Adaptive Latency Insensitive Protocols andElastic Circuits with Early Evaluation: A Comparative Analysis

Adaptive Latency Insensitive Protocols and Elastic Circuits with Early Evaluation: A Comparative Analysis

Mario R. Casu[1]

Politecnico di Torino
C.so Duca degli Abruzzi 24, I-10129 Torino, Italy

Luca Macchiarulo[2]

University of Hawaii at Manoa
2540 Dole Street Honolulu, HI 96822, USA

Abstract

Latency Insensitive Protocols (LIP) and Elastic Circuits (EC) solve the same problem of rendering a design tolerant to additional latencies caused by wires or computational elements. They are performance-limited by a firing semantics that enforces coherency through a lazy evaluation rule: Computation is enabled if all inputs to a block are simultaneously available. Adaptive LIP's (ALIP) and EC with early evaluation (ECEE) increase the performance by relaxing the evaluation rule: Computation is enabled as soon as the subset of inputs needed at a given time is available. Their difference in terms of implementation and behavior in selected cases justifies the need for the comparative analysis reported in this paper. Results have been obtained through simple examples, a single representative case-study already used in the context of both LIP's and EC and through extensive simulations over a suite of benchmarks.

Keywords: Latency Insensitive Design, Elastic Circuits.

1 Introduction

Latency Insensitive Protocols (LIP's) were introduced to derive a functionally correct implementation of a system - designed under the hypothesis of zero latency between modules - in the presence of arbitrary added latencies, such as those deriving from long on-chip wires [1]. Elastic architectures were invented in order to tolerate any latency variability in a design due to either additional wire delays or computational causes [2]. In both cases, the *firing rule* requires availability of all

[1] Email: mario.casu@polito.it
[2] Email: lucam@hawaii.edu

inputs. Both systems can be modeled with Marked Graphs, a subclass of Petri nets, in which functional units are transitions (represented by bars or boxes in this paper) and channels are places (represented by edges connecting the transitions) [3]. Places are marked with *tokens* if they hold valid data. Edges without tokens represent "bubbles" - absence of data. For example, pipelining a wire by adding a clocked repeater will be modeled with an additional node which holds - initially - no data. A functional block will advance computation *iff* all its inputs contain a token (static firing rule). The consequences of such rule in terms of system throughput can be dramatic: If the graph consists of a loop of m places with initial tokens, and n bubbles (places with no tokens) its throughput is $m/(m+n)$. For example, if there are as many bubbles as nodes (e.g. every single wire has been pipelined with a clocked repeater) the throughput will be 1/2: one computation every two cycles.

The restriction on strict input availability is one of the reasons of the performance degradation. Both in the context of LIP's [4][5][6][7] and of Elastic systems [8][9] researchers have tried to get rid of this limitation by allowing nodes to fire if a subset of inputs - or even none of them - is available. As a simple example, consider a *2-way multiplexer* which alternately reads one of its two inputs: Computation can be enabled if the selected input holds a token, regardless of the invalidity of the unselected one. This new firing rule is key to increase the performance of such systems. In the following the inputs that are necessary at any given time in order to perform the computation will be called the *processed* inputs.

This paper proposes a comparative analysis of elastic and latency insensitive systems equipped with this new firing semantics that allows a dynamically *adaptive* computation as opposed to the more *static* and predictable one that comes out as a consequence of the standard firing rule. The comparison is conducted by means of real implementations. As for elastic systems, sufficient details for implementation are given in [9] and [10] which discuss Elastic Circuits with Early Evaluation (ECEE). Concerning LIP's, the so-called Adaptive Latency Insensitive Protocols (ALIP) whose gate-level description is detailed in [6] have been used.

The paper is so organized: Section 2 details analogies and differences of ECEE and ALIP. Section 3 discusses two simple but common configurations for which the differences influence performance. A significant yet sufficiently simple realistic case is used in section 4 to better contrast performances, while in section 5 the two implementations are confronted with a suite of random benchmarks. Conclusions of this work are drawn in section 6, together with hints for future investigations.

2 Analogies and Differences

In this section we describe the two approaches [6] and [9], which will be called ALIP and ECEE in the following. We refer the reader to the original papers for a detailed discussion of their properties. ALIP and ECEE are sufficiently different between each other as their static counterparts LIP and EC (see [2]), to justify a detailed analysis of their pros and cons.

The ECEE constituent blocks are the *elastic controllers* which enable or stop

the clock of a register when computation is or is not allowed. This may occur in case of invalid input, in case of back-pressure (stop) on a valid output and also if a counterflow *killing* information is received on output while the input is valid. A two-phase clocking style allows to make a separate use of the two constituent latches of an edge-triggered register. Multi-input and multi-output blocks require *join* and *fork* structures to feed the elastic controllers. Join controllers allow for both static or adaptive firing semantics. Adaptive firing requires a synchronization of the inputs: A counterflow *kill* signal is sent back to the unselected inputs which were not valid yet. Such information, once received by an elastic controller on its output, will be used to cancel a valid token, or backpropagated further. According to [9], this counterflow signal will be called *anti-token* or *negative token* in the following.

Similarly to standard latency insensitive design, ALIP's wrappers gate the clock, like in the ECEE case, in presence of invalid inputs or back-pressure (stop) on a valid output. Since a single-phase clocking is used, an input queue with at least one place avoids losing data when an output stop is received. Such queue, akin to a skid buffer, can be designed in such a way to avoid additional latency when stops are not received. We will refer to such queue in the following as *half relay station* (HRS), as it contains half as much data registers as regular *relay stations* described in [11]. Firing occurs like in ECEE depending on static/adaptive semantics. However, no kill signals are sent back and the input synchronization is solved locally. The wrapper keeps track of the number of times a new valid input is observed (*computation number*) at each of its inputs, and proceeds to its next computation step *iff* all the inputs needed at any given time are available and their computation numbers are coherent with the ideal clock tick of the computation that has to be performed next. In fact, it is not necessary to keep track of the number of valid inputs since the last system reset, but rather only the relation between them: This is obtained by implementing up-down counters that are controlled by the arrival of valid data on each input, and the execution of a new computation step by the block [6][7].

