Elastic Systems

Jordi Cortadella
Universitat Politècnica de Catalunya
Barcelona, Spain

Marc Galceran-Oms
Universitat Politècnica de Catalunya
Barcelona, Spain

Mike Kishinevsky
Strategic CAD Lab, Intel Corporation.
Hillsboro, OR USA

***Abstract*—Elastic systems provide tolerance to the variations in computation and communication delays. The incorporation of elasticity opens new opportunities for optimization using new correct-by-construction transformations that cannot be applied to rigid non-elastic systems.**

The basics of synchronous and asynchronous elastic systems will be reviewed. A set of behavior-preserving transformations will be presented: retiming, recycling, early evaluation, variable-latency units and speculative execution. The application of these transformations for performance and power optimization will be discussed. Finally, a novel framework for microarchitectural exploration will be introduced, showing that the optimal pipelining of a circuit can be automatically obtained by using the previous transformations.

I. Introduction

Since the early days of Electronic Design Automation, designers have been using optimization tools to improve the quality of the circuits (area, delay or power). For example, the techniques for two-level and multi-level *combinational logic synthesis* exploit the properties of Boolean algebra to transform gate netlists within the boundaries of the sequential elements of the circuit. With this approach, the behavior of the primary outputs and state signals is preserved, whereas the combinational parts are optimized.

A new generation of techniques enabled the crossing of the sequential boundaries and introduced new optimizations that can change the behavior of the state signals while preserving the behavior at the primary outputs. This field is known as *sequential logic synthesis* and includes transformations such as state encoding, redundant latch removal, and retiming.

All the previous transformations still preserve a *cycle-accurate* behavioral equivalence of the system: what is observable at the i-th clock cycle is independent from the optimizations performed on the system.

The scaling down to nanometric technologies is inherently associated to an increase of complexity of the systems that can be integrated in a chip. The clock network becomes more sophisticated, variability increases and timing closure turns to be a challenging problem with no trivial solution.

Maintaining the cycle accuracy imposes severe constraints on the type of optimizations that can be used in a circuit. For example, changing the structure of a pipeline or modifying the latency of a functional unit may not be applicable unless the global architecture of the system is transformed and adapted to the new timing requirements. This limitation is unacceptable for large systems that may be susceptible to late-stage re-design decisions to meet the timing specifications.

Elasticity has emerged as a new paradigm to overcome these limitations, enabling the design of systems that are tolerant to

(a)

cycle	1	2	3	4	5	6
a	3	1	2	3	1	0
b	5	0	4	6	2	4
$a+b$	8	1	6	9	3	4

(b)

cycle	1	2	3	4	5	6	7	8	9	10
a	3		1	2		3	1		0	
b		5	0	4	6			2	4	
$a+b$		8		1	6		9	3		4

Fig. 1. (a) Non-elastic behavior, (b) Elastic behavior.

the dynamic changes of the computation and communication delays. The concept of elasticity has been largely used in asynchronous circuits. For example, the term *Micropipeline* was proposed by Sutherland [40] to denote event-driven elastic pipelines. The tolerance to delay variability requires the incorporation of handshake signals to synchronize the agents that communicate through a channel. These handshake signals are often called *request* and *acknowledge*.

When elasticity was discretized to work with synchronous systems, the term *Latency Insensitivity* was coined [9]. In these systems, the handshaking is produced at the level of cycle with events that are synchronized with the clock. A pair of handshake signals, typically called *valid* and *stop*, indicate the validity of the data in the communication channels and the back-pressure produced by stalled units. Different variants of synchronous elasticity have been proposed later [19], [42].

With elasticity, the concept of behavioral equivalence is relaxed in a way that no cycle accuracy is required. Instead, the sequence of *valid* data is observed and preserved, as if one would connect FIFOs with non-deterministic delays at the inputs and outputs of the system.

Figure 1(a) shows an example of the timing of a non-elastic circuit performing additions. The circuit receives inputs and produces outputs at every cycle. The environment is designed under the assumption that all operations will take one cycle.

Figure 1(b) shows an example of the timing of an elastic version of the same circuit, in which the empty cells represent non-valid data. It can be observed that the traces of valid data for each signal are the same as in the non-elastic circuit. Although the timing is elastic, the causality relations between data items is preserved, e.g., the value $a+b$ is always generated after having received the corresponding values of a and b.

