Half-Buffer Retiming and Token Cages for Synchronous Elastic Circuits

Mario R. Casu

Abstract

Synchronous elastic circuits borrow the tolerance of computation and communication latencies from the asynchronous design style. The datapath is made elastic by turning registers into elastic buffers and adding a control layer that uses synchronous handshake signals and join/fork controllers. Join elements are the objective of two improvements discussed in this paper. Half-buffer retiming allows the creation of input queues by relocating one of the latches of the elastic buffer which follows the join controller. Token cages improve the performance of join controllers that use the early evaluation firing rule. Their effect on throughput is discussed by means of examples representative of typical topologies, simulations with synthetic benchmarks and a realistic microarchitecture. Area and power costs of the control logic and the possible impact on the datapath are evaluated, based on the results of logic synthesis experiments on a 45 nm CMOS technology.

Index Terms

Elastic circuits, Latency tolerance.

I. Introduction

Elastic circuits, also termed Latency-Insensitive systems, offer designers a uniform methodology to render their products tolerant to unpredictable latencies coming from excessive wire or computational delays. Latency tolerance is obtained by enabling computational elements to “fire” only when the set of inputs needed for a given computation is available, and making them wait until this condition is satisfied [1]. To this aim, data needs to be paired with a validity signal. A counterflowing stop signal is also needed to block fast data when other late data are not yet available. Elastic circuits, in essence, implement a handshake protocol at the control level based on “valid” and “stop” signals. The interaction of the protocol with the datapath realizes a fine-grain clock-gating of memory elements which become “elastic”, as they are able to stretch the validity of data until necessary in case of stop. This kind of communication protocol is used by asynchronous designers to tolerate delays that vary within a range. The extension to the case of standard synchronous designs allows them to be rendered to elastic against “discrete latencies”, that is counted in terms of clock ticks.

Mario R. Casu is with the Dipartimento di Elettronica, Politecnico di Torino, C.so Duca degli Abruzzi, 24, I-10129, Torino, Italy e-mail: (mario.casu@polito.it.)

To handle blocks with more than one input, the protocol utilizes join controllers that produce a single valid output and propagate back as many stop signals as the input channels. Fork controllers do the opposite with multi-output blocks, that is replicate valid output and send back a single stop signal. Two types of join controllers have been devised, a simple one that validates the output when all inputs are valid [2], and another that "early evaluates" only a subset of valid inputs [3].

Synchronous elastic circuits, and in particular join controllers, are the subject of two innovations that we propose in this paper. In particular, our contributions are the following.

- In some cases we need to add or resize buffers to adjust latencies and increase throughput of elastic systems. We propose a new way to provide a multi-input elastic block with bypassable buffers, obtained by retiming one of the latches that form a master-slave pair in a standard elastic buffer.
- We show that there are cases in which fast and useless data can be discarded, but to keep the computation coherent we need something to remember that data was canceled. We thus propose a simple circuit based on a two-states finite states machine (FSM) that we add to the input of an elastic controller.

We term these two innovations "half-buffer retiming" and "token cages", respectively. The meaning will become clear when we explain how they work. The first one can be applied to elastic buffers used in conjunction with both standard and early evaluation join controllers. The second one is used only with controllers of the latter type.

We start with a brief introduction to synchronous elastic architectures in section II and motivate the need for the new contributions in section III. Sections IV-V illustrate the two main findings, whose application is better explained in section VI through a microarchitectural example, simple enough for illustrative purposes yet not too far from a realistic case. Section VII describes experiments of simulation with synthetic benchmarks and of logic synthesis and mapping on a 45 nm CMOS technology, through which we evaluated performance, power and area costs at control and datapath level. The work related to the topics discussed in this paper is presented in section VIII. Finally, conclusions are drawn in section IX.

II. Basics on Synchronous Elastic Circuits

We showed in the introduction that synchronous elastic circuits (SEC) use a pair of control signals, valid and stop, which control the flow of data by implementing a synchronous handshake. In particular their behavior obeys to a protocol called SELF [2] which consists of three basic actions: Transfer, Retry and Idle. Transfer occurs when valid data flow unstopped. Idle is when data are not valid and therefore cannot be stopped (a stop on invalid data doesn't get propagated back). Retry occurs when valid data get stopped and need to be held until the stop is removed. At the same time the stop propagates back to the source of the valid data, usually with a unitary backward latency (one clock cycle). It is clear that in a retry state, data which are valid and stopped on the current clock cycle need to be saved for the next clock cycle, and this requires a memory element. But another memory element is needed to save the incoming new data that otherwise will be lost, due to the latency of the back-propagated stop. (The stop is not immediately propagated back to avoid global signals and combinational loops.) Therefore a buffer with a capacity of two data is required. A clever implementation consists of a pair of level-sensitive latches which 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 [2][4]. This circuit is called Elastic Buffer (EB) and is made of two Elastic Half-Buffers (i.e. the two latches), each made of a datapath and a control part. The latter is the *elastic half-buffer (EHB) controller* that defines latch transparency and memory conditions.

Computational blocks are made elastic by replacing their registers with elastic buffers. (Registers can also be clustered and controlled by a single elastic controller.) In this case, elastic buffers are initialized with valid data. But EB's can also be added to increase the throughput of the system, an operation called *buffer insertion*, or to break long delays in wires and combinational logic, then implementing a pipeline. In this case EB instances are devoid of data and their validity is zero at inception. It is important to remark that inserting buffers initialized with invalid data does not change system functionality.

Elastic blocks with multiple inputs and/or multiple outputs need join and/or fork controllers. Fork elements send valid tokens along the various output directions, if the receivers are ready to get them. If instead one or more are not ready, fork controllers keep data valid only for them (Retry) and invalidate channels that were not stopped (Idle). Join elements may implement two kinds of "firing" rule. The simplest one, the *AND* firing rule, asserts a valid output only when *all inputs are simultaneously valid*, otherwise the valid ones get stopped. An *early evaluation* type join controller speeds computation up because enables firing as soon as the subset of needed inputs are valid, regardless the status of the unneeded ones. This implies discarding the unneeded tokens which sooner or later arrive at the join inputs. Two techniques are used to this aim. One consists in sending back *negative* tokens, a.k.a. *antitokens*, which travel in the opposite direction of standard tokens. When a token and an antitoken meet, they cancel one another [3]. A full-blown implementation of this method requires doubling the protocol signals with negative valid and stop wires. Another technique consists in counting negative tokens locally, and canceling them when positive tokens arrive [5][6].

Figure 1 shows an example of elastic block partitioned in datapath and control. The output of the combinational logic 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 [2] as clear from the gate in figure that asserts V' when both V_{in1} and V_{in2} are valid.

In the example in Figure 2 the join controller implements early evaluation and supports antitoken generation and propagation [3].

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

$$EE = (\overline{P_1} \vee V^{+}_{in1}) \wedge (\overline{P_2} \vee V^{+}_{in2}). \tag{1}$$

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 valid 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 because the focus of our investigation is the optimization of elastic structures that use join controllers.

III. Motivation

The reason of the two contributions discussed in this paper can be explained with a simple example of a typical problem that affects join controllers implemented as shown in Figures 1-2. Two different solutions to this problem can be devised, depending on the type of join controller. One applies to the early evaluation case only, another one can be used both in case of standard and of early evaluation controllers.