2.1 Elastic Protocol Implementation

The killing information that ECEE adopts to enforce data coherency is distributed in the system by employing a clever duplication of the normal elastic architecture, and by coupling the two flows in order to annul valid signals. The "second protocol" requires two additional signals: a *Negative Valid*, i.e. the anti-token, conveying the information about computations that need to be killed, and a *Negative Stop* that ensures that no negative valid is lost. Figure 1(a) reports the I/O set of positive and negative signals in a 2in/1out ECEE wrapper and the enable signals ENH and ENL used to control clock gating of transparent latches in a two-phase clocking style. In order to understand how the protocol can locally enforce data coherence, it is expedient to look at the simple example of figure 2(left) in which antitokens are represented as open circles and tokens as filled circles. In the figure, the given configuration of token is supposed to be produced by surrounding modules, and the timing evolution of the protocol is followed (processed inputs are marked with the letter "P"). At time t=1, as only the first input is processed, the protocol requires

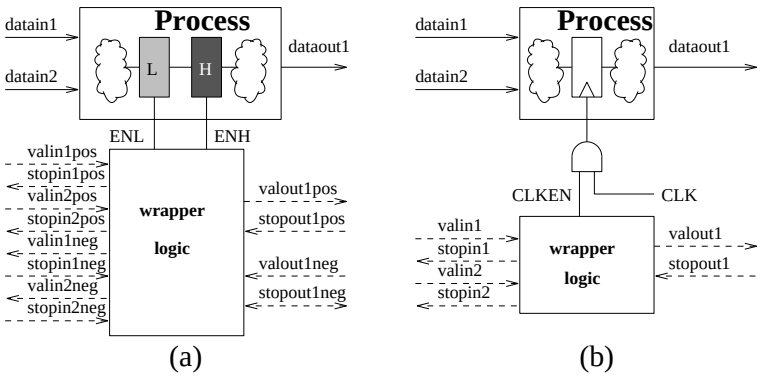

Fig. 1. ECEE(a) and ALIP(b) 2in/1out wrappers.

the generation of 2 antitokens for the other inputs. As input b holds a valid datum, a token and an anti-token meet at the transition, and get annihilated. This is correct, because the input b, though not used, is coherent with input a, and no skew gets accumulated between valid inputs. On the other hand, the antitoken on input c is free to propagate in counterflow. In the following time frame (t=2), input b is needed and two new antitokens are generated as b is valid while no valid data is present at inputs a and c. During the following time frame (t=3), input a is processed again, but in this case no valid data is present on a, and the data on b is stopped. No antitokens are generated. As it will be clear later when ALIP will be discussed, stopping b in this case is not strictly necessary because the datum present on input b is not needed for the computation. However, the ECEE implementation proposed in [9] forces the generation of a stop on each valid input if the selected input is not present, or in general, if the early evaluation does not produce a valid token on the controller's output. One important observation is that the antitokens are actually given a double semantics: **1.** they are used to make sure that valid data that were not used because of an early evaluation are discarded, by forcing an invariant on each loop (sum of tokens - sum of anti-tokens), and **2.** they are interpreted by each module as a kill signal to interrupt the present computation - if any. For example, in the figure, the block marked with an arrow has its computation killed. If we count the number of times a computation is performed at every block since the beginning of the system's operation, it might be the case that, contrary to the ALIP case, *certain blocks have operated less than others*. This fact, that is constitutive of the elastic protocol, has important consequences: Blocks that hold a state, or locally maintain registers that are re-used, need to expose the internal state to the protocol. In practice, if feedback loops are present in a block that has to be wrapped in ECEE style, the output-input loop has to be dealt with like any other channel in the system and provided with the protocol signals. This precaution avoids losing coherency as it is illustrated after the second protocol is analyzed.

2.2 Adaptive LI Protocol Implementation

Figure 1(b) is the ALIP counterpart of the ECEE 2in/1out example of figure 1(a). Only *positive* signals and a single phase clocking style are used. Considering the