A. Microarchitectural transformations

Elasticity opens the door to a new avenue of correct-by-construction behavior-preserving transformations for optimizing systems that cannot be systematically applied in the non-elastic context. The insertion of “empty” sequential elements in a pipeline, the execution of variable-latency computations,

978-1-4244-7886-6/10/$26.00 ©2010 IEEE

the addition of bypass circuits or the speculative execution of operations are some of the transformations that can be considered to increase the performance of a circuit.

By using an appropriate kit of elastic transformations, different architectures can be derived for the same functionality, thus enabling the capability of creating microarchitectural exploration engines that can guide the design of high-performance and low-power circuits.

This paper gives an overview of the underlying concepts that support elastic systems and describes a set of transformations for microarchitectural exploration that can be used to obtain efficient pipelines from functional specifications. An automated microarchitectural exploration engine is discussed and the efficacy of this approach is illustrated with an example.

II. Elastic Systems

When designing elastic systems one needs to address a few design decisions.

Granularity. It is possible to use the idea of elasticity at different levels of granularity. On the one extreme, a system can be composed from large synchronous blocks with elastic communication between them in order to use modularity or power advantage of running blocks at different clock frequencies or in anticipation of design variations in communication channels. On the other extreme, one can consider a synchronous system as a set of gates. Every gate is then viewed as an elastic island and the system can be redesigned to become elastic at the gate level. While such a design can tolerate changes in latencies or delays at the very low level of granularity, the cost of elasticity may become prohibitive. In this paper for illustration purposes we focus at the the intermediate point within the range of possible design partitioning: we view design at the register transfer level and consider every register (and its elastic counterpart, an elastic buffer) as an object of interest. This view is similar to the one used in retiming. However, even with focus on RTL level we can easily group different registers together or decompose them if it is beneficial for performance or area of the design. The block level view using different forms of elastic shells around existing blocks is presented in [10], [41], and [18, §8].

Communication protocol. A particular handshake protocol between elastic blocks should be selected. Section II-A will discuss one for synchronous elastic systems.

Elastic FIFOs. They serve as buffers between communicating elastic islands. Since a back-pressure (or an acknowledgment) signal that informs the sender that the receiver is busy takes some latency (or delay) to propagate it is necessary to give the sender some flexibility in sending data items while the back-pressure signal is in flight. To guarantee that the sent items are not lost they must be captured within the elastic buffers at the boundary between the sender and the receiver or within the receiver. Multiple constructions of elastic FIFOs have been studied in asynchronous and synchronous design communities (see, e.g., [16]). In Section II-B we will give an example of low overhead implementation of elastic buffers that can be used for replacing synchronous registers.

Elastic controllers. The data and control flow is managed by elastic controllers. While in some design styles a controller can be completely merged with the data-path based on using delay-insensitive codes, our objective is to use the data-path that is almost identical with the data-path of a standard synchronous system. This achieves low overhead and enables the use of a standard design methodology and tools. In this case a controller can be viewed as a separate distributed clock-gating layer that decides on stop and go of the elastic blocks. Note that even though logically we think of a separate controller, physically it can be merged with blocks of the data-path. A few details of such controllers are described in Section II-C.

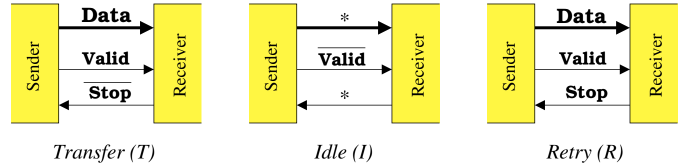

Fig. 2. SELF protocol states

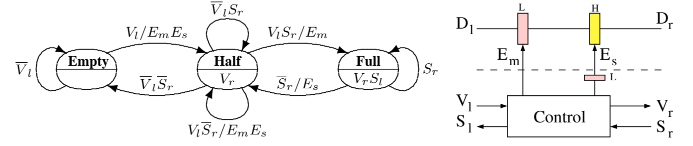

Fig. 3. Control specification for the latch-based EB.

A. A synchronous elastic protocol

To discuss the characteristics of synchronous elastic protocols, we will focus on the one presented in [19], [27] (SELF: Synchronous Elastic Flow). This protocol is similar to the one presented in [24], [34] and it is conceptually similar, but not equivalent, to the one presented in [12].

The two handshake signals, *valid* (V) and *stop* (S) determine three possible states in an elastic channel (see Fig. 2):

Transfer, ($V \wedge \neg S$): the sender provides valid data and the receiver accepts it.

Idle, ($\neg V$): the sender does not provide valid data.

Retry, ($V \wedge S$): the sender provides valid data but the receiver does not accept it.

