time of Lin:

1) $t_{ck-q} + max(T_{d1}, T_{d2}) > T_{ck}/2$.
2) $t_{ck-q} + max(T_{d1}, T_{d2}) \leq T_{ck}/2$.

The combination of these two inequalities with (2) results in a duration of T_{dj} shorter or longer than $T_{ck}/2 - t_{su}$. In the first case, corresponding to the middle part of Figure 7, Hout arrives $2t_{dq} - t_{su}$ after clock's edge. Such quantity is close to t_{cq} if both input-output delay t_{dq} and setup time t_{su} are on the same order of the clock-output delay, as it usually occurs. In this fortunate case, the arrival time of Hout does not change appreciably and there's no extra time to borrow

with respect to the case we had before retiming.

In the second case, exemplified in the bottom part of Figure 7, there is a slack slk between the arrival time of Lin and the opening of the active-low latch (which occurs after 1/2 clock cycle). Therefore, the arrival time of Hout exceeds the clock period of $slk + t_{dq} + t_{cq} - t_{su}$. If again $t_{dq} \simeq t_{su}$, slk is more or less the quantity in excess of t_{cq} that needs to be borrowed, if available, from the next stage.

In case of early evaluation controllers, L latches must be retimed also on P1 and P2 inputs of Figure 2 and a similar analysis must be done for timing paths that pass through them. Usually the simpler protocol logic has more slack than datapath logic making it easier to fix timing constraints.

It's worth noting that retiming helps tolerate delay variability as it relaxes the constraint of (2): The computation is allowed to take more time than nominal T_{dj}, provided that a slack can be borrowed from stage i+2.

B. The "late antitoken bounce" problem

Assume we want to apply retiming to a join controller with early evaluation. We still want to solve the bubble bounce problem by delaying the backpressure signal that stops an "early" valid while we're waiting for the other token. However, when such early valid is the needed one, we don't stop it and send instead an antitoken back in the bubble direction. Without retiming, such antitoken is immediately, that is combinationally, sent back. Again, such token-antitoken bounce could be timing critical in the same way as it was the bubble bounce. But apart from this potential problem, such immediacy may be important for throughput reasons. With retiming, the early token reaches the join controller and generates an antitoken half clock cycle later due to the latch crossing. In turn, this "bounced" antitoken reaches the upstream EB another half clock cycle later, due to the other latch. Overall, the antitoken arrives after one clock cycle, a latency which may negatively impact performance, as explained in Figure 8. In the example labeled "before retiming", C and D implement early evaluation. "P" indicates the needed input for present computation. The join controller of D immediately generates an antitoken – a white circle in figure – at T=0. Such antitoken goes back and annihilates the token on B's output at T=1. All blocks except C become and remain enabled (gray shading) after 2 clock cycles from inception. The system throughput is one operation per clock cycle. The fact that C never operates has no effect on global performance as its output is unnecessary to D, and hence unnecessary to the whole system.

The example in the middle of Figure 8 represents the same system in which retiming was applied to block D. The antitoken arrives in B at time T=2, one clock cycle later than in the previous example. The token on C's input at time T=1 is not annihilated and gets stopped as a consequence of a bubble bounce. Such stop back-propagates and makes stall first B (T=1), second A (T=2), and finally D (T=3, not