To illustrate the example and the solutions, we make use – here and in the remainder of the paper – of simplified Marked Graphs, with black tokens indicating data validity and with arrows indicating stopped data. Absence of valid data are called "bubbles". Marked Graphs, a subset of Petri Nets [7], are a powerful tool which permits analysis of behavior and calculation of throughput of SEC's, that is the average number of computations made by each block per time unit (per clock cycle in a synchronous system). The simplified version is sufficient for our illustrative purposes, and most of all for not making the treatment of this subject unnecessarily burdensome.

Suppose then that A, B and C in 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. Blocks enabled to fire are represented as gray rectangles, disabled blocks are white. The fact that block C is not enabled may represent two different situations related to its join controller:

A. It implements a strict AND firing rule and so needs both inputs to be valid. Since only one is valid, C cannot fire.
B. It's of the early evaluation type but the input from A is not processed, whereas the one from B is. Since the processed input is not valid, C cannot fire.

In both cases the bubble from B turns into a stop for the valid input from block A which stalls at T=0 (block A is white) and saves the incoming token in the L latch of its EB (token shown within the block at T=1). We call this as the "bubble bounce problem."

The first consequence emerges if A and B are physically distant from C. 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 [8].

The second consequence is a throughput reduction for topologies that contain reconverging branches with unequal latencies, like on top of Figure 3 where the outputs of A reconverge on C after different latencies. The stop repeats periodically and every block is enabled to fire once every two clock cycles. The throughput of the system in the figure is then 1/2 and is said to be bubble limited [1]. As we previously said, adding buffers does not change system functionality. So a way to improve the throughput could be that of adding a bubble in the "fast" channel so as to balance the channel latencies. The throughput increases from 1/2 to 2/3 because all blocks are enabled to fire twice every three cycles (the simple verification is left to the reader). However, in systems more complex than our simple example, adding bubbles can solve a local problem but may degrade the overall system throughput [9]. Another option without negative consequences is *buffer sizing* which consists in increasing the capacity of buffers in short branches without changing the forward latency. In the problem at stake, we need an input buffer with zero forward-latency and one-cycle backward latency in the fast channel. In practice, a bypassable FIFO queue which stores the data in case of a 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 middle 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: block A 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.

The queue can be applied to both cases of controllers, with and without early evaluation. But another solution to the bubble bounce problem, whose application is not alternative but can instead be used in conjunction with the former, may be devised only if C's join controller implements an early evaluation rule. Suppose that for the present computation C needs data from B. 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 [3]). To figure out the situation we can look 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, giving rise to the problem depicted in Figure 3.

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. 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. C's two inputs in the bottom part of Figure 3 are provided with cages. The time evolution describes a situation in which the processed input (indicated by P) is not valid whereas the other one is valid but useless. The useless token gets caged – and not stopped – 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. In the example, only one of the cages is used and if we are sure that this is always the case we can add just one cage on the input that needs it. We show later on in section V a possible implementation of the cage.

IV. Half-Buffer Retiming

In section III we illustrated with an example the need to provide the inputs of an elastic block with bypassable FIFO queues that store "fast" valid input data while late ones are still on their way. We now discuss their implementation via elastic half-buffer retiming. Figure 4 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 $i+1$. Even though we discuss the case of a 2-input join, the method can be extended to a generic n-input case. 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}.

We create bypassable queues on the inputs of stage $i+1$ in two steps whose final result is shown in Figure 5:

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 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 those flip-flops from positive to negative.

Retiming does not change the system functionality [10]. The sequence of firing actions is preserved. However, timing at gate level may change. Moreover, the antitokens produced by an early evaluating controller exit from retimed latches later than in the case without retimed latches, a delay that may reduce the throughput in some particular cases. The following two subsections face these two matters.

A. *Timing check after latch retiming*

We make the worst case hypothesis that before retiming there was no slack left in the path which crosses the two datapath stages, ith and $i+1$th, that is, the retimed path was a critical one. 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,

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

This condition is graphically exemplified in Figure 6, 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.

Retiming the two latches ahead may have two consequences, a positive and a negative one. The positive one is that the formerly "hard" clock edges become "softer" and time can be borrowed from one clock-phase to the other to help complete computation, a well-known property of latch-based design. For instance, if the next $i+2$th stage, not shown in figure, had a positive slack, the computation of stage $i+1$ may last more than T_{dj} time units and extend over stage $i+2$, without the need to increase clock period beyond the limit of Eq. (2). Or, longer delays due to worse process, temperature or voltage conditions could be tolerated, whereas in the edge-triggered case any excessive delay would end up in a setup violation.

However, and this is the negative consequence, a time borrow may be required to complete the computation of the "cloud" in Figure 5 even in the nominal case. Suppose we use a single-phase clock scheme with symmetric duration of low and high phases, 1/2 clock cycle each. The arrival time of Lin can be longer or shorter than a half clock cycle:

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, which corresponds to the middle part of Figure 6, 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 6, 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 protocol logic is simpler and has more slack than datapath logic, making it easier to fix timing constraints.

B. Delayed antitoken

Comparing Figures 4 and 5 is helpful to understand the problem at stake. Suppose that the join controller implements an early evaluation policy. Suppose that eventually a valid token pops up in one of the two channels in stage i, the channel that is processed. In the case depicted in Figure 4 an antitoken would be immediately, that is combinationally, sent back in the direction of the non valid channel. Again, such token-antitoken bounce could be timing critical in the same way as it was the bubble bounce described in section III. But apart from this potential problem, such immediacy may be important for throughput reasons. In the case in Figure 5, 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 7. In the example labeled "before retiming", both C's and D's join controllers implement early evaluation. "P" indicates the needed input. The controller of D immediately generates an antitoken – a white circle in figure at T=0 – that 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 the beginning, leading to a system throughput of one operation per clock cycle. The fact that C is always disabled has no effect on global performance, because its output is never used.

In the example in the middle of Figure 7 retiming was applied to block D. The antitoken arrives in B at time T=2, one clock cycle later than in the case without retiming. The token from B on C's input at time T=1 is not annihilated yet and gets stopped. Such stop back-propagates and makes stall first B (T=1), second A (T=2), and finally D (T=3, not 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.

In the bottom part of Figure 7, we finally applied retiming to both blocks C and D. Now at time T=1 data from B to C is not stopped anymore and gets instead stored in C's retimed latch. At T=2 such stored token gets annihilated and leaves its place free for the newly arrived token. The final throughput is again 1, the same of the original case.

This example showed that a careful analysis of potential performance degradation is needed when half-buffer retiming is applied to early evaluation controllers. The retiming technique can be instead applied with no throughput penalty to AND firing rule join controllers.

V. Token Cages

In section III we illustrated the case of a token that is not processed by an early evaluation controller and that arrives before the processed one. We observed that we can cancel the non processed data and remember just the validity bit by putting it into a cage. The circuit in Figure 8 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 [3]

$$\overline{V_1'^{-} \wedge S_1'^{+}}, \qquad \overline{V_1'^{+} \wedge S_1'^{-}},$$

$$\overline{V_2'^{-} \wedge S_2'^{+}}, \qquad \overline{V_2'^{+} \wedge S_2'^{-}}. \tag{3}$$

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 [2][3]. Although it is always possible to resort to model checking, we found rather easy to fully prove these properties because of the simplicity of the cage circuit. For a better readability of the paper we postponed these proofs to Appendix A.

When we introduced the need for a token cage we observed that it can be a powerful tool to increase the throughput of elastic circuits with early evaluation. Of course, such increase comes at the cost of an area overhead. However, this cost impacts only the control part of an elastic circuit and not its datapath and, as Figure 8 demonstrates, only few gates are needed. We quantify later on in section VII such small overhead.