The sender has a *persistent* behavior when a *Retry* cycle is produced: it maintains the valid data until the receiver is able to read it. For example, the input channel a of the adder in Fig. 1(b) is in transfer state during cycle 9, but in idle or retry state during cycle 10. Not enough information about valid/stop signals is shown in this figure to distinguish between the two.

We will also differentiate between useful pieces of data inside elastic channels or elastic buffers ("data tokens") and pieces of data that can be ignored ("bubbles").

Token, (V): a useful piece of data that should be processed.

Bubble, ($\neg V$): a don't care piece of data that can be ignored.

We will depict tokens as dots inside the boxes representing elastic buffers, while bubbles are depicted as empty boxes.

B. Elastic buffers

Figure 3 depicts the FSM specifications for the control of a latch-based elastic buffer (EB) and the overall structure of the design. The transparent latches are shown with single boxes, labeled with the phase of the clock, L (active low) or H (active

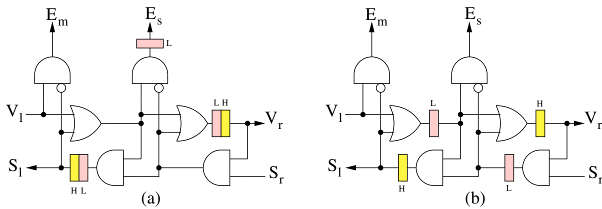

Fig. 4. Two implementations of an EB control.

high). The flip-flops are drawn as two transparent latches back-to-back corresponding to their master-slave implementation. The control drives latches with enable signals. To simplify the drawing, the clock lines are not shown. An enable signal for transparent latches must be emitted on the opposite phase and be stable during the active phase of the latch. Thus, the E_s signal for the slave latch is emitted on the L phase.

The FSM specification of Fig. 3 is simply the specification of a 2-slot FIFO. In the *Empty* state, no valid data is stored in the latches. In the *Half-full* state, the slave latch stores one valid data item. Finally, in the *Full* state, both latches store valid data and the EB emits a *stop* to the sender. The specification is a mixed Moore/Mealy FSM with some output emissions associated with the states and some other with the transitions. For example, the transition from the *Half-full* state to the *Full* state occurs when the input channel carries valid information (V_l is high), but the output channel is blocked (S_r is high). An output signal enabling the master latch is emitted (E_m). In addition the valid bit is emitted to the output channel ($V_r = 1$), since this Moore style signal is emitted at any transition from the *Half-full* state.

After encoding *Empty*, *Half-full*, and *Full* states of this FSM with $(V_r, S_l) = (0,0),(1,0),(1,1)$ correspondingly we can derive the implementation shown in Fig. 4(a), where flip-flops are drawn as two back-to-back transparent latches. By splitting the flip-flops and retiming the latches, a fully symmetric latch-based implementation can be obtained (Fig. 4(b)).

Elastic buffers with capacity two can be designed in a similar way using different constructions of the data-path, e.g. using ping-pong registers composed to a multiplexer (see [18] for details). Buffers of larger capacity are typically designed as a FIFO with read and write pointers and latency of one clock cycle between the read and the write. For zero latency elastic buffers, FIFOs with a bypass are used.

C. Join and Fork

In general, EBs can have multiple input/output channels. This can be supported by using elastic *Fork* and *Join* control structures. Figure 5(a) shows an implementation of a Join. The output valid signal is only asserted when both inputs are valid. Otherwise, the incoming valid inputs are stopped. This construction allows composing multiple Joins together in a tree-like structure.

Figure 5(b) depicts a *Lazy Fork*. The controller waits for both receivers to be ready ($S = 0$) before sending the data[1]. A more efficient structure shown in Fig. 5(c), the *Eager Fork*, can send data to each receiver independently as soon as it is ready to accept it. The two flip-flops are required to "remember"

[1]This implementation is identical to the one presented in [24].

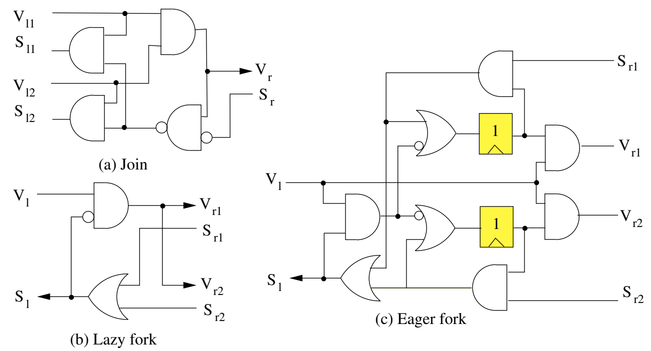

Fig. 5. Controllers for elastic Join and Forks.