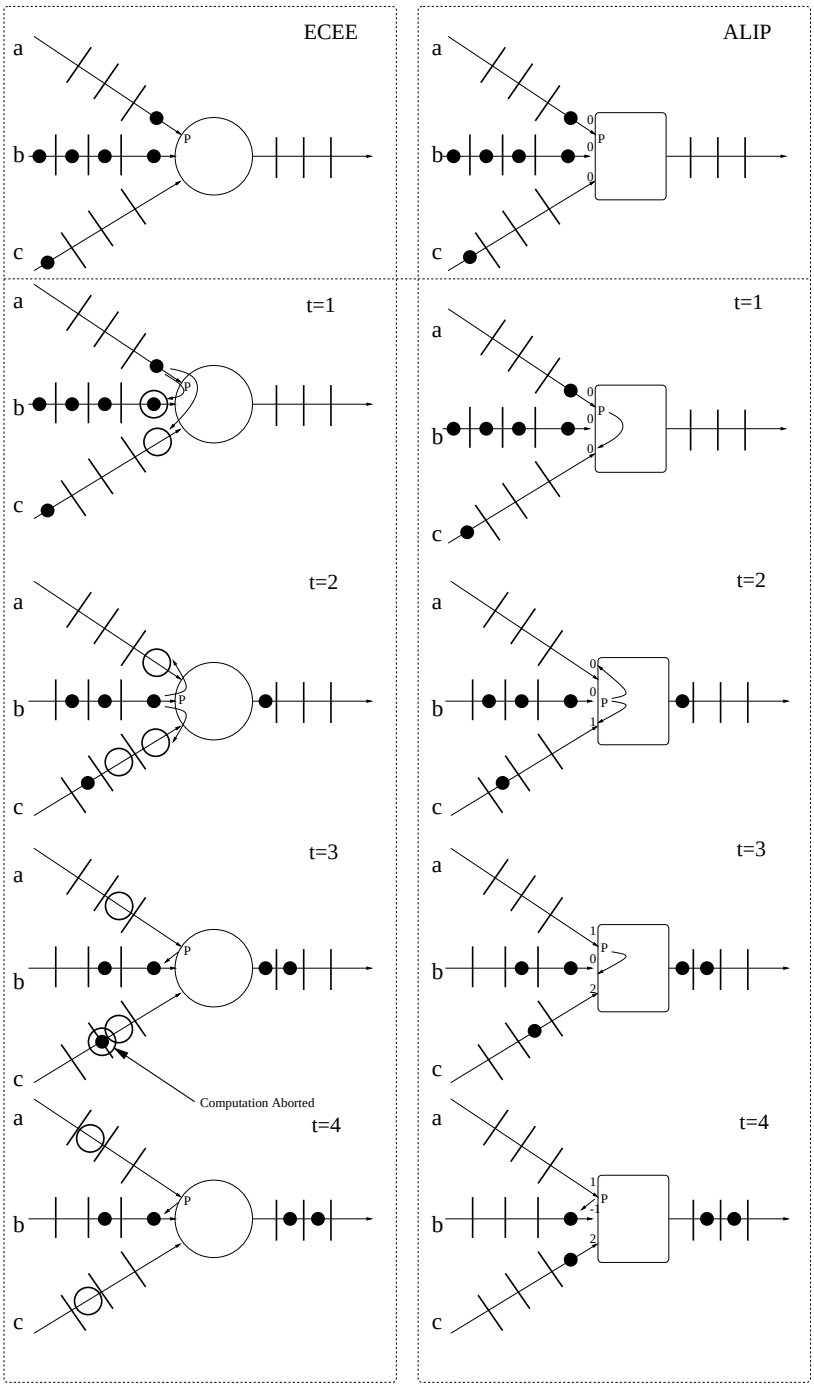

Fig. 2. ECEE and ALIP compared.

same system discussed in the ECEE case, the ALIP evolution is shown in figure 2(right). At the beginning the three inputs are coherent (as in the previous case), so that the relative computation numbers (stored in up/down counters) are all zero, as depicted in the figure. At time t=1, the module is active, because input a is the only processed, and it holds valid data. At the same time, because the unnecessary

input c is not valid, input c has to record a skew between it and the module by incrementing its counter to 1. The value of 1 means that the third input is behind the other two by one step. The counter on b, even though the input was not used by the module, remains at 0 because it contains valid data as well (data discarded). In the second time frame, the processing is on input b, that holds valid data, so the computation is performed, while the counters for inputs a and c have to be incremented because they hold no data. In the third time frame, the input a needs to be processed, but it is not valid, so no computation is done. Input c is also not valid, so that its skew counter is not modified (same relative skew). Contrarily to the ECEE case where valid b is stopped at t=3, here b is safely discarded because it is not processed and its counter is zero (in ECEE, b will be discarded only when a valid datum on selected input a will eventually reach the join controller). The counter on input b is decremented to -1. The new datum on b at t=4 has to be stopped because it is not possible to know if it will be used in the future, not until the processed input misalignment will be reabsorbed. Therefore -1 is the lower limit of the up/down counter. Such first stop on b is absorbed by the previously mentioned HRS which is a one-place FIFO and will store the datum. If another valid datum will eventually appear at the b input at t=5 (not shown in figure), the stop will be propagated back by the HRS which is sized to store just one datum.

2.3 Pros and Cons of ALIP and ECEE

Previous discussion about the behavior of ECEE and ALIP in the example of figure 2 highlighted their analogies and differences. There are conditions in which ALIP reduces the number of stalling signals, like at t=3, due to both the use of counters and of the HRS. It is then likely that some ALIP systems will perform better, and this will be clear in sections 3-4. However, this increase of performance does not come for free, as HRS as well as input counters are area-consuming. We performed logic synthesis of all the elements of the two protocols with a modern CMOS technology, and the results show that the area cost of an HRS is very similar to that of an ECEE join element, if the data parallelism is 16 bit. In general, the area cost for the various experiments we performed are roughly 50% to 100% higher for ALIP, mainly due to the added HRS. It is possible to employ fine-tuning to reduce the impact of HRS and this will be investigated in a future work.

It must be said that HRS are needed only if queues are not already present on the input channels, as it happens when Relay-Stations have already been added. As for counters, their size determines performance, because a stop will inevitably be sent when they reach the maximum count, even if, in general, small counters will guarantee close to best performance, as shown in section 5. This limitation is avoided in ECEE as they can generate as many antitokens as the surrounding system is able to absorb. But on the other hand, this advantage is counterbalanced by the extra-routing needed for antitokens and for negative stops (each point to point communication requires 2 more wires than its ALIP counterpart). It is possible to show that in general not all connections require these extra signals to gain the maximum throughput. But such advantage will result in a similar reduction for

ALIP systems, due to the hardware simplification that ordinary LIP allow. Another drawback of ECEE, for certain design style, is the use of a two-phase clocking style, that increases the cost of timing verification and complicates testing compared to ALIP, let alone the constraint set on the phases' duration and the need to substitute a design's all edge-triggered registers with latch pairs.

Although we did not make any power evaluation, it might be the case that less power is consumed in ECEE - at the same performance of ALIP - due to the possibly higher number of clock-gating events caused by antitokens. But it can also be the case that the extra (possibly long) wires and extra control logic add more power than local counters in ALIP. More work is needed on this point.