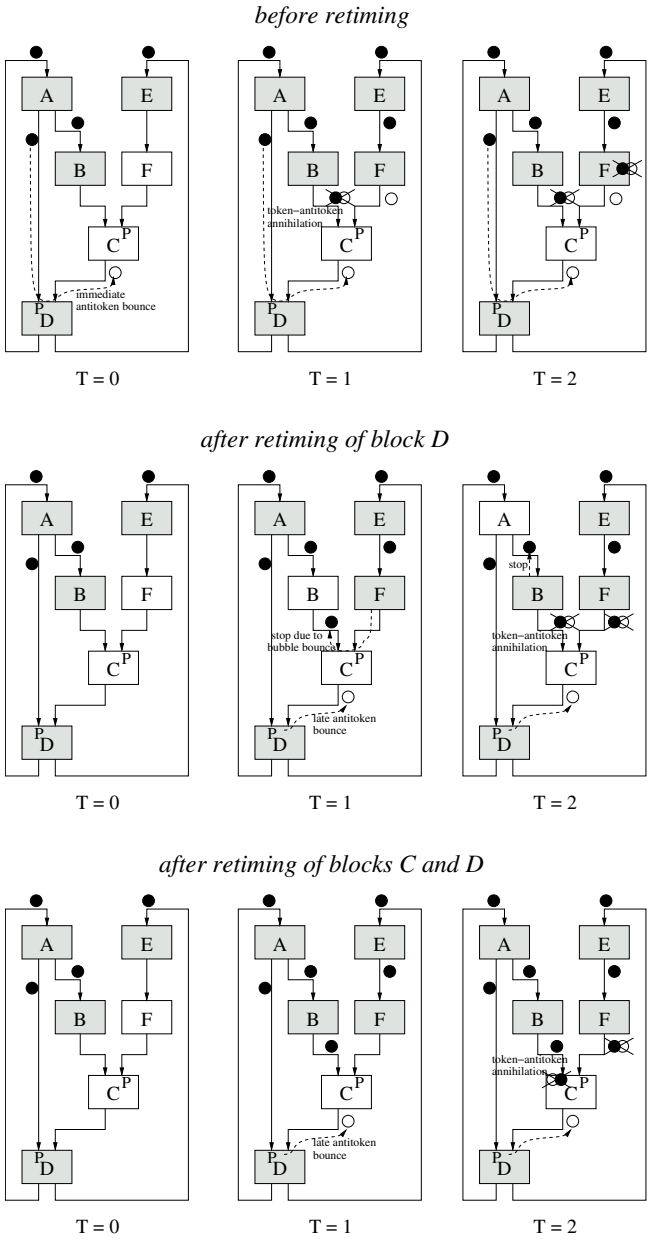

Figure 8. Example of timing evolution before and after retiming with early evaluation blocks.

shown). If the configuration of processed channels never changes, this situation repeats periodically leading to an overall throughput of 4/5, 20% lower than the original case.

If we finally apply retiming to both blocks C and D, we remove the bubble bounce problem which occurred at time T=1 by storing the token on C's input in the retimed latch. As the bottom part of Figure 8 shows, at T=2 such token gets annihilated and leaves its place free for the newly arrived token. In the end, the final throughput is 1, with no performance degradation compared to the original case.

The last example shows that the application of retiming

needs to be done judiciously whereas early evaluation blocks are used. As for the join controllers without early evaluation, the retiming technique can be always applied without any throughput penalty, after a proper timing verification.

V. Token Cages

The join controller with early evaluation of Figure 2 sets the output token if none of the flip-flops holds a previously stopped antitoken and the output of the early evaluation block (EE) is also true. The latter condition occurs if the processed channel is valid, regardless the state of the not processed one, as explained by equation (1). If instead the processed channel is not valid, EE's output is false and the controller stops the not processed input, if valid (unless there was a previously stopped antitoken stored in the flip-flops which cancels the valid token). The not processed token is stopped even though it will never be used. As we previously noted in section III-B, instead of stopping it we can put it in a *cage* waiting for the right conditions to kill it, unless another token was previously caught and not killed yet.

The circuit in Figure 9 implements the token cages of a 2-in join controller. The addition of cages concerns only the control part of a circuit, the datapath remains unmodified. If cages are free, that is the flip-flops content is zero, the conditions under which tokens get caged are

$$S_1'^{+} \wedge V_{in1}^{+} \wedge \overline{P_1} \wedge \overline{EE},$$
$$S_2'^{+} \wedge V_{in2}^{+} \wedge \overline{P_2} \wedge \overline{EE}.$$

A caged token gets killed, that is the FF content is zeroed, when the join element removes the corresponding stop signal $S_1'^{+}$ or $S_2'^{+}$. An antitoken $V_1'^{-}$ or $V_2'^{-}$ gets propagated to V_{in1}^{-} or V_{in2}^{-} only if the cage is free. The join controller operation guarantees the following invariants [12]

$$\overline{V_1'^{-} \wedge S_1'^{+}}, \quad \overline{V_1'^{+} \wedge S_1'^{-}},$$
$$\overline{V_2'^{-} \wedge S_2'^{+}}, \quad \overline{V_2'^{+} \wedge S_2'^{-}}.$$

Therefore, the join controller cannot send a negative token back without deasserting the corresponding positive stop signal. So, if the cage is occupied and an antitoken ($V_1'^{-}$ or $V_2'^{-}$) is sent, a false value of $S_1'^{+}$ or $S_2'^{+}$ will clean the cage (its token gets annihilated by the incoming antitoken).