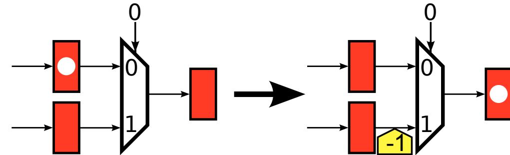

Fig. 6. Firing of an early evaluation node

which output channels already received the data. This structure offers performance advantages when the two output channels have different back-pressures.

D. Early evaluation

The join controller introduced above is based on *late evaluation*: the computation in the corresponding block is initiated only when all inputs are available. Sometimes this requirement is too strict. Consider a multiplexor with the following behavior:

$$o = \texttt{if } s \texttt{ then } a \texttt{ else } b.$$

If the value s is known, then it is only necessary to wait for the required token in order to compute o. If, for instance, s and a are available and the value of s is *true*, the result $o = a$ can be produced without waiting for b to arrive and the value of b can be discarded when it arrives at the multiplexor.

Early evaluation takes advantage of this flexibility to improve system performance. A care must be taken of the late arriving irrelevant tokens to avoid spurious enabling of functional units. When early evaluation occurs, a negative token, also called *anti-token*, is generated in the late channels that were not using for enabling the block, as shown in Fig. 6. When an anti-token and a token meet in the same channel, they cancel each other.

Anti-tokens can be *passive*, waiting for the token to arrive inside the counters associated with an early evaluation join controller, or *active*, traveling backwards through the control until they meet a token. [17] describes a method of implementing elastic systems with early evaluation (both in the active and the passive form). [29] suggests a method for implementing early evaluation based on the analysis of don't care conditions of the data-path logic. Their implementation, like the one proposed in [14], is of a passive form. Implementation of anti-tokens can be optimized using token cages [15]. Elastic FIFOs can store and propagate anti-tokens the same way they store and propagate tokens. There are also some implementations for asynchronous circuits [5], [36], [1].

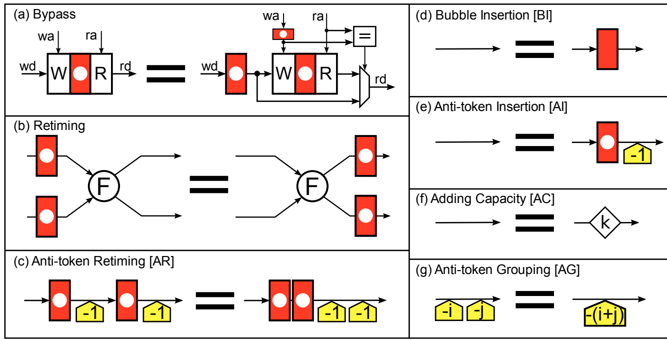

Fig. 7. Correct-by-construction transformations [26]

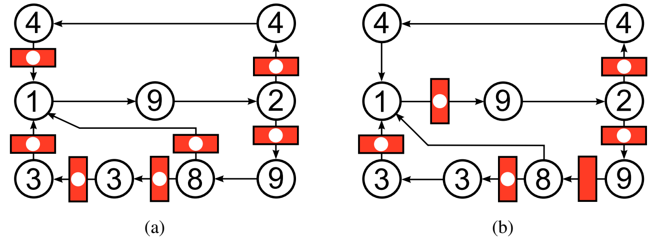

Fig. 8. Design optimized using (a) Retiming, (b) Retiming and recycling

III. TRANSFORMATIONS

One of the major features of synchronous elastic systems is their tolerance to latency changes. Such tolerance can be used to introduce novel correct-by-construction transformations enabling the exploration of new microarchitectural trade-offs [26]. Figure 7 shows bubble insertion (**BI**), adding capacity (**AC**), anti-token insertion (**AI**), anti-token grouping (**AG**) and anti-token retiming (**AR**), as well as the elastic version of classical bypass and retiming. In this figure (and future ones), elastic channels are represented as arrows, and elastic buffers as boxes (with a dot if they contain a token). Control details are not explicitly displayed, only the data dependencies are drawn.

The transformations presented in this section can modify the latency of the communications and computations of an elastic system while preserving its functionality. In some cases, the cycle time of the system can be reduced by increasing the latency of some operations. By properly balancing cycle time and throughput, the system with the optimal effective cycle time can be achieved[2].