As it was anticipated at the end of section 2.1, another major difference between the two approaches is the way gates holding a state, like FSM's registers, are dealt with. Feedback loops internal to the FSM are used to store states in registers. Such loops are often invisible from the FSM's external interface as in the simple example of figure 3 which can be seen as a 1-input/1-output sequential circuit. Imagine also

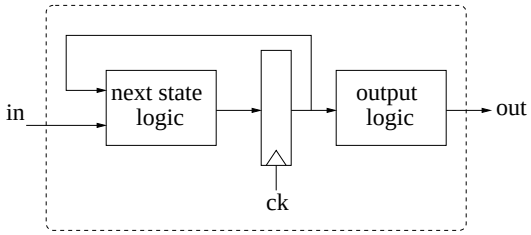

Fig. 3. Simple 1-in/1-out FSM.

to make the FSM elastic by wrapping it with a 1-in/1-out ECEE controller, replacing the register with a pair of latches, and by not touching the inner loop. We model the initial state with a positive token indicating FSM's output validity. According to the ECEE protocol, every time an invalid input is received, the FSM's output will become invalid. Suppose now that at the same time a token and an antitoken are received. If we let the computation be aborted we will lose the possible state change that the valid input may imply, but this is clearly a mistake if we don't want the protocol signals to change the way the system evolves. There are two (provably behaviorally equivalent) ways to solve the problem: **(1)** exposing the feedback loop to the protocol by inserting a 1-in/2-out fork on the register's output and a 2-in/1-out join on the register's input; **(2)** stop all incoming negative tokens at the output. It is still possible to maintain the early evaluation feature of the protocol if the element with state is a join, by letting it *generate* negative tokens, but never *propagate* existing ones. The architectural modifications for the second option are trivial, and consist in substituting the ECEE buffer with an elastic buffer without early evaluation, and interfacing it as detailed in [10]. Exposing the feedback loop to the protocol is relatively easy when the block consists of a single or few state registers. Applying a similar strategy to a more complex situation, like in the case of IP-blocks, requires a perfect knowledge of the block, then somewhat reducing the flexibility of the protocol. Stopping output antitokens might better fit this case.

Blocking the backpropagation of antitokens may lead to a performance limitation

which does not occur in ALIP's where negative tokens are not used: Whatever the case of firing semantics, any FSM can be simply wrapped and provided with its clock gating function without the need to expose the internal feedback to the protocol.

From this discussion it is clear that general rules of better applicability of one protocol with respect to the other are hard to define. In particular cases, like those reported in the next two sections, the supremacy of ALIP in terms of performance emerges clearly. Section 5 will show that the combination of multiple conditions, emulated by means of a set of benchmarks, makes the analysis rather difficult.

3 Simple Examples

In this section we illustrate the cases in which we observed some penalties that affect ECEE compared to ALIP with simple and readily understandable examples. The first one is the classic *reconvergent fanout* case in which the output of a block is sent to two (or more) branches with different latencies. The second one is representative of the class of problems that arise when state holding blocks are involved.

Reconvergent Fanout with static protocols

To begin with, let us assume that block B in the graph depicted in figure 4 need all input to be valid in order to fire. At time t=0 B stalls - it has only one valid

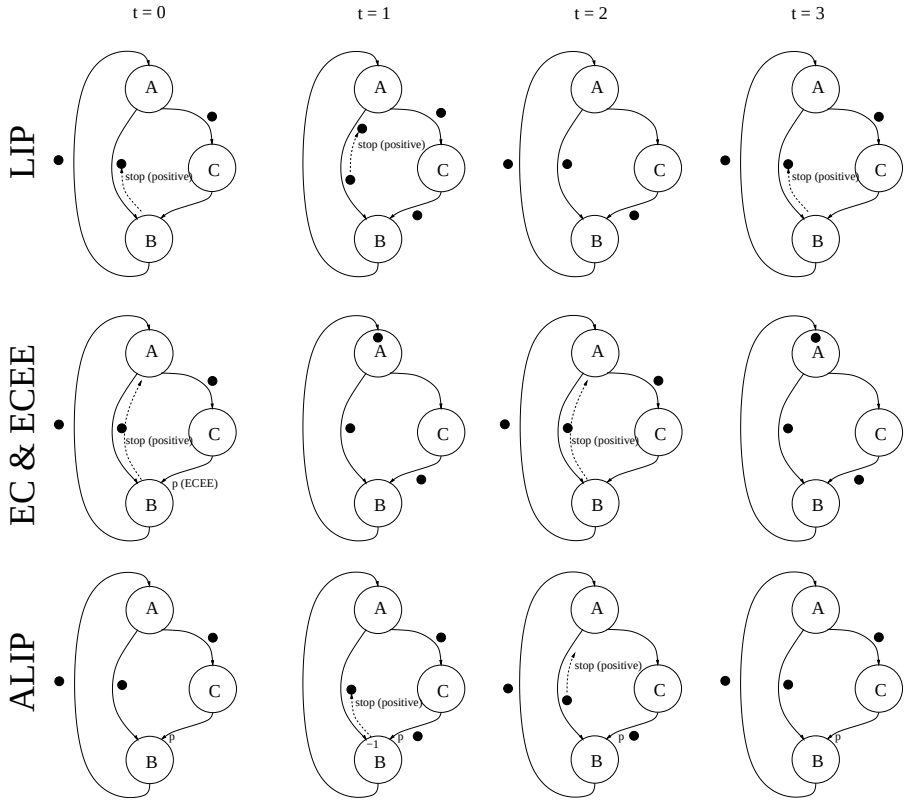

Fig. 4. Example of reconvergent fanout.

input - and stops the token from A. As a result, depending on the back-pressure path latency from B to A, sooner or later block A receives an output stop signal. Now, if the stop signal arrives when there is no token on A's input, its effect on throughput is null, as it does not add any other computation stall. If instead A is ready to fire, the output stop would actually stall A for an additional cycle.