It is important to check that the insertion of cages respects the invariants and other SEC properties like persistence and liveness [10][12]. Although it is always possible to resort to model checking for such verification, it is rather easy to fully prove them because of the simplicity of the cage. We did not report proofs for reasons of space but we can tell that by assuming the absurd hypothesis of properties violation by the cages, one concludes that it is the surrounding environment to violate them, which cannot be true.

The example in Figure 10 is similar to the one we previously analyzed in Figure 8 in which block D was retimed and C was not, and which had a throughput of 4/5. Here the difference is that cages have been added to the inputs of block C[1]. At time T=1, the bubble on C's input does not turn into a stop because the useless token gets caged. At time T=2, the caged token gets killed but the new value needs to be stopped, as we don't know if it will be necessary in the following computation (the processed input may switch) and because there's no place to store it as we could do with retimed latches. The stop signal then back-propagates and progressively stalls all blocks along the (B,A,D,C) loop. The resulting throughput will be 5/6, higher than the value of 4/5 that we obtain without cages.

The capture of useless tokens helps improve throughput for a small area overhead that we quantify later on in section VII.

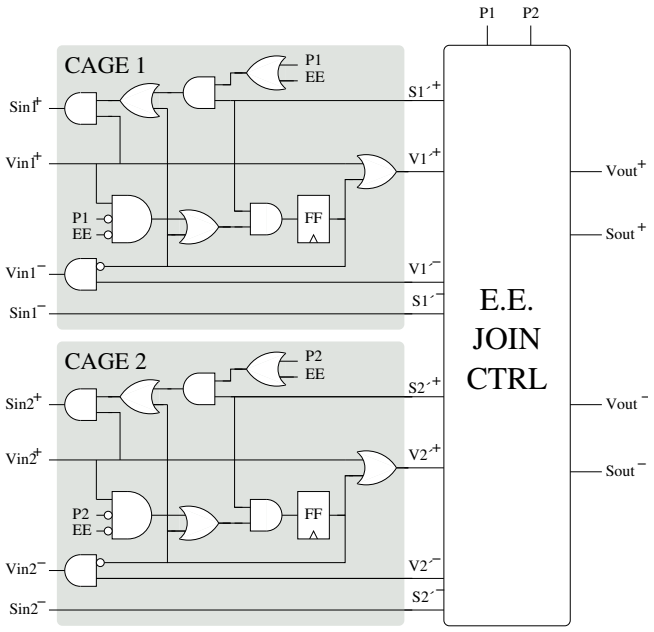

Figure 9. Insertion of two token cages on the join controller inputs.

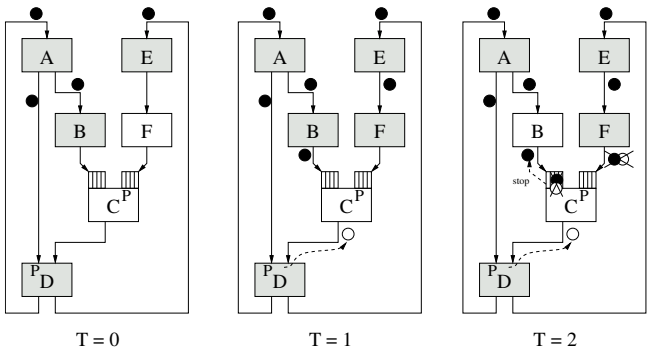

Figure 10. Example of timing evolution after cage insertion.

[1]It's not necessary to add token cages to all of the inputs, but just to the ones that need them to increase the throughput.