A. Latency Preserving Transformations

Most design transformations used in conventional synchronous systems can also be applied in elastic systems.

1) Bypass: At a certain level of abstraction, a register file can be represented by a monolithic register and additional logic to write (W) and read (R) data. In Fig. 7(a), the channels wd and rd represent data, whereas the channels wa and ra represent addresses.

Bypasses, which were already used in the late 50s [4], are widely used to resolve data hazards in processors [23]. Figure 7(a) shows a register file after a bypass transformation. One EB delays the write operation, and a forwarding path is added, so that if the read address is equal to the write address of the previous operation (RAW dependency), the correct data value can be propagated, even though it has not been written in the register file yet. The area overhead of the bypass transformation is one multiplexor, one unit to detect dependencies by comparing the write and read address, and two EBs to delay the write data and the write address channels. The multiplexor selecting between the forwarded data and the register file data is typically an early evaluation multiplexor.

2) Retiming: Registers can be moved across combinational logic preserving the functionality, as shown in Fig. 7(b). Retiming [28] is a traditional technique for sequential area and delay optimization. It can also be used to reduce the power of a system [32]. The initial value of the register must be taken into account if it is important for the correct initialization of the system.

Figure 8(a) shows an example with an optimal retiming. The combinational nodes (shown as circles) are labeled with their delays. The boxes (labeled with a dot) represent the elastic buffers with tokens of information (i.e. registers with valid data inside). The cycle time of this design is 17 time units.

B. Recycling

It is always possible to insert and remove an empty EB (a bubble) on any channel of an elastic system (for formal proof see [27]). This transformation is shown in Fig. 7(d).

Bubble insertion is also known as recycling, and was initially introduced in [13]. The concept of inserting empty buffers for optimizing system performance was long known in asynchronous design [43], [31]. In [2], [35], exact algorithms for slack matching on choice-free asynchronous systems were presented. This problem is similar to solving the recycling problem. Moreover, in synchronous elastic designs, recycling can be combined with retiming leading to a more powerful design optimization [11], [8].

Figure 8(b) shows an optimal configuration combining retiming and recycling for the example from Fig. 8(a). The cycle time has been reduced to 11 units. The throughput is determined by the slowest cycle. The token to register ratios for each cycle are 1, 4/5 and 2/3. Therefore, the throughput is 2/3, and the average number of cycles to process a token is 3/2. This gives an *effective cycle time* of 16.5 time units ($16.5 = 11 \cdot 3/2$). It means that a new token is processed on average every 16.5 time units - an improvement compared to the 17 units of the optimally retimed design.

C. Early Evaluation

Introducing early evaluation in some nodes can be considered an optimization technique that allows firing a token even if some of the inputs are not available.

The performance of a system with early evaluation is no longer determined by the slowest cycle, since average-case performance is achieved instead of worst-case. There is no known efficient exact method to compute the throughput of a system with early evaluation. An upper bound method using linear programming is presented in [25]. Each input must be

[2]The effective cycle time is a performance measure similar to the time-per-instruction, TPI, in CPU design. It captures how much time is required to process one token of information - the smaller the better.

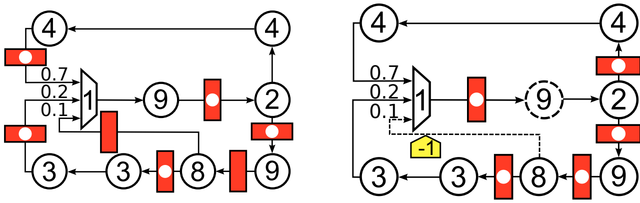

Fig. 9. Optimal configuration with (a) early evaluation and (b) anti-token insertion

assigned a probability so that performance can be analyzed. Such probabilities should be obtained by running a typical application on the system, and then counting how often each input is selected.

Figure 9(a) shows the example from Fig. 8 after adding early evaluation to the node with delay 1, and adding a bubble on one of the input channels of the multiplexor. Let us assume that the upper channel of the multiplexor is selected with probability 0.7, the middle one with probability 0.2 and the lower one with probability 0.1.