In the latency-insensitive case (LIP in figure), every block's input must be provided with by-passable queues (HRS) that are able to absorb and delay stop signals. If the queue has one place, the stop will reach block A one cycle after B's stall (time t=1). As a result, A receives both an invalid token (due to B stall) and an output stop. Since A would stall in any case, due to the invalid input, the stop has no effect. The system's throughput would be given by the minimum cycle ratio of the corresponding marked graph that amounts to 2/3 in this case: two tokens and a delay of three in the loop A-C-B. Accordingly, after three time steps (t=3 in figure) the graphs gets back to the initial state after every block has issued two tokens.

Contrary to the LIP case in which input queues are necessary for the proper operation of the protocol - and thus are already present - elastic circuits wrappers do not natively include input queues. Therefore, in the example above - t=0 of the EC example in figure 4 - the stop signal from B to A will immediately reach its final destination, leading to an additional stall. The elastic throughput would be 1/2. It can be argued that by-passable queues may be added to elastic circuits when necessary, and this is precisely the point of [13] which discusses throughput optimization by buffer resizing (and insertion). The difference lies then in that in ECEE by-passable elements should be first added and then resized while the first step in ALIP systems is unnecessary as they come with HRS built-in.

Reconvergent Fanout with adaptive systems

Now consider the case in which B reads alternately A and C outputs with an adaptive firing semantics. It is tempting to state that the reconvergent fanout problem has been solved. In fact, if only A's output were chosen, the system's throughput would be given by the ratio between the number of tokens in the loop formed by A and B and the overall latency of the loop, that is 2/2=1. If it were B, than one might similarly expect 2/3 be the throughput, because of the two tokens and the latency of three in the A-B-C loop. However the behavior is more subtle and less intuitive, both in the LIP and elastic cases.

In both cases, B stalls because the selected input from C is not valid. However, the ALIP wrapper (ALIP case in figure 4) discards the valid (not processed) token from A and decreases the counter to -1 - indicating that next token will be an "early" value. At t=1, B receives a valid datum from C and can fire. It also receives an early token from A which has to be stopped. The HRS will delay this stop that will eventually reach A at t=2. Since A's output has no token (A stalled because of the previous B's stall), the stop effect is null. At t=3 the system is in the starting state and every block has done two computations - throughput is still 2/3.

Contrary to the ALIP case, when a processed input is unavailable the ECEE implementation stops a valid datum on the unselected input even though it will never

be used: no anti-token is sent back on the non-processed input because the protocol requires that a output positive token be generated (to maintain the cycle invariant). As a result, if a bypassable queue is not present, the back-propagated stop immediately reaches block A, thus leading to a performance degradation identical to the static case and so to a throughput of 1/2.

When both inputs are selected with 50% probability, the average throughput of ALIP is 3/4, while ECEE performance is limited to 2/3.

Finite State Machine

Block A and C in figure 5 are FSMs like the one in figure 3 and feed block B. If B

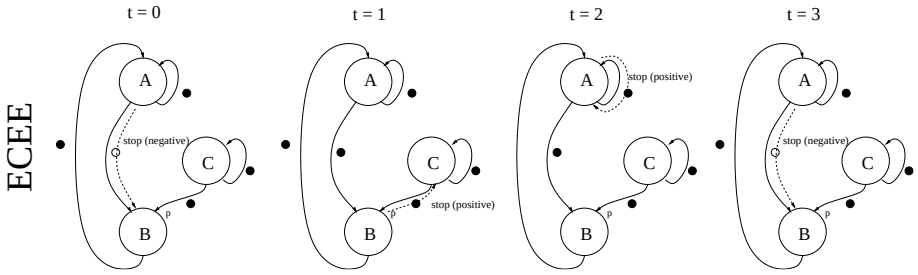

Fig. 5. Example of ECEE FSM behavior.

implemented a static firing rule, a throughput of 1/2 would be obtained. If instead B chose only the input from C, which always holds a valid token, the expected performance would be 1. This is precisely what happens in the ALIP case. In ECEE, when B fires, an antitoken is sent back to A. The particular combination of a non valid token and an antitoken in the link to A, together with the absence of an antitoken in the FSM loop leads to a "negative" stop of the generated antitoken as depicted in figure, t=0. The implementation of the "join" element with early evaluation described in [9] is such that in this situation, at t=1 block B stalls and sends a stop toward C, because no further antitoken can be generated. As a result, performance degradation ensues. The stop is sent once every three cycles, leading to an ECEE throughput of 2/3, 1/3 less than ALIP.

It is worth mentioning that such phenomenon, for static protocols, can be solved with extra buffers or buffer sizing [13]. However, adaptive protocols might depend on more subtle data-related interaction between the various parts of the system that cannot be as efficiently modeled and optimized out. Furthermore, such optimizations could be substantially expensive either in hardware or optimization time.

4 The DLX case

The DLX processor has been used in various disguises in the context of LIP's and elastic circuits as a benchmarking tool [1][2][8]. We implemented a slightly modified version of the graph reported in [2] both in ALIP and ECEE style, with the aim of building a more complex and significant example than the simplest ones discussed

in the previous section. Although still far from a real implementation of a "latency-insensitive" processor, our embodiment allows to reproduce some of the issues of in-order pipelined processors, like structural hazards, data and control dependencies.

In particular, we inserted two additional units in parallel with the already present 0-latency ALU: a pipelined unit (e.g. a pipelined multiplier) and a multi-cycle unit (e.g. a non-pipelined divider). The decoded instruction activates one of the execution units. The alert reader will notice that such "case switch" is out of the scope of Marked Graphs' theory (but not outside the scope of the protocols, as they do respond to data dependent behavior). Nonetheless the ensemble of the execution units, seen from the I/O perspective, behave as a 1-in/1-out dataflow operator which perfectly complies with the theory. The system's graph is shown in figure 6. The switch-select couple which encloses the three units is designed in such a way to respect the theory. In particular, it guarantees the execution order by stalling "fast" instructions while "slow" ones are on their way (e.g. an ALU operation which follows a just started pipelined multiplication with latency greater than 1 or a multi-cycle division must wait). Basically, tokens entering the execution units exit in the same order in a first-in first-out fashion, irrespective of the operation latency.