VI. A Realistic Example

To illustrate the positive impact of buffer sizing obtained through half-buffer retiming and of token cages insertion, we selected a simple but close to real-life microarchitecture. Figure 9(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): $X + Z$

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 unit includes a multiplier and an adder/accumulator. The usual branch loop found in all processors was not shown for simplicity in Figure 9(a) but was taken care of in the simplified Marked Graphs reported on the right of the figure. IFD and RF blocks have been clustered into a unique element, called IFD+RF, and are then under the domain of a single elastic controller. The self-loop of 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 [11].

The simplified Marked Graph in Figure 9(b) does not contain bubbles and the corresponding throughput is 1 (all blocks always enabled). We assume that join elements do not implement early evaluation, for now. The critical path of the circuit, highlighted in Figure 9(a) with a dashed line, goes through the multiplier and the adder. As we stated before, elastic circuits allow to insert a buffer initialized with no valid token, that is with a bubble, so as to break a critical path while not altering the functionality. The dashed darker gray register in Figure 9(a) represents this buffer. The simplified Marked Graph in Figure 9(c) in which we inserted a gray buffer can be used to evaluate the new throughput. Due to the stop arrow in figure, the throughput is 1/2, exactly like in the example of the top part of Figure 3. It's apparent that the reduction of throughput wastes any frequency increase.

A bypassable queue on the fast input from IFD+RF block to EXE, obtained via half-buffer retiming, increases throughput up to 2/3. If instead we insert another buffer on that channel (pale gray boxes in figure) we will be able to equalize the latency of the paths, as clear from the graph in Figure 9(d), making the throughput increase up to 3/4.

Buffer insertion is the best option if we assume that the join controller of the EXE block does not implement any form of early evaluation. However, that choice would prevent from reaching unitary throughput, something that instead would be possible with early evaluation. Assume then that EXE join controller implements this firing policy. Without buffer insertion, we may have two different situations depicted in Figure 10(a) and 10(b), depending on the type of operation. In the left graph, P letter indicates that 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), see Figure 10(b), the fast token gets stopped, though not processed. That leads to a throughput of 1/2, as it was for the example on top of Figure 3. Now, if we first create an additional bypassable queue through latch retiming we'll manage to increase it up to 2/3. Finally, the combination of retiming and cages raises the throughput to 3/4, the maximum possible since the active loop now contains 3 tokens and 1 bubble.

Now suppose that instead of cages and retiming a second buffer was inserted to balance the two latencies. Figure 10(c)-(d) shows the same two processing configurations of Figure 10(a)-(b). It's clear that whatever the processing case, the active loop contains three tokens and one bubble and throughput can never be higher than 3/4.

In conclusion, with buffer insertion we cannot reach unitary throughput. With token cages and retiming, worst case performance is the same of the case with buffer, but best case throughput is maximum, that is unitary.

VII. Simulation and Synthesis Experiments

In this section we first present performance results obtained after logic level simulation experiments on a set of ten synthetic benchmarks. Following the approach described in [18] we first created VHDL netlists with random number of blocks and random proportion between 2-input join elements and 1-input blocks. Then the netlists have been simulated for a sufficiently long time so that throughput figures reach a steady-state. We refer the reader to the cited paper for the details of netlist generation.

In the remainder of the section we discuss the results of logic synthesis and technology mapping experiments on a 45 nm 1.1 V CMOS technology. The figures we report – area, dynamic and leakage power – will help quantify the trade-off between increase of performance and additional cost. More in details, we first compared the costs of the newly proposed controllers with the corresponding basic implementations that we discussed in section II. When half-buffer retiming is applied, there is for sure the additional cost of the more complex controller, but there may also be a datapath cost. Thus we evaluated both control and datapath cost when retiming is applied to the case of an elastic adder and an elastic multiplier.

A. *Simulation results*

Tables I and II report throughput figures for a set of ten synthetic benchmarks with an increasing number of blocks and of 2-inputs join elements. 50% of the blocks have been randomly assigned one token each at the beginning of the simulation. The remaining 50% blocks were initialized with no tokens. Results in Table I were obtained with half of join blocks that implemented an early evaluation firing rule and half that fired according to the standard AND rule. Table II refers instead to a case where 100% of join blocks were of the early evaluation type. Tables compare the standard elastic implementation – reference column in tables – with three new cases:

- Retiming only column: all join blocks have been retimed (and cages have not been used).
- Cages only: all join blocks with early evaluation use cages (and none uses retiming).
- Retiming + cages column combines the first two cases (all join blocks were "retimed" and those which use early evaluation were provided with cages).

TABLE I

THROUGHPUT OBTAINED AFTER LOGIC SIMULATIONS. 50% OF JOIN BLOCKS USE EARLY EVALUATION.

Bench	blocks	joins	reference	retiming only	cages only	retiming + cages
r1	6	1	0.524305	**0.541508 (+3.3%)**	**0.541508 (+3.3%)**	0.541508 (+3.3%)
r2	7	1	0.500100	0.500100	0.500100	0.500100
r3	8	4	0.333467	**0.500100 (+50%)**	0.333467	0.500100 (+50%)
r4	9	4	0.272755	0.272755	0.272755	0.272755
r5	10	4	0.333367	**0.333467 (+0.03%)**	0.333367	0.333467 (+0.03%)
r6	11	4	0.500000	0.536607 (7.3%)	0.500000	**0.538308 (+7.7%)**
r7	12	5	0.300160	**0.328966 (+9.6%)**	0.300160	0.328966 (+9.6%)
r8	13	6	0.285757	**0.285857 (+0.03%)**	0.285757	0.285857 (+0.03%)
r9	14	9	0.347870	0.376075 (+8.1%)	0.356071 (+2.3%)	**0.379476 (+9.1%)**
r10	15	8	0.333467	0.441888 (+33%)	0.437688 (+31%)	**0.444689 (+33%)**

TABLE II

THROUGHPUT OBTAINED AFTER LOGIC SIMULATIONS. 100% OF JOIN BLOCKS USE EARLY EVALUATION.

Bench	blocks	joins	reference	retiming only	cages only	retiming + cages
r1	6	1	0.333367	**0.541508 (+62%)**	0.541508 (+62%)	0.541508 (+62%)
r2	7	1	0.573215	0.573215	0.573215	0.573215
r3	8	4	0.452090	0.523205 (+16%)	0.521004 (+15%)	**0.560812 (+24%)**
r4	9	4	0.289258	**0.290758 (+0.05%)**	0.290058 (+0.03%)	0.290758 (+0.05%)
r5	10	4	0.439488	0.447189 (+1.8%)	0.445189 (+1.3%)	**0.448190 (+2%)**
r6	11	4	0.543609	0.576915 (+6.1%)	0.570414 (+4.9%)	**0.582717 (+7.2%)**
r7	12	5	0.405581	0.423685 (+4.4%)	0.416883 (+2.8%)	**0.427385 (+5.4%)**
r8	13	6	0.358372	0.331166 (-7.6%)	**0.360172 (+0.5%)**	0.331166 (-7.6%)
r9	14	9	0.411782	0.464693 (+12.8%)	0.480496 (+16.7%)	**0.502901 (+22%)**
r10	15	8	0.446589	0.470694 (+5.4%)	**0.547209 (+22.5%)**	0.476695 (+6.7%)

Bold figures indicate the highest throughput for a given benchmark. Results show a throughput greater than (significantly greater in some cases) or equal to the reference one except for just one case: random benchmark r8 in the second set of experiments, retiming only and retiming + cages columns. The reason for this apparently strange result is simply a case of "delayed antitoken" that we discussed in section IV-B. Although general conclusions cannot be drawn, the fact that only one out of twenty experiments was affected by this problem is symptomatic of the rareness of such event.