Retiming and recycling can be successfully applied for systems with early evaluation to achieve better performance [6]. The example from Fig. 9(a) has a cycle time of 10 time units, which is lower than the 11 units in Fig. 8(b). If there were no early evaluation, its throughput would be 0.5, determined by the slowest cycle. Then, the effective cycle time would be 20 units - worse than the 16.5 units obtained for the previous configuration without early evaluation. However, when early evaluation is introduced, the cycle with the worse throughput is only selected by the multiplexor in 10% of the cases. If the system is simulated using the given probabilities, the obtained throughput is 0.79. Thus, in this example, early evaluation allows one to reduce the effective cycle time to 12.65 units $(10/0.79)$ by using early evaluation.

D. Anti-token Insertion

Anti-tokens are used to cancel spurious computations in early evaluation nodes, but they can also be used to enable new retiming configurations. An empty EB is equivalent to an EB with one token of information followed by an anti-token injector with one anti-token (drawn as a pentagon), as shown in Fig. 7(e). When a token flows through a non-empty anti-token injector, the token and the anti-token cancel each other.

Anti-token counters can be retimed (as in Fig. 7(c)) and grouped (as in Fig. 7(g)). When retiming anti-tokens, care must be taken with the initial values of the registers so that functionality does not change.

Anti-token insertion can be often applied to enable retiming of EBs that have been initialized with a different number of tokens. For example, Fig. 9(b) shows a system where anti-token insertion has been applied to the dashed channel. Then, the new EB can be retimed backwards. This new configuration has a cycle time of 11 units, but its throughput is very high, 0.918, since there is only one cycle with a bubble (a sum of a token and an anti-token is equal to zero) as compared to Fig. 9(a), where two of the three cycles have bubbles. The resulting effective cycle time is 11.98 units. This configuration can only be achieved by using the anti-token insertion transformation.

E. Variable-latency units

Variable-latency units can be handled in a natural way in synchronous elastic systems. A handshake with the datapath unit is required so that the control can keep track of the status of operation, as shown in [19].

For example, an ALU may spend one clock cycle to compute frequent operations with small operands (i.e. operands with few significant digits), and spend two clock cycles for rare operations involving larger operands. This is a typical example of a telescopic unit [3], [39]. Variable latency units can improve the performance by decreasing the overall cycle time, and they can also improve the area of the design by reducing the number of logic gates per pipeline stage.

Other examples of variable-latency units are large register files with access time ranging from one to two cycles for different partitions, variable hit time caches (so called pseudo-associative caches), video decoding blocks processing different video symbols with largely varying probabilities and any other operation where there is a significant discrepancy between the typical and the worst case pattern of operation.

In the example from Fig. 9(b), the critical cycle is determined by the dashed node with delay 9 followed by the node with delay 2. Assume we can replace the dashed node with a variable-latency node that has a typical delay of 7 time units at the cost of spending an extra cycle (i.e. 14 time units) in rare cases. Then, the cycle time of the system will drop from 11 to 9 units.

Let us assume that the short operation can be applied 95% of the times. Then, the throughput of the system is 0.881, estimated by simulating the controller. The resulting effective cycle time is 10.216 units (9/0.881), compared to the previous 11.98. Overall, correct-by-construction transformation that modify latency have provided a 66% improvement in performance for this example.

F. Buffer Capacities

While the **BI** rule in Fig. 7(d) is formulated for the elastic buffer with capacity two, it holds for the elastic buffer of any capacity $k \geq 0$. Moreover, if the latency of the buffer is equal to 0 (implementable as a FIFO with a bypass), the performance of the design as measured by the throughput cannot decrease. The rhombus in transformation **AC** in Fig. 7(f) stands for a 0-latency buffer (a so called skid buffer) with capacity k.

In some cases, adding capacity can prevent a deadlock or increase the performance of the system. Consider retiming as an example. In standard synchronous systems retiming moves registers through combinational logic. In elastic systems retiming moves EBs instead of simple registers. Each of such retiming moves involves moving the data tokens residing inside the EBs as well as capacity of the EBs. Consider a fragment of an elastic system shown in Fig. 10(a). After applying backward retiming through node F, one obtains the system in Fig. 10(b). This system has a deadlock, as can be seen by the analysis of the corresponding marked graph shown in Fig. 10(c)[3]. There are two back-pressure edges (shown with

[3]To be more precise this model corresponds to the dual guarded marked graphs capable of modeling early-evaluation and anti-tokens [25]. However the intuitive analysis does not require detailed knowledge of this model.