VI. A Realistic Example

Figure 11(a) shows a simplified elastic processor the execution unit of which supports 4 types of operations:

- addition (ADD):$X + Y$
- multiplication (MUL):$X \cdot Y$
- multiply & accumulate (MAC): $X \cdot Y + Z$
- add & accumulate (AAC): $X + Z$

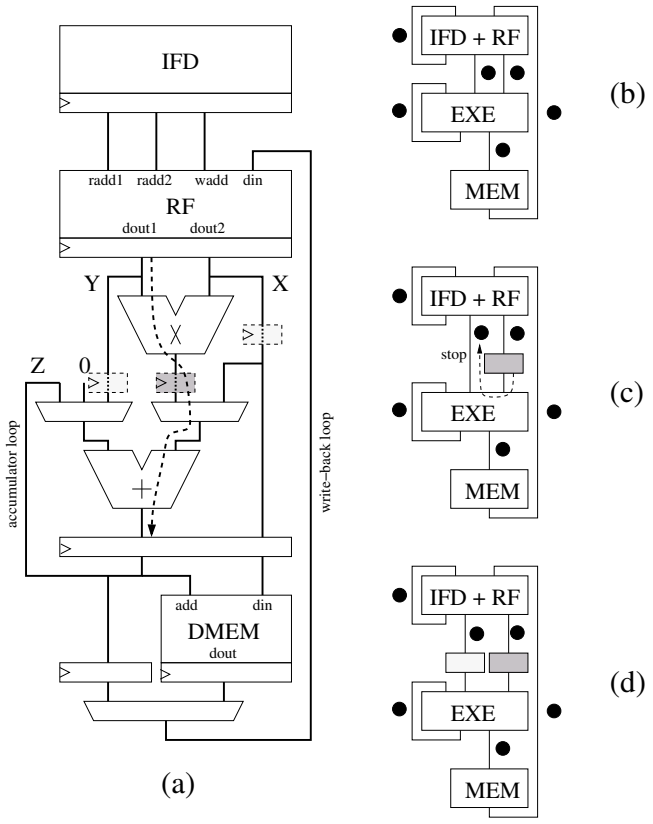

Figure 11. Example of elastic processor: (a) datapath, (b) marked graph w/o buffers, (c) w/ one buffer to break critical path, (d) w/ a second buffer to balance latencies.

The IFD block performs instruction fetch and decode, the RF block is the register file and the DMEM block is the data memory accessed via load-store instructions. The execution units includes a multiplier and an adder/accumulator. The usual branch loop found in all processors was not shown for simplicity in Figure 11(a) but was taken care of in the graphs reported on the right of the figure itself. The self-loop of the IFD+RF block on the one hand describes the branch loop, on the other hand represents a "state" and the fact that the register file always stores a valid token [13].

The graph in Figure 11(b) does not contain bubbles and the corresponding throughput is 1 (all blocks always enabled). Join elements do not implement early evaluation, for now. The critical path of the circuit, highlighted in Figure 11(a) with a dashed line, goes through the multiplier and the adder. Suppose that we want to increase the clock frequency by breaking the critical path and inserting an elastic buffer initialized without a token, i.e. a bubble, which corresponds to the dashed darker gray register in Figure 11(a). The throughput can be calculated by inspection of the simplified marked graph in Figure 11(c) in which we inserted a gray buffer. Due to the bubble bounce problem (highlighted with a stop arrow in figure), the throughput is 1/2, exactly like in the example we reported in the top part of Figure 3. The reduction of throughput wipes out any frequency increase.

If we now apply half-buffer retiming to the EXE join element to solve the bubble bounce problem, we can raise the throughput up to 2/3. If we instead insert buffers on the register file outputs (pale gray boxes in figure) we will be able to equalize the paths, as clear from the graph in Figure 11(d), making the throughput increase up to 3/4.

Buffer insertion seems the best option, but we did not take early evaluation under consideration so far. Assume that the EXE join controller implements early evaluation. Suppose we do not insert the second buffer that equalizes latencies. Then we have two different situations depicted in Figure 12(a) and 12(b), depending on the type of operation. In the left graph, the P letter indicates that the EXE unit is processing the low latency result, coming from the adder (ADD or AAC instructions) or directly from the RF. When this fast path is selected, the throughput is maximum, that is 1. The dashed loop which crosses only blocks that hold tokens and no bubbles demonstrates this. When the late path is chosen (MUL or MAC operations), Figure 12(b), a bubble bounce occurs and the fast token gets stopped, though not processed. If no retiming nor cages are used, the throughput is 1/2 as it was for the example on top of Figure 4. Latch retiming raises it up to 2/3 and finally retiming plus cages reaches a throughput of 3/4, the maximum possible since the active loop now contains 3 tokens and 1 bubble.