Interestingly, when 50% of join blocks use early evaluation (Table I) and so few cages can be added, retiming, which can be instead applied to all types of join elements, gives best results. When all join blocks fire *a la* early evaluation (Table II), the combination of retiming and cages results in a superior performance.

B. Controller costs

We first described in synthesizable VHDL the following four elastic controllers which combine the basic elements described in [2] and [3] and that we used as reference designs:

- *2i/1o:* 2-input join and EB controller (like in Figure 1).
- *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 design costs 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 III reports all the results we obtained. They are expressed as both absolute values and overhead (in percentage) with respect to the reference designs after [2][3]. If we look at the controllers figures only and ignore the controlled datapath, the overheads of the new elastic controllers compared to the original ones proposed in [2][3] ("ovh. (%)" columns) are quite significant, especially when retiming and cages are used together. However the elastic controllers are usually a small part of an elastic circuit. The controlled datapath contributes for the largest part to area and power consumption, as we show in the following.

TABLE III

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

	area		dynamic power		leakage power	
	(μm^2)	ovh. (%)	(nW/MHz)	ovh. (%)	(nW)	ovh. (%)
2i/1o after [2][3]	32.10	–	49.76	–	6.71	–
2i/1o ret.	41.98	+30.8	66.32	+33	7.79	+16
EE 2i/1o after [2][3]	82.91	–	109.12	–	12.85	–
EE 2i/1o ret.	101.61	+22.6	131.74	+20.7	17.33	+34.9
EE 2i/1o cages	110.07	+32.8	141.0	+29.2	16.89	+31.4
EE 2i/1o ret. & cages	132.65	+60.0	158.9	+45.6	21.96	+70.9
2i/2o after [2][3]	44.81	–	71.24	–	6.53	–
2i/2o ret.	60.33	+34.6	90.88	+27.6	9.79	49.9
EE 2i/2o after [2][3]	109.02	–	138.66	–	15.84	–
EE 2i/2o ret.	128.77	+18.1	164.38	+18.5	19.78	+24.9
EE 2i/2o cages	136.18	+24.9	172.14	+24.1	20.49	+29.4
EE 2i/2o ret. & cages	159.82	+46.6	192.98	+39.2	25.79	+62.8

C. Datapath costs with half-buffer retiming

Figure 11 reports four different elastic control configurations of the same datapath, a 2-in/1-out arithmetic block. Figure 11(a) is the standard case in which the output of the arithmetic block is registered by an elastic buffer and controlled by a 2-in join controller. Figure 11(b) is a latch-retimed version of it. The circuit in Figure 11(c) is functionally equivalent to the retimed version, but the input by-passable queues have been explicitly added – rather then obtained via retiming – and contain edge-triggered registers. The fourth configuration in Figure 11(d) represents a case in which we wanted to add a queue on just one input, something that we can't do with retiming where we are forced to provide both inputs with retimed latches. We described the four configurations in synthesizable VHDL and tried both the cases of a 32-bit adder and 16x16-bits multiplier. Input and loading, frequency constraint and other conditions were the same of the controller synthesis.

The comparison of area and power of the four configurations of adder and multiplier, reported as histograms in Figure 12, allowed us to draw some important conclusions.

- All data confirm that control costs – be area, dynamic or leakage power – are a small fraction, almost irrelevant, of the datapath costs, and so of the overall system. (Area is between 2% and 9%, dynamic power between 1% and 8%, leakage in 2%-12% range.) Based on these and previous results, we judge that in practical cases the additional cost of using the new more complex controllers proposed in this paper will be marginal compared to using standard elastic controllers.
- The cost of half-buffer retiming depends, as expected, upon the ratio between output and input bitwidths. For a 16x16 bits multiplier with 32 bits output, moving the negative latches ahead of combinational logic comes at almost no area cost, whereas for a 32x32 bits adder with 32 bits output, retiming means doubling the size

of the negative latches.

- As far as the comparison between configurations B and C goes, in terms of area the cost of creating bypassable queues via retiming compared to adding explicit queues is lower. The difference is particularly evident in the adder case. In terms of power the costs are similar.

VIII. Related work

Synchronous circuits capable of latency tolerance appeared in various shapes in the literature of the last decade, first termed as latency-insensitive design and then as elastic circuits. The different proposals rely all on a handshake protocol based on validity and back-pressure signals which travel in the same and in the opposite direction of data, respectively [2][3][4][5][12][13][14]. In this sense, this branch of synchronous design is indebted to asynchronous design, where handshake-based transactions are the norm. The use of early evaluation firing policies [15] and the idea of back-traveling anti-tokens [16][17] also come from that world.

Elasticity and Latency-Insensitivity are now, more or less, synonyms. However the *Latency-Insensitive Design* approach described firstly in [12][13] was proposed as a mean to cope with excessive wire delays in system-on-chip design. Whereas *Elastic Circuits*, starting from Jacobson et al.'s work [4], resumed and ameliorated in [2][3], was rather a proposal for variability tolerant circuit and micro-architectural design [11]. Although elastic and latency-insensitive systems can be reconducted to the same archetype, mostly because they can be modeled with Marked Graphs, there are subtle differences at the implementation level that lead to a slight performance difference [18].

Modeling issues, formal techniques, and methods for the computation or optimization of throughput in latency-insensitive systems are an active area of research [9][19][20][21][22][23].

It is straightforward to couple elastic circuits' handshake protocol with information about late completion of datapath operations or late arrival of signals from long wires. It is then easy to use this information to stretch computation in time (in a discrete way, limited in granularity by clock ticks) and make it insensitive to excessive latency caused, for instance, by variability of delays due to process or other sources of variations. Knowing the latency in advance by means of built-in self tests, or discovering it at run-time, and using this information to make a circuit work in spite of it, is one of the possible approaches to overcome variability that some researchers nowadays advocate, involved or not in elastic design [24][25][26]. Another way to combat the same form of variability is based instead on an error detection and correction approach [27][28][29]. Both approaches promote the "average case design" as a way to cope with the limits of nanoscale integrated circuits, and particularly the large variance of performance – die-to-die or within-die – for which a standard worst-case corner design is bound to waste the advantage of scaling.

IX. Discussion and conclusions

In this paper we proposed and discussed two contributions in the field of synchronous elastic circuits: half-buffer retiming and token cages. They can be used, alone or combined, to increase the throughput of elastic systems beyond what standard state of the art elastic controllers can do. We did not just discuss their benefits, we also warned about potential problems that may arise from the use of retimed half-buffers. A realistic microarchitecture helped us exemplify their use and potential. Finally, we evaluated the performance on a set of synthetic benchmarks and the cost of the enhanced elastic controllers in terms of area and power. The comparison with datapath costs helped us draw the conclusion that the new controller costs, albeit larger than the old ones, represent a marginal fraction of the total. As for the additional datapath cost of half-buffer retiming, it cannot be evaluated in absolute terms and a case-by-case analysis is required. However, the examples we chose showed that in some cases the overhead is negligible. In some other, the additional cost is still smaller than that of other solutions that are functionally equivalent and give the same additional performance, like standard bypassable queues used to realize what is called – in elastic jargon – buffer sizing. In complex designs, of which the datapath cost represents the largest design's share, we believe that the marginal cost of the new solutions proposed here will be a worth price paid for the performance improvement.

This paper did not tackle an important facet of elastic circuits that concerns design automation. We thus conclude with few hints regarding this point.