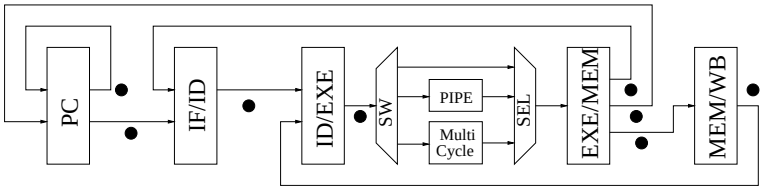

Fig. 6. The graph corresponding to the DLX pipeline.

All DLX pipeline registers (IF/ID, ID/EXE, EXE/MEM, MEM/WB) and the program counter (PC) hold a token. The pipelined unit is initialized with as many bubbles as its internal pipeline stages. The multi-cycle unit is devoid of tokens as well. The blocks with two inputs implement an adaptive firing rule. As an example, the ID/EXE stage always requires the input from IF/ID while the input from MEM/WB is needed only in case of data-dependency. Similarly, the PC and IF/ID stages speculatively proceed unless a control dependency occurs which requires observing the input from EXE/MEM (branch taken or not).

We implemented an ALIP and an ECEE version of this simplified DLX in behavioral VHDL which allows to vary parameters, such as the mix of instructions, the percentage of data-dependency, the possibility to send instruction bursts (e.g. many consecutive pipelined MUL operations followed by many ADD ones). With this setting we were able to gather many interesting data and compare the behavior of ALIP and ECEE in a rather complex yet still manageable example.

We obtained the curve labeled "ALIP & ECEE 100% data dependency" in figure 7(left) by simulating the system in a situation of complete data-dependency: Every issued instruction required a source operand that had not been written back yet in the register file. In this basic implementation we did not include any pipeline by-pass, therefore the token on the IF/ID stage's output is stopped and consumed

only when a token arrives from the MEM/WB stage [3]. In other words, the ID/EXE stage implements a static firing policy. Four out of five curves (labeled ALIP, ECEE

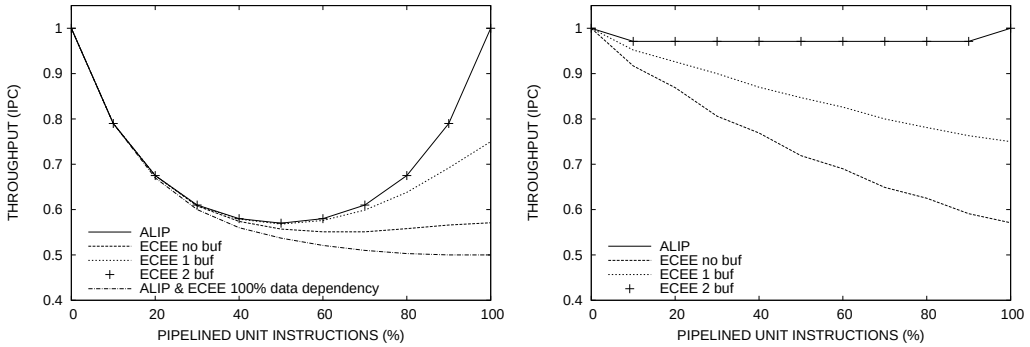

Fig. 7. DLX throughput as a function of the pipelined instruction fraction. Random instructions (left) and burst of instructions (right).

no buf, 1 buf and 2 buf) were instead obtained in case of no data-dependency. In this case every issued instruction already finds all required operands in the register file (MEM/WB-ID/EXE link ignored, adaptive firing rule).

Both in case of data dependency and of independence we varied the percentage of instructions that use the pipelined unit from 0% to 100% in steps of 10%. In this experiment, the latency of the pipelined unit was set to 3, while the multi-cycle unit was never used (execution latency of all instructions 0 or 3 cycles). We discuss first the results in the extreme cases (0% and 100%) and then the intermediate ones.

Data independence, 0% and 100% pipelined instructions

As expected, the throughput is 1 if the pipelined unit is never used (0%). One expects also that the throughput be 1 even when the pipelined unit is fully used (100%). The ALIP implementation in fact reaches maximum throughput (ALIP labeled curve), while ECEE does not (ECEE no buf, throughput 0.571 at 100% pipelined instructions). The reason why ECEE contradicts the expectation is a latency mismatch between the antitokens sent back by PC and IF/ID to EXE/MEM and the antitoken sent by ID/EXE back to MEM/WB. The former ones arrive earlier and must wait (negative stops are exerted in this case, similarly to what happens in the FSM case depicted in figure 5). As it is well known, buffer insertion can be used for latency equalization. The curves labeled "ECEE 1 buf" and "ECEE 2 buf" were obtained by inserting one and two elastic buffers along the EXE/MEM-PC and EXE/MEM-IF/ID links. Both solutions increase the ECEE performance but two buffers are necessary in order to attain a throughput of 1 in the 100% case.

Data independence, 10%-90% pipelined instructions

Figure 7(left) shows that structural hazards in this in-order processor reduce throughput when a mix of pipelined and non-pipelined instructions is fetched. ALIP

[3] Simply but unrealistically, we don't look at the actual required register that is being written back but remove stall as soon as a token arrives. We will complicate the control behavior in a future work in order to mimic a more realistic processor.

and ECEE without buffers behave similarly in range 10%-50% but ALIP is "faster" in rising back to the maximum throughput in range 50%-90%. The ECEE case with two buffers is just as good as the ALIP case (the two curves perfectly overlap).

Data dependency

In the case of complete data-dependency (that is no early evaluation in the ID/EXE stage), the corresponding curve in figure shows that ALIP and ECEE behave exactly in the same way (that is why we reported a single curve). Increasing the instructions that use the pipeline unit reduces the throughput down to 1/2 in the 100% case, as can be easily calculated from the ratio between the number of tokens in the execution loop and the overall latency (3 and 6).