Now suppose a second buffer was inserted for balancing the two latencies. Figure 12(c)-(d) shows the same two processing configurations of Figure 12(a)-(b). It's clear that whatever the processing case, the active loop contains three tokens and one bubble and the throughput is 3/4, always.

In conclusion, with buffer insertion we cannot reach unitary throughput. The throughput with token cages and retiming is instead as much as the buffer insertion one in the worst case and maximum, i.e. unitary, in the best case.

VII. Synthesis and Mapping Experiments

The performance results of the proposed controllers come at an area and power cost that we quantified so as to have a clearer picture. We first described in synthesizable VHDL the following four elastic controllers which combine the basic elements described in [10] and [12] and that we used as reference designs:

- *2i/1o:* 2-input join and EB controller (like in Figure 1).

Table I
LOGIC SYNTHESIS AND TECHNOLOGY MAPPING RESULTS ON A CMOS 45 NM TECHNOLOGY.

	area			dynamic power			leakage power		
	(μm^2)	ovh. (%)	vs EB (%)	(nW/MHz)	ovh. (%)	vs EB (%)	(nW)	ovh. (%)	vs EB (%)
64 bits EB (datapath)	1099.32	–	–	452.13	–	–	38.46	–	–
2i/1o after [10][12]	32.10	–	2.9	49.76	–	11.0	6.71	–	17.4
2i/1o ret.	41.98	+30.8	3.8 (+0.9)	66.32	+33	14.7 (+3.7)	7.79	+16	20.3 (+2.9)
EE 2i/1o after [10][12]	82.91	–	7.5	109.12	–	24.1	12.85	–	33.4
EE 2i/1o ret.	101.61	+22.6	9.2 (+1.7)	131.74	+20.7	29.1 (+5.0)	17.33	+34.9	45.1 (+11.7)
EE 2i/1o cages	110.07	+32.8	10.1 (+2.6)	141.0	+29.2	31.2 (+7.1)	16.89	+31.4	43.9 (+10.5)
EE 2i/1o ret. & cages	132.65	+60.0	12.1 (+4.6)	158.9	+45.6	35.1 (+11)	21.96	+70.9	57.1 (+23.7)
2i/2o after [10][12]	44.81	–	4.1	71.24	–	15.8	6.53	–	17.0
2i/2o ret.	60.33	+34.6	5.5 (+1.40)	90.88	+27.6	20.1 (+4.3)	9.79	49.9	25.4 (+12.0)
EE 2i/2o after [10][12]	109.02	–	9.9	138.66	–	30.7	15.84	–	41.2
EE 2i/2o ret.	128.77	+18.1	11.7 (+1.8)	164.38	+18.5	36.4 (+5.7)	19.78	+24.9	51.4 (+10.2)
EE 2i/2o cages	136.18	+24.9	12.4 (+2.5)	172.14	+24.1	38.1 (+7.4)	20.49	+29.4	53.3 (+12.1)
EE 2i/2o ret. & cages	159.82	+46.6	14.5 (+4.6)	192.98	+39.2	42.7 (+12.0)	25.79	+62.8	67.1 (+25.9)

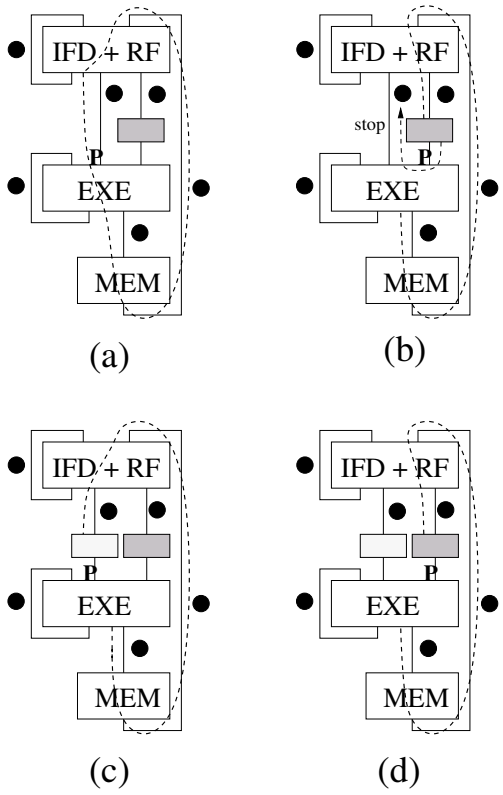

Figure 12. EXE join with early evaluation and different processing conditions: (a) no buffer and fast input processed, (b) no buffer and slow input, (c) buffer and fast input, (d) buffer and slow input.