As for elastic half-buffer retiming, automation should entail two aspects, 1) to which join elements retiming should be applied and 2) how to check timing constraints after retiming. Concerning the first point, we should recall that retiming is just a smart way to do buffer sizing. Therefore algorithms that deal with buffer/queue sizing to increase performance can be used [20]. As far as timing verification is concerned, standard timing analysis tools are sufficient. We used the Synopsys suite for this purpose but are confident that other commercial programs would work equally fine.

A selective application of cages to join blocks with early evaluation requires more research. One can include cages every time a join block with early evaluation is used, a move that always results in equal or better performance with a small cost, as we previously discussed. But it's not an optimal choice. One affordable way to find suitable places for cages insertion is to simulate a system at logic level without cages and to instrument the simulator so as to report situations like the one discussed in section III and described in Figure 3. For systems with a limited number of early evaluation blocks, an alternative is to repeat a Markov chain based throughput analysis, like the one proposed in [22], by increasing the number of cages. Fortunately, the throughput increases or stays constant as the number of cages grows. Therefore two analyses should be first run with no cages at all and with all possible cages, to check whether the maximum throughput is significantly larger than the value without cages and so if it is worth to further seek for the optimal number and location of cages. A way to solve the general problem that efficiently applies even to large netlists has to be still investigated.

Appendix A
Proofs of token cage properties

We prove that a token cage does not change the properties of persistence, liveness and the invariants of a synchronous elastic circuit in which the cage was inserted. First of all, let's define such properties more formally.

Concerning the invariants, as stated by logic conditions (3), it can never be the case that an elastic block sends back at the same time a positive stop and an antitoken (that is $\overline{V^- \wedge S^+}$ is always true) or that asserts in the forward direction both a positive token and a negative stop (always $\overline{V^+ \wedge S^-}$).

Persistence, requires that a Retry state of the elastic protocol cannot be followed by an Idle state. In essence, when a token, be positive or negative, is stopped, it will be held until the stop is released. That means that retry conditions $V^+ \wedge S^+$ and $V^- \wedge S^-$ imply necessarily V^+ and V^-, respectively, in the next protocol transaction (controlled by a clock under the synchronous hypothesis).

Finally liveness guarantees that transfer states will always occur in the future, basically averting the risk of deadlock. This can be expressed by stating that retry conditions $V^+ \wedge S^+$ and $V^- \wedge S^-$ cannot persist forever and that sooner or later will be followed by $\overline{S^+}$ and $\overline{S^-}$.

To check these properties, we will often refer to the logic circuits of Figure 8. Since the token cage is a 2-states FSM, let us define state A when the flip-flop output is at logic zero and state B when at logic one.

Lemma 1. *A token cage respects the channel invariants as long as the join controller that follows the cage and the elastic block that precedes the cage respect them.*

Proof: We prove it in two parts, and show first that invariant $\overline{V_{in}^- \wedge S_{in}^+}$ at the input of the cage is respected. If we make the absurd hypothesis that it is not respected, both V_{in}^- and S_{in}^+ must be true. The first condition requires that the join controller is sending back an antitoken V'^- and that the FSM state must be A with flip-flop at logic zero (due to the AND gate the output of which is V_{in}^-). The second condition requires, if state is A due to the first condition, that the join controller that follows the cage wants to raise a positive stop S'^+. But it can't both raise a positive stop and asserts a negative token, because the controller guarantees the invariant, therefore the absurd hypothesis is false.

We prove now that invariant $\overline{V'^+ \wedge S'^-}$ is respected at the output of the cage. Assume the absurd hypothesis that both V'^+ and S'^- are true. The first condition implies that the join controller, which receives this positive token cannot raise the negative stop S'^-, because it's supposed to respect the invariant. But then S'^- cannot be true. ■

Lemma 2. *A token cage guarantees persistence as long as the join controller that follows the cage and the elastic block that precedes the cage guarantee persistence and respect channel invariants.*

Proof: We start with the persistence on the positive channel. Suppose that $V'^+ \wedge S'^+$ is true in Figure 8. S'^+ must be then forced true by the join controller. As for V'^+, it is true if the elastic block that precedes the gate is true OR if state is B (flip-flop at logic one). In this second case, persistence is guaranteed because next state will be again B, and V'^+ will be asserted again. In case state is A, then to have a valid positive output we need a valid positive input V_{in}^+ in Figure 8. Now, if the stop is propagated back, then persistence is guaranteed by the preceding block. If it's not propagated back, that means that $\overline{P_1} \wedge \overline{EE}$ is true. But then next state is B, and so persistence of V'^+ is guaranteed.

As for the persistence in the negative channel, we start with condition $V_{in}^- \wedge S_{in}^-$ true. To have V_{in}^- true, the join controller must assert V'^- and the cage must be in state A. Then, next state will still be A because a state change would require V^+_{in} to be true, an impossible condition because S^-_{in} is also true: The two true conditions would imply that the preceding block violates the invariant, something that contradicts our hypotheses. Therefore, next V^-_{in} will be true and persistence is guaranteed. ■

Lemma 3. *A token cage inserted in a synchronous elastic circuit cannot create a deadlock unless the surrounding environment does it.*

Proof: We can prove this by ensuring that condition $V \wedge S$, both for positive and negative channels, cannot last forever. The negative part is quickly verified because a negative stop S^-_{in} that stops antitoken V^-_{in} will be released by the upstream block, sooner or later, because the surrounding environment behaves correctly by hypothesis.

As for the positive channel, a received token can be stopped in both A and B states. In state A, it's the join controller that takes control of the stop when $P \vee EE$ is true (and so next state will still be A). But since the controller complies with the protocol, the stop will be surely released in the future. If state is B, the only way to keep stopping the token in the following cycles is that state B continues to last. But this requires the join controller after the cage to continuously stop the token, which contradicts the hypothesis that the outside world is free from deadlock. ■

References

[1] Carmona J., Cortadella J., Kishinevsky M., Taubin A., "Elastic Circuits," IEEE TCAD, vol. 28, no. 10, Oct. 2009, pp. 1437–1455.

[2] Cortadella J., Kishinevsky M., Grundmann B., "Synthesis of Synchronous Elastic Architectures," Proc. DAC, July 2006, pp. 657–662.

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

[4] Jacobson H.M., Kudva P.N., Bose P., Cook P.W., Schuster S.E., Mercer E.G., Myers C.J., "Synchronous Interlocked Pipelines," Proc. ASYNC, Apr. 2002, pp. 3–12.

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

[6] Li C.-H., Carloni L.P., "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.

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

[8] Li C.-H., Collins R., Sonalkar S., Carloni L.P., "Design, Implementation, and Validation of a New Class of Interface Circuits for Latency-Insensitive Design," Proc. MEMOCODE, May 2007, pp. 13–22.

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

[10] Leiserson C.E., Saxe J.B., "Retiming Synchronous Circuitry", Algorithmica, vol. 6., no. 1, pp. 5–35, 1991.

[11] Kam T., Kishinevsky M., Cortadella J., Galceran-Oms M., "Correct-By-Construction Microarchitectural Pipelining", Proc. ICCAD, Nov. 2008, pp. 434–441.

[12] Carloni L.P., McMillan K.L., Saldanha A., Sangiovanni-Vincentelli A.L., "A Methodology for Correct-by-Construction Latency Insensitive Design," Proc. ICCAD, Nov. 1999, pp. 309–315.

[13] Carloni L.P., McMillan K.L., Sangiovanni-Vincentelli A.L., "Theory of Latency-Insensitive Design," IEEE TCAD, vol. 20, no. 9, Sep. 2001, pp. 1059–1076.