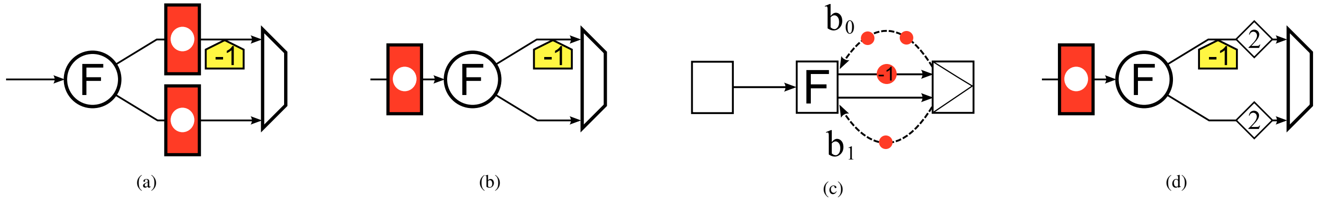

Fig. 10. (a) Example before retiming, (b) after retiming with deadlock, (c) after retiming with conservative capacity sizing, (d) marked graph that illustrates why the deadlock appears

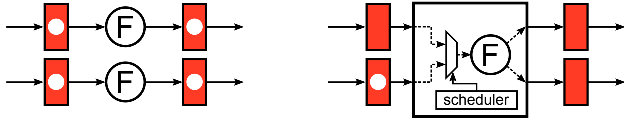

Fig. 11. (a) Logical view of a shared unit, F is considered a variable-latency unit, (b) physical view of a shared unit, the scheduler of the shared unit grants access to one of the channels

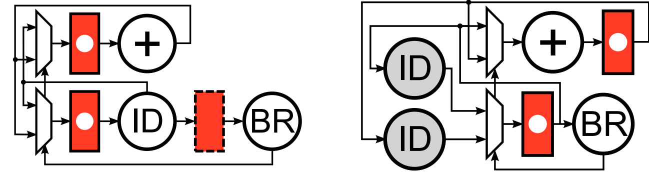

Fig. 12. Branch prediction using shared units

dotted lines) going from the multiplexor node to node F. The upper one, b_0, is associated with the upper forward arc. The sum of tokens on the cycle they form is equal to 1 (2 - 1 = 1) that models the fact that the node F can fire only once without firing the multiplexor node. The lower backpressure arc, b_1, is associated with the lower forward arc, and it has 1 token so that the sum of tokens on this cycle is also 1. However, consider the cycle composed by the upper forward edge and b_1. The sum of tokens on this cycle is zero, a characteristic property of a deadlock in a marked graph [33].

In order to avoid this deadlock, it is sufficient to add a skid-buffer of capacity one to the lower channel. Then, b_1 would have one more token (with two tokens total) due to the capacity of the skid-buffer. In general, the optimal capacity for the skid buffers can be obtained by solving an ILP problem [30]. Resizing of these capacities can also be combined with recycling as an optimization procedure [7]. Early evaluation may increase the required size of the buffers for some examples [26]. However, instead of solving an optimization problem after each retiming move, we can conservatively add capacity of 2 to the channels where the `EB`s were placed before the retiming, as shown in Fig. 10(d). This corresponds to applying a correct-by-construction transformation AC(2) before the retiming move and then performing the move. It can be guaranteed that the resulting system is deadlock free, since the capacity of the channels is never reduced. At the end of the exploration, the optimal buffer sizes can be found by solving the optimization problem once.

G. Sharing of Functional Units

When a system includes early evaluation, some computations are not always required, and hence, they can be delayed for some cycles with no performance penalty or even canceled. As a result the actual utilization of some units can be way below 100%.

Different modules with the same behavior (for example, two adders in the design), can be merged into a single module, which is then shared by the input channels that compete for this resource. Sharing may provide a reduction of area and power in the design, hopefully at a low (or zero) performance degradation. Using a shared module is like using a module followed by a buffer with unbounded but finite latency, since each data token may have to wait for a certain number of cycles until it is allowed to use the shared module.

A local scheduler decides at each clock cycle which input channel can use the shared resource. The performance variation compared to using unshared resources depends on whether the scheduler can distribute the load accurately among the different users. For better performance, the scheduler should take into account the elastic protocol: an invalid or a stalled channel cannot use the shared unit even if selected. For correctness a scheduler should be fair avoiding starvation of the channels: every token that reaches the shared module must eventually be allowed to use it unless it is cancelled by an anti-token.

For example, Fig. 11(a) shows two channels, each one using a node called F, which compute exactly the same function. If both F are shared into a single physical entity, the logical view remains the same, although the latency of F becomes variable. Physically, a scheduler selects which channel can use F every clock cycle, as shown in Fig. 11(b).