- *EE 2i/1o:* 2i/1o with early evaluation (like in Figure 2).
- *2i/2o:* 2i/1o and 2-outputs fork controller.
- *EE 2i/2o:* EE 2i/io and 2-outputs fork controller.

Then we described the following eight new controllers:

- *2i/1o ret.:* 2i/1o with EHB retimed.
- *EE 2i/1o ret.:* EE 2i/1o with EHB retimed.
- *EE 2i/1o cages:* EE 2i/1o with cages.
- *EE 2i/1o ret. & cages:* EE 2i/1o with EHB retimed and cages.
- *2i/2o ret.:* 2i/1o ret. and 2-outputs fork.
- *EE 2i/2o ret.:* EE 2i/1o ret. and 2-outputs fork.
- *EE 2i/2o cages:* EE 2i/1o cages and 2-outputs fork.
- *EE 2i/2o ret. & cages:* EE 2i/1o ret. & cages and 2-outputs fork.

Finally, we evaluated area and power consumption (dynamic and leakage) after logic synthesis and technology mapping on a 45 nm 1.1 V CMOS technology using Synopsys Design Compiler. We set a 500 MHz clock frequency constraint, with all inputs driven by a fanout-of-one (FO1) inverter and all outputs loaded with four FO1 inverters (FO4 load), and nominal process, voltage and temperature conditions. A 50% switching probability was set on valid and stop inputs.

Table I reports all the results we obtained. They are expressed as both absolute values and overhead (in percentage) with respect to the reference designs after [10][12]. They are also expressed as percentage of the area and power consumed by the datapath portion of a 64-bits Elastic Buffer (64 bits latch pair). This comparison will help figure out the controller overhead on the datapath. We did not deliberately report the possible datapath overhead arising from the duplication of input latches on retiming cases, as a correct (and fair) evaluation depends on the bitwidth of inputs and outputs of the logic sandwiched between L and H latches. As we already noticed in section IV, it might turn out that there's no overhead or that an area saving may even occur. A case-by-case analysis is required for this type of evaluation.

If we look at the controllers figures only and ignore the controlled datapath, the results in table show that the overheads of the new elastic controllers compared to the original ones proposed in [10][12] ("ovh. (%)" columns) are quite significant (in the 20-30% range for the case with retiming or cages only and in the 50-60% range when retiming and

cages are used together). However the controllers always come along with an elastic datapath and so the comparisons with an EB are more significant and appropriate. When we compare the controllers figures with a 64 bits EB ("vs. EB (%)" columns), the extra area cost with respect to the standard controllers (value between parentheses in "vs. EB (%)" columns) is less than 5%, the dynamic power overhead less than 12% and up to 25.9% for the leakage contribution in the case of the most complex controller.

It's certainly true that comparative results depend on the EB size chosen and that a smaller choice (e.g. 16 or 32 bits) is bound to make relative figures look bigger. But it's also true that even in case of 16 or 32 bits datapaths, a single stage does not just contain one register, it may contain two registers (e.g. two operands or data and address pair) plus random and/or structured combinational logic that we could not account for because, again, a case-by-case analysis would be required. Based on the obtained results we judge that the area and power overhead is a tolerable amount for a large set of typical and practical cases of application.

VIII. Conclusion

In this paper we proposed two improvements that apply to the design of synchronous elastic circuits: half-buffer retiming and token cages. They aim to solve some performance issues that arise in the standard elastic buffer controllers. They can also be combined together for a further throughput enhancement. We discussed their benefits and warned about potential problems that may arise from the use of retimed half-buffers. We chose a realistic example to better display their use and potential. Finally, we evaluated the cost of the enhanced elastic controllers in terms of area and power. In a complex design, of which the datapath represents the largest part, our opinion is that the marginal cost of the new controllers will be a worth price paid for the performance improvement. However, a complete evaluation requires to know the characteristics of the involved datapath, something that in this preliminary analysis was left out as we focused mainly on the control part of synchronous elastic circuits. In the future, we plan to build real-size designs for this purpose. This will allow also further types of analysis like, for instance, the comparison between EHB retiming which creates a bypassable queue on all of the inputs of a datapath stage and the insertion of custom queues just on the inputs that requires them for throughput reasons.