[14] Casu M.R., Macchiarulo L., "A New Approach to Latency-Insensitive Design," Proc. DAC, June 2004, pp. 576–581.

[15] Reese R., Thornton M., Traver C., Hemmendinger D., "Early Evaluation for Performance Enhancement in Phased Logic," IEEE TCAD, vol. 24, no. 4, April 2005, pp. 532–550.

[16] Brej C.F., Garside J.D., “Early Output Logic Using Anti-Tokens,” Proc. Int. Workshop on Logic Synthesis, May 2003, pp. 302–309.

[17] Ampalam M., Singh M., “Counterflow Pipelining: Architectural Support for Preemption in Asynchronous Systems Using Anti-Tokens,” Proc. ICCAD, 2006, pp. 611–618.

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

[19] Carloni L.P., Sangiovanni-Vincentelli A.L., “Performance Analysis and Optimization of Latency Insensitive Systems,” Proc. DAC, 2000, pp. 361–367.

[20] Lu R., Koh C.-K.,“Performance optimization of latency insensitive systems through buffer queue sizing of communication channels,” in Proc. ICCAD, 2003, pp. 227-231.

[21] Lu R., Koh C.-K., “Performance Analysis of Latency-Insensitive Systems,” IEEE TCAD, vol. 25, no. 3, March 2006, pp. 469–483.

[22] Júlvez J., Cortadella J., Kishinevsky M., “Performance Analysis of Concurrent Systems with Early Evaluation,” Proc. ICCAD, Nov. 2006, pp. 448–455.

[23] Bufistov D., Júlvez J., Cortadella J., “Performance Optimization of Elastic Systems Using Buffer Resizing and Buffer Insertion,” Proc. ICCAD, 2008, pp. 442–448.

[24] Ghosh S., Bhunia S., Roy K., “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.

[25] Xiaoyao Liang, Gu-Yeon Wei, Brooks D., “Revival: A Variation-Tolerant Architecture Using Voltage Interpolation and Variable Latency,” IEEE Micro, vol. 29, no. 1, Jan.-Feb. 2009, pp. 127–138.

[26] Bañeres D., Cortadella J., Kishinevsky M., “Variable-Latency Design by Function Speculation,” Proc. DATE, Apr. 2009, pp. 1704–1709.

[27] Austin T., Bertacco V., Blaauw D., Mudge T., “Opportunities and Challenges for Better Than Worst-Case Design,” Proc. ASP-DAC, Jan. 2005, pp. 2–7.

[28] Blaauw D., Kalaiselvan S., Lai K., Wei-Hsiang Ma, Pant S., Tokunaga C., Das S., Bull D., “Razor II: In Situ Error Detection and Correction for PVT and SER Tolerance,” Proc. ISSCC, Feb. 2008, pp. 400–622.

[29] Bowman K.A., Tschanz J.W., Nam Sung Kim, Lee J.C., Wilkerson C.B., Lu S.-L.L., Karnik T., De V.K., “Energy-Efficient and Metastability-Immune Resilient Circuits for Dynamic Variation Tolerance,” IEEE JSSC, vol. 44, no. 1, Jan. 2009, pp. 49–63.

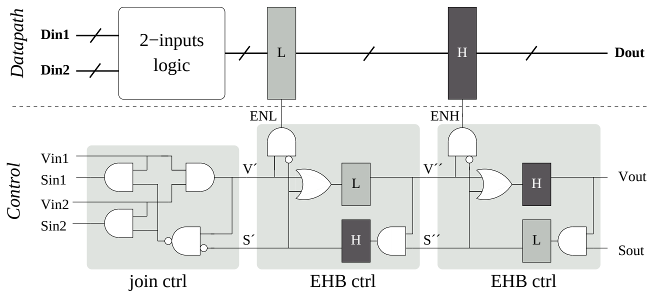

Fig. 1. Elastic datapath and control: 2-input join and elastic half-buffer controllers [2].

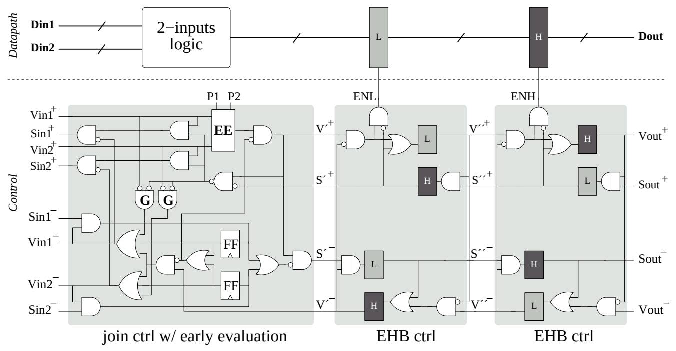

Fig. 2. Elastic control: 2-input join with early evaluation and elastic half-buffer controllers with support for antitokens [3].

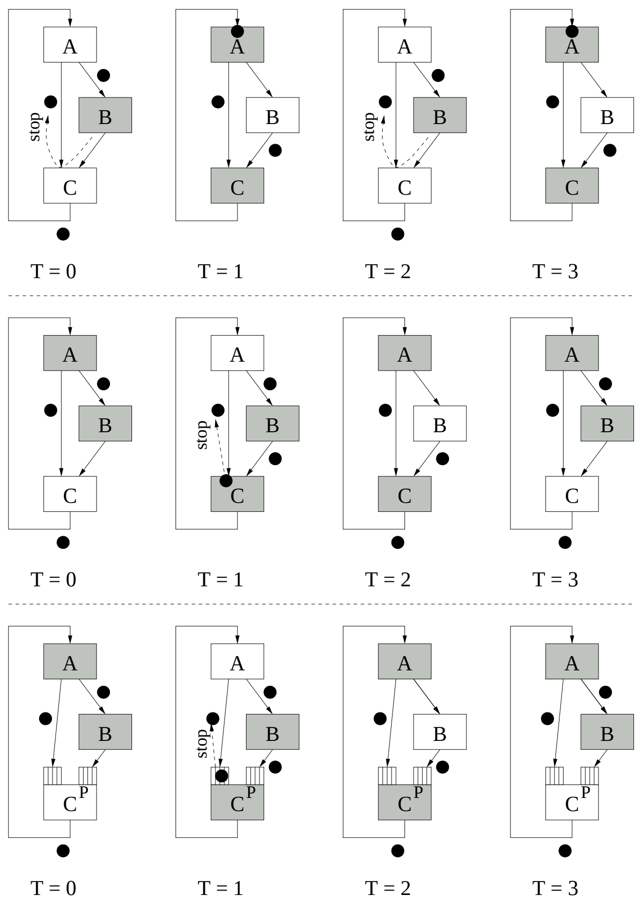

Fig. 3. The bubble bounce problem (top). Bypassable queue inserted in edge A-C (middle). Token cages added on block C's inputs (bottom).

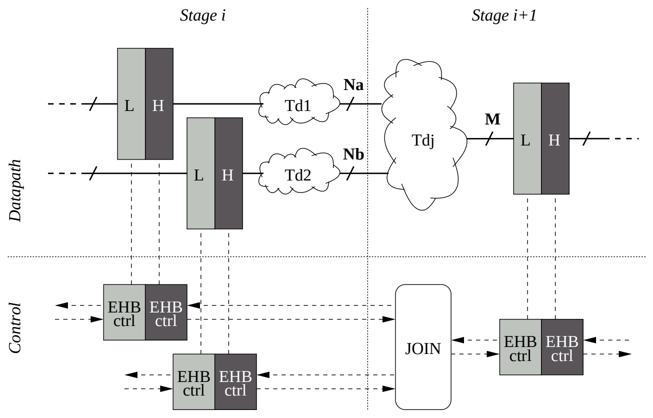

Fig. 4. Datapath: Two data inputs merge on the same output data. Control: The join controller merges/forks valid/stop signals.