In the simulations reported in figure 7, left plot, both ECEE and ALIP are limited in performance by the structural hazard, while ECEE suffers also from the aforementioned latency unbalance. In order to decouple the two issues, we made an experiment in which instructions of the same type are issued in bursts instead of in a random way as it was for the previous case. Figure 7, right plot, reports the results for the case of data independence. When we lessen the common performance limiter, the difference between ALIP and standard (no buffer) ECEE grows, meaning that the latency mismatch has a large impact. Again, two buffers are needed to close the gap.

Finally we tested the multi-cycle unit by issuing only multi-cycle instructions in case of dependency and independence. As expected, the performance is $1/ML$, that is the inverse of the multi-cycle latency, in both cases and both for ALIP and ECEE.

The previous relevant DLX case exemplifies some of the remarks of previous sections concerning the better potential for performance in the ALIP case. However this does not prevent from possible better performance of ECEE in particular cases. Complex interactions between blocks in a topology with a larger number of components might favor the ECEE case, as next section shows.

5 Results over a suite of benchmarks

In order to perform a comparison over a sizeable set of potential topologies, we randomly generated benchmarks that consist in circuit graphs with differing numbers of gates and proportion between 2-input and 1-input gates. The circuits were generated by making sure that the graph is strongly connected (this pass is necessary for ECEE that assumes strongly connected components). Each module has been assigned an initial token randomly, and each 2-input node has been implemented with a mux-like adaptive module or a normal 2-input module (that works under the static hypothesis). The branch chosen by the adaptive components was decided randomly at each computation with a probability of 0.5. We tested a total of four combinations for each benchmark, obtained by selecting (through a simple greedy

heuristic) **(1)** either a minimal number of tokens to guarantee liveness (at least one token must be initially present in every topological cycle), or a percentage of 50% modules with a token (always guaranteeing the liveness condition); and **(2)** different number of multiplexers: 100% and 50% of the 2-input modules, randomly selected. These four combinations were tested (the exact same netlist, initial token distribution and identical random seeds for each selection) with 3 different protocols: ALIP, ECEE and non adaptive LIP (indicated as *static* in the tables). A fourth case, EC with no early evaluation performed as the static LIP system and is therefore not reported. The test consisted in generating the VHDL code of the fully instrumented systems, and running 10,000 cycles of simulation in order to get minimum and maximum throughput (defined as number of performed computation in the ALIP case, and using the formula reported in [10], for the case of ECEE). During the simulations, assertions where used to check for protocol coherency.

The results in tables 1-2 report the throughput measured in the various experiments. *Blocks* and *joins* columns report the number of components in the graph and the number of 2-input components for each benchmark. Boldface figures represent the best result for a benchmark - when not present two values are sufficiently close to each other - less than 1%. The ALIP counters were chosen sufficiently small (4 bits), so that the excess area due to their presence was kept to a minimum. It did not appear to substantially decrease the performance. The experiments confirmed expectations, for the greater part: the static case is always worse than (in certain cases just as good as) any other protocol, and the potential gains, for both ALIP and ECEE cases can be quite significant (up to almost 300% for certain benchmarks).

As far as the comparison between the ALIP and the ECEE protocol goes, on average ALIP performs better than ECEE. There are a small number of cases, though, in which ECEE outperforms ALIP. We believe that there might be particular topologies combined with patterns of processing that excite some conditions in which mutual interaction between blocks, and in particular killing of upstream blocks, favors ECEE compared to ALIP. Unfortunately, we weren't able to reproduce this type of behavior with simple and intuitively understandable examples: Future investigations are planned that will address this issue.

6 Conclusions

In this paper we reviewed and compared Adaptive Latency Insensitive Protocols (ALIP) and Elastic Circuits with Early Evaluation (ECEE). The comparison was aimed at clarifying the rationale for the difference that occurred in simple examples and in the specific DLX case in which ALIP exhibits a performance advantage. The general case seems to confirm this trend, but some randomly generated benchmarks show a preference toward ECEE whose exact cause remains to be investigated. It is important to consider that such advantage can be an artifact of the random benchmarks suite, therefore extra work will be performed both in the complete analysis of conditions leading to an advantage to ALIP or ECEE, and in ensuring benchmarking for real-life topologies (extensions of DLX to more complex or different circuit

Table 1
Random netlists simulation results: A = ALIP, E = ECEE, Token = MIN.

				MUX = 50%		MUX = 100%	
Bench	blocks	joins	Static Thr	A Thr	E Thr	A Thr	E Thr
r1	6	1	0.20	0.20	0.20	0.22	0.22
r2	8	8	0.25	0.34	**0.35**	0.42	**0.44**
r3	9	2	0.25	0.25	0.25	**0.27**	0.26
r4	9	2	0.13	**0.16**	0.15	0.18	0.18
r5	9	3	0.17	0.17	0.17	**0.26**	0.22
r6	11	11	0.20	**0.26**	0.25	0.33	**0.37**
r7	15	15	0.25	0.27	0.27	**0.38**	0.36
r8	13	4	0.11	0.11	0.11	0.17	0.17
r9	13	5	0.20	0.24	**0.25**	0.28	**0.30**
r10	13	7	0.13	0.14	0.14	0.17	**0.19**
r11	14	4	0.13	0.13	0.13	**0.17**	0.14
r12	20	2	0.17	0.27	**0.30**	**0.39**	0.34
r13	16	4	0.08	0.09	0.09	**0.14**	0.12
r14	16	8	0.09	0.11	0.11	**0.18**	0.15
r15	18	8	0.09	**0.11**	0.10	**0.14**	0.13
r16	20	8	0.13	0.16	0.16	0.21	0.21
r17	21	5	0.17	0.17	0.17	0.20	**0.25**
r18	22	22	0.09	0.19	0.19	**0.33**	0.30
r19	25	25	0.11	0.13	0.13	**0.30**	0.28
r20	28	28	0.17	**0.19**	0.18	**0.31**	0.25
r21	28	28	0.09	**0.16**	0.15	**0.31**	0.26
r22	35	35	0.11	0.14	0.14	**0.27**	0.26
arithmetic average			0.15	0.18 (+20%)	0.18 (+20%)	**0.26 (+71%)**	0.25 (+64%)
harmonic average			0.13	0.16 (+20%)	0.16 (+20%)	**0.23 (+75%)**	0.22 (+65%)

Table 2
Random netlists simulation results: A = ALIP, E = ECEE, Token = 50%.