References

[1] T. Austin *et al.*, " Opportunities and Challenges for Better Than Worst-Case Design," Proc. ASP-DAC, Jan. 2005, pp. 2–7.

[2] D. Blaauw *et al.*, "Razor II: In Situ Error Detection and Correction for PVT and SER Tolerance," Proc. ISSCC, Feb. 2008, pp. 400–622.

[3] K.A. Bowman *et al.*, "Energy-Efficient and Metastability-Immune Resilient Circuits for Dynamic Variation Tolerance," IEEE JSSC, vol. 44, no. 1, Jan. 2009, pp. 49–63.

[4] S. Ghosh *et al.*, "CRISTA: A New Paradigm for Low-Power, Variation-Tolerant, and Adaptive Circuit Synthesis Using Critical Path Isolation," IEEE TCAD, vol. 26, no. 11, Nov. 2007, pp. 1947–1956.

[5] X. Liang *et al.* "Revival: A Variation-Tolerant Architecture Using Voltage Interpolation and Variable Latency," IEEE Micro, vol. 29, no. 1, Jan.-Feb. 2009, pp. 127-138.

[6] D. Bañeres *et al.*, "Variable-Latency Design by Function Speculation," Proc. DATE, Apr. 2009, pp. 1704–1709.

[7] J.L. Hennessy and D.A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan-Kauffman, 2006.

[8] L.P. Carloni *et al.*, "A Methodology for Correct-by-Construction Latency Insensitive Design," Proc. ICCAD, Nov. 1999, pp. 309–315.

[9] H.M. Jacobson *et al.*, "Synchronous Interlocked Pipelines," Proc. Int. Symp. Adv. Res. Asynchronous Circuits Syst., Apr. 2002, pp. 3–12.

[10] J. Cortadella *et al.*, "Synthesis of Synchronous Elastic Architectures, Proc. DAC, July 2006, pp. 657–662.

[11] M.R. Casu and L. Macchiarulo, "Adaptive Latency-Insensitive Protocols," IEEE Des. Test Comput., vol. 24, no. 5, Sep./Oct. 2007, pp. 442–452.

[12] J. Cortadella and M. Kishinevsky, "Synchronous Elastic Circuits with Early Evaluation and Token Counterflow," Proc. DAC, June 2007, pp. 416–419.

[13] T. Kam *et al.*, "Correct-By-Construction Microarchitectural Pipelining", Proc. ICCAD, Nov. 2008, pp. 434-441.

[14] J. Carmona *et al.*, "Elastic Circuits," IEEE TCAD, vol. 28, no. 10, Oct. 2009, pp. 1437–1455.

[15] L.P. Carloni *et al.*, "Theory of Latency-Insensitive Design," IEEE TCAD, vol. 20, no. 9, Sep. 2009, pp. 1059–1076.

[16] T. Murata, "Petri nets: Properties, Analysis and Applications," Proc. IEEE, vol. 77, no. 4, pp. 541–580, Apr. 1989.

[17] C.-H. Li and L.P. Carloni, "Leveraging Local Intracore Information to Increase Global Performance in Block-Based Design of Systems-on-Chip," IEEE TCAD, vol. 28, no. 2, Feb. 2009, pp. 165–178.

[18] M.R. Casu and L. Macchiarulo, "Adaptive Latency Insensitive Protocols and Elastic Circuits with Early Evaluation: A Comparative Analysis," Elec. Notes in Theor. Comp. Science (ENTCS), 245 (2009), pp. 35–50.

[19] C.-H. Li *et al.*, "Design, Implementation, and Validation of a New Class of Interface Circuits for Latency-Insensitive Design," Proc. Int. Conf. Formal Methods Models Codesign, May 2007, pp. 13–22.

[20] R. Collins and L.P. Carloni, "Topology-Based Optimization of Maximal Sustainable Throughput in a Latency-Insensitive System," Proc. DAC, June 2007, pp. 410–416.