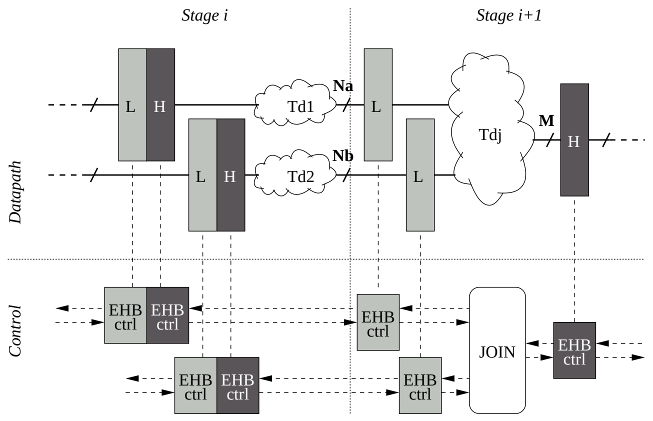

Fig. 5. Two negative-level sensitive latches (datapath layer) and their half-buffers controllers (control layer) have been retimed.

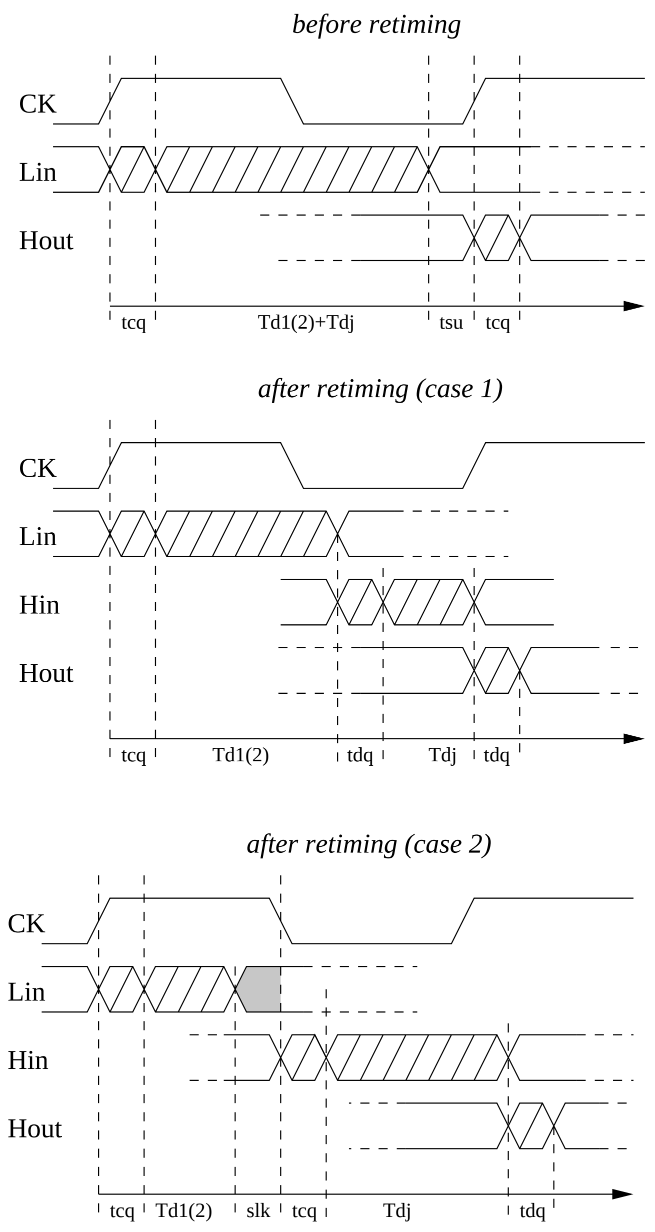

Fig. 6. Timing waveforms before and after retiming.

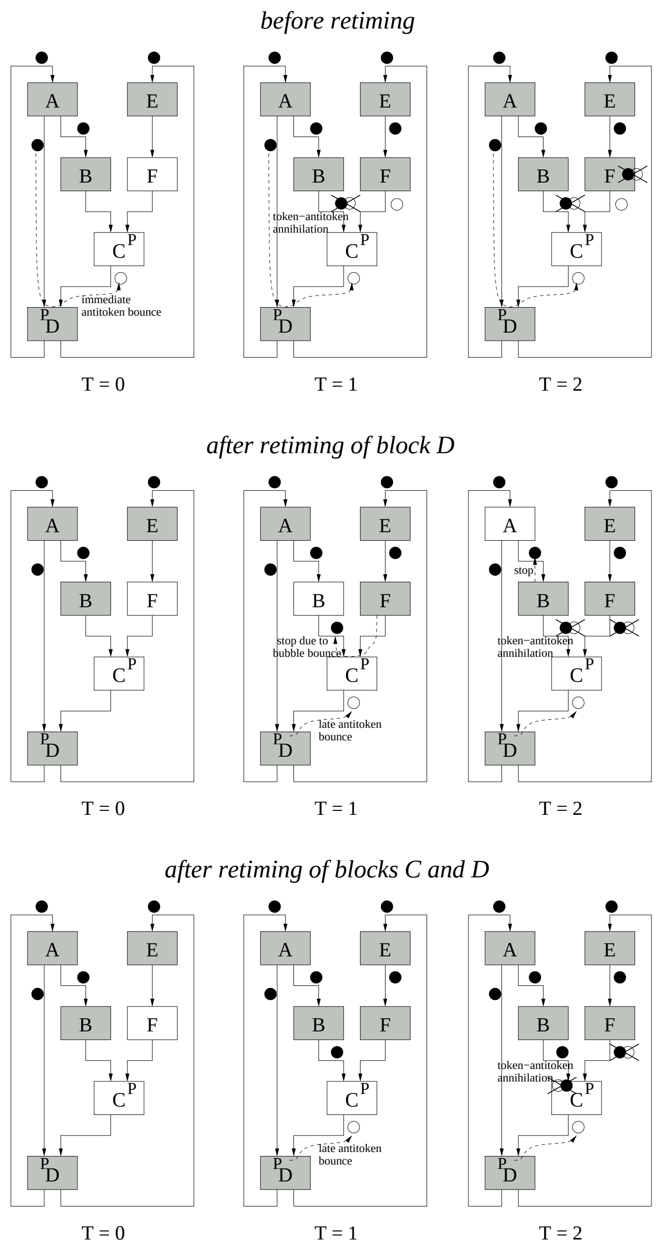
before retiming
immediate antitoken bounce
token−antitoken annihilation
T = 0
T = 1
T = 2
after retiming of block D
stop due to bubble bounce
late antitoken bounce
stop
token−antitoken annihilation
T = 0
T = 1
T = 2
after retiming of blocks C and D
late antitoken bounce
token−antitoken annihilation
T = 0
T = 1
T = 2
A
B
C
D
E
F
P

Fig. 7. Example of timing evolution before and after retiming with early evaluation blocks.