				MUX = 50%		MUX = 100%	
Bench	blocks	joins	Static Thr	A Thr	E Thr	A Thr	E Thr
r1	6	1	0.40	0.40	0.40	0.44	0.44
r2	8	8	0.33	**0.45**	0.40	0.56	**0.58**
r3	9	2	0.57	0.57	0.57	**0.62**	0.61
r4	9	2	0.33	0.41	0.41	0.45	**0.46**
r5	9	3	0.5	0.5	0.5	0.5	0.5
r6	11	11	0.33	0.33	0.33	**0.60**	0.58
r7	15	15	0.33	**0.46**	0.34	**0.59**	0.52
r8	13	4	0.50	0.50	0.50	**0.59**	0.53
r9	13	5	0.55	**0.57**	0.50	**0.59**	0.54
r10	13	7	0.29	0.29	0.29	0.33	0.33
r11	14	4	0.38	0.38	0.38	**0.46**	0.44
r12	20	20	0.50	0.50	0.50	**0.66**	0.58
r13	16	4	0.33	**0.43**	0.27	**0.50**	0.33
r14	16	8	0.43	**0.45**	0.44	**0.53**	0.47
r15	18	8	0.33	**0.38**	0.37	**0.43**	0.42
r16	20	8	0.25	0.25	0.25	**0.40**	0.38
r17	21	5	0.50	0.50	0.50	0.56	**0.57**
r18	22	22	0.50	0.50	0.50	**0.60**	0.55
r19	25	25	0.40	**0.45**	0.57	**0.57**	0.48
r20	28	28	0.33	0.33	0.33	**0.56**	0.43
r21	28	28	0.40	**0.43**	0.40	**0.57**	0.51
r22	35	35	0.29	**0.34**	0.31	**0.48**	0.40
arithmetic average			0.40	**0.43 (+7%)**	0.40 (+1%)	**0.53 (+32%)**	0.48 (+21%)
harmonic average			0.38	**0.41 (+8%)**	0.38 (+1%)	**0.51 (+35%)**	0.47 (+24%)

graphs). In terms of complexity, ECEE outperforms ALIP, but detailed logic synthesis results have not been discussed and will be the subject of a future analysis. However, most of the ECEE's area gain is due to the two-phase clocking choice, an option that not all designers will pursue with a light heart. Furthermore, it is likely that a two-phase clocking redesign of ALIP will result in a significant area saving.

As in the general case it was not possible to discern which protocol is better, this calls for further analysis that helps select the best strategy given a particular design rather than the best overall. In any case, the results show that both protocols significantly increase the performance over the solutions without early evaluation.

References

[1] L.P. Carloni *et al.* "Theory of Latency-Insensitive Design," IEEE TCAD, vol. 20, No. 9, Sept. 2001, pp. 1059-1076.

[2] J. Cortadella *et al.*, "Synthesis of Synchronous Elastic Architectures," Proc. 43rd Design Automation Conf. (DAC 06), 2006, pp. 657-662.

[3] T. Murata, "Petri Nets: Properties, analysis and applications," Proc. IEEE, pp. 541-580, Apr. 1989.

[4] M. Singh and M. Theobald, "Generalized Latency-Insensitive Systems for Single-Clock and Multi-Clock Architectures," Proc. Design, Automation and Test in Europe Conf. (DATE 04), vol. 2, pp. 1008-1013.

[5] A. Agiwal and M. Singh, "An Architecture and a Wrapper Synthesis Approach for Multi-Clock Latency-Insensitive Systems," Proc. Intl Conf. Computer-Aided Design (ICCAD 05), pp. 1006-1013.

[6] M.R. Casu and L. Macchiarulo, "Adaptive Latency-Insensitive Protocols," IEEE Design and Test of Computers, Sep.-Oct. 2007, pp. 442-452.

[7] C.H. Li and L.P. Carloni, "Using Functional Independence Conditions to Optimize the Performance of Latency-Insensitive Systems," Proc. Intl Conf. Computer-Aided Design (ICCAD 07), pp. 32-39.

[8] J. Júlvez *et al.*, "Performance analysis of concurrent systems with early evaluation," Proc. Intl Conf. Computer-Aided Design (ICCAD 06), pp. 448-455.

[9] J. Cortadella and M. Kishinevsky, "Synchronous Elastic Circuits with Early Evaluation and Token Counterflow," Proc. 44th Design Automation Conf. (DAC 07), pp. 416-419.

[10] J. Cortadella and M. Kishinevsky, "Synchronous elastic circuits with early evaluation and token counterflow," Technical Report LSI-07-13-R, 2007. www.lsi.upc.edu/techreps/files/R07-13.zip.

[11] L.P. Carloni *et al.*, "A Methodology for "Correct-by-Construction" Latency Insensitive Design", Proc. Intl Conf. Computer-Aided Design (ICCAD 99), pp. 309-315.

[12] R.L. Collins and L.P. Carloni, "Topology-Based Optimization of Maximal Sustainable Throughput in a Latency-Insensitive System", Proc. 44th Design Automation Conf. (DAC 07), pp. 410-415.

[13] D. Bufistov *et al.*, "Performance optimization of elastic systems using buffer resizing and buffer insertion,", Proc. Intl Conf. Computer-Aided Design (ICCAD 08), pp. 442-448.