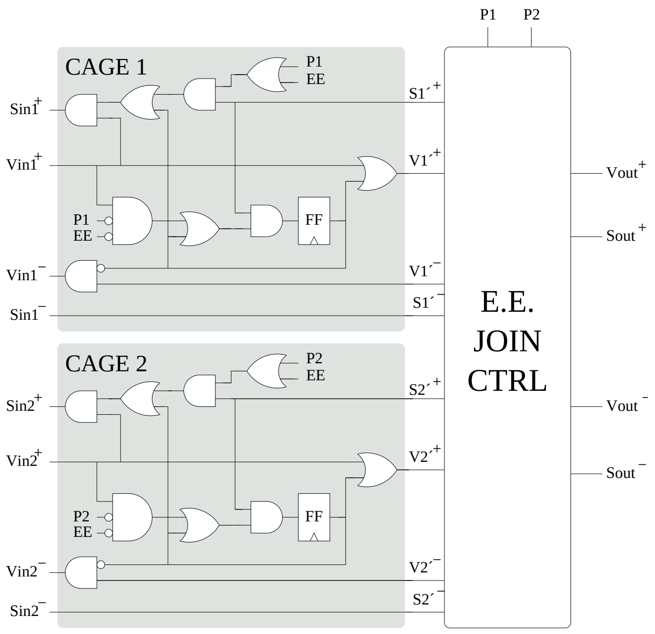

Fig. 8. Insertion of two token cages on the join controller inputs.

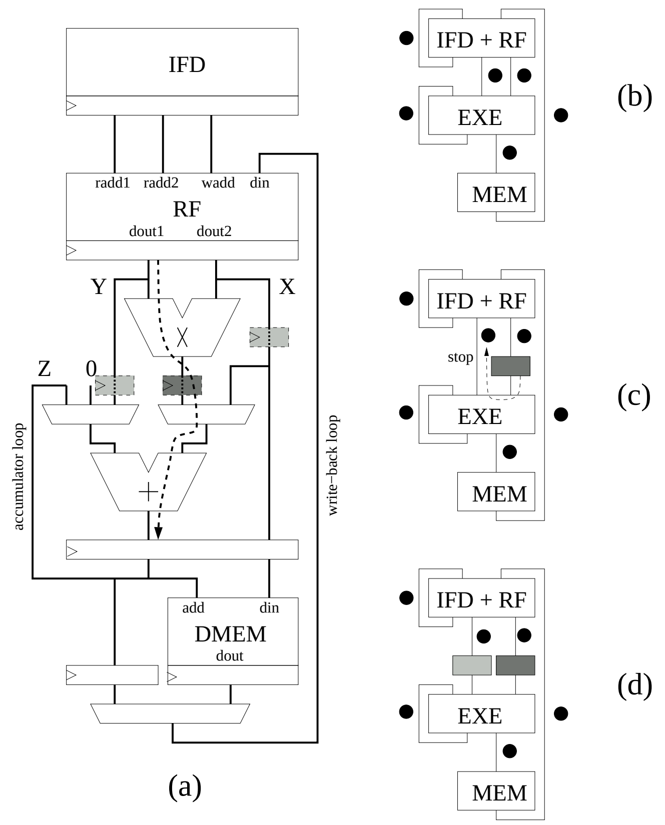

Fig. 9. Example of elastic processor: (a) datapath, (b) marked graph w/o buffers, (c) w/ one buffer to break critical path, (d) w/ a second buffer to balance latencies.

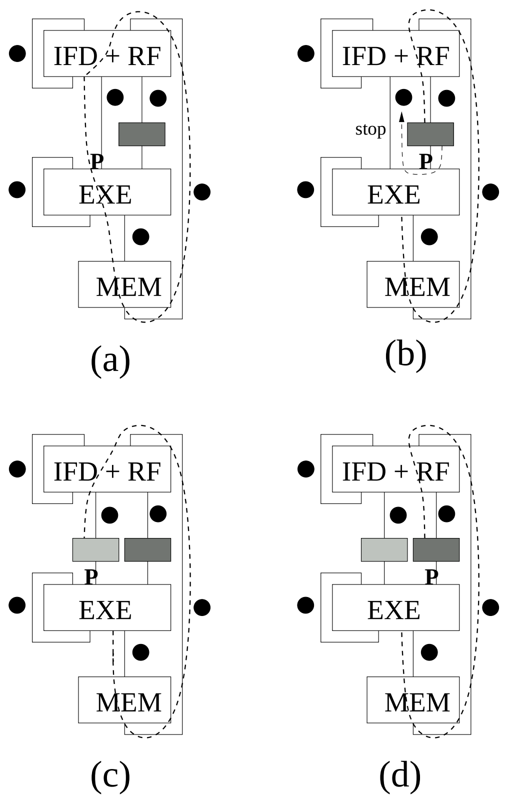

Fig. 10. EXE join with early evaluation and different processing conditions: (a) no buffer and fast input processed, (b) no buffer and slow input, (c) buffer and fast input, (d) buffer and slow input.

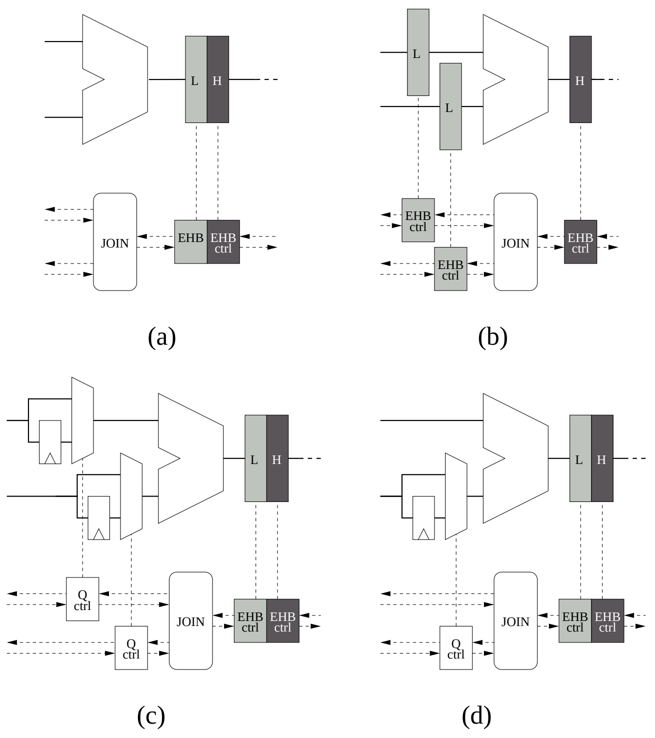

Fig. 11. Four elastic configurations of an arithmetic unit: standard case with logic and elastic buffer (a); input queues created through half-buffer retiming (b); standard input queues (c); only one input queue (d).

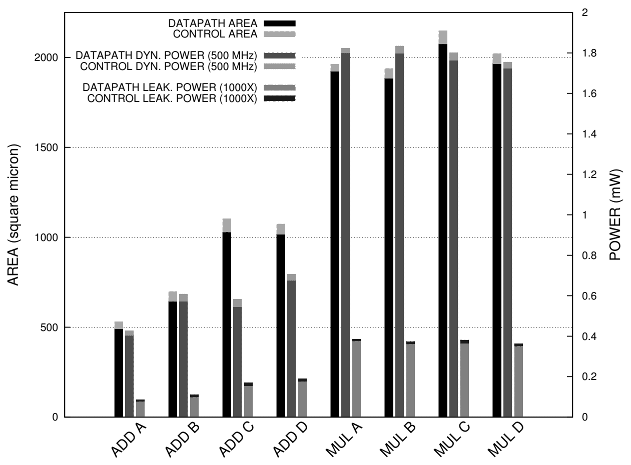

Fig. 12. Area, dynamic and leakage Power (x1000) of four different configurations of adder and multiplier according to the (A,B,C,D) schemes of Figure 11.