H. Speculative Execution

Sharing of functional units can be used to implement correct-by-construction speculative execution [20]. Consider the example from Fig. 12(a) showing a possible branch instruction in a microarchitectural graph, if the dotted bubble is ignored. Each time a new instruction address must be generated, it is chosen between the previous one plus a constant, the lower input in the multiplexors, or an address coming from the instruction decoder (ID). The selection depends on the value generated by the node named BR, which may look at some register in the ALU or it may perform some operation indicated by the decoder stage.

Let us assume that ID and BR cannot be executed within one clock cycle because their total delay is too large. In Fig. 12(a), the only way to cut this path is to add the bubble between ID and BR drawn with a dotted line. However, BR, ID and one of the multiplexors form a cycle, and hence, adding this bubble limits the throughput of the design to 0.5. Early evaluation cannot help increasing the throughput in this example.

Given a multiplexor with several inputs, it is possible to move a functional block from the output of the multiplexor to its inputs using Shannon decomposition (viewed also as

a multiplexor retiming) [37]. The example from Fig. 12(a) can be transformed into the design in Fig. 12(b) by using multiplexor retiming and register retiming. In this second design, there is no critical cycle going through the control of the multiplexor, and the only combinational path going through two units is the one formed by BR going to the upper multiplexor and then to the adder. If this path became critical, multiplexor retiming could also be applied to the upper multiplexor.

The performance gain of Fig. 12(b) comes at a cost of duplicating the ID stage, with the resulting area overhead. Here is where speculation comes into play. The two ID nodes can be merged into a single one which is shared by the two inputs of the multiplexor. Hence, each clock cycle the scheduler of the shared ID module must perform a branch prediction, and decide which of the tokens arriving to the ID stage should be granted access to the unit so that the throughput is maximized. The scheduler can implement any state-of-the-art branch prediction algorithms, enhanced to understand the elastic protocol.

Misprediction and correction are handled naturally by the handshake between the shared module and the multiplexor. If the multiplexor requires a channel, and the scheduler predicted the other channel would be needed, the scheduler will see back-pressure coming from the predicted channel and will be able to correct the misprediction.

This speculation framework can also be used to efficiently integrate into elastic systems telescopic units and error correction and detection protocols [20]. Using speculation and anti-token insertion, precomputation [22] can also be added to the set of possible transformations.

I. Verification

One can verify correctness of the previous transformations using model checking. Given the subgraph of the system where the transformation has been applied, it must be verified that the original subgraph is *transfer equivalent* to the transformed subgraph. This can be proven by checking that each channel complies with the handshake protocol before and after the transformation, data token order is preserved and the symbolic function computed by the datapath is preserved at the outputs of the system even if the latency has changed.

For module sharing, the absence of deadlocks has been verified for *any scheduler* that satisfies the fairness assumption [20]. To prove that the fairness of the schedulers is a sufficient condition for liveness, the refinement verification is applied. It is proven that a shared module sequentially composed with an EB that has an undetermined finite response latency is a refinement of an EB specification itself, provided that the shared module has a nondeterministic fair scheduler.

In [27], it is proven that an elastic module connected to another elastic module forms a new elastic module, even if a feedback loop is added (as long as there is at least one token in the loop). The result is proven for elastic systems without early evaluation and anti-tokens. However, it can also be applied to the designs where early evaluation and token counterflow can be encapsulated within an elastic sub-module that has a regular valid/stop handshake interface. In [38] a refinement technique is suggested for verifying that an optimized elastic system complies with the original specification.

IV. Microarchitectural Exploration

Starting with a functional specification graph of a design, it is possible to obtain a pipelined design by using elastic transformations. This section presents a framework to explore different pipelines automatically, trying to optimize any cost function that combines effective cycle time and area.

Bypasses with early evaluation multiplexors are essential for pipelining, since they introduce new EBs that can be retimed backwards. In order to pipeline a design, bypasses must be inserted around register files and memories of the functional model. Then, the graph is modified to enable forwarding to the bypass multiplexors. Finally, the system can be pipelined by retiming the EBs inserted with the bypasses and using other transformations such as recycling or anti-token insertion. Speculation and insertion of variable-latency units can also be considered in the exploration framework.

A. Example: Pipelining a Reduced Instruction Set

Figure 13(a) shows a specification of a simple design. The register file *RF* is the only state holding block. *IFD* fetches instructions and decodes the opcode and register addresses. *ADD* and *M* are arithmetic functions. The results are selected by the multiplexor for *RF* write-back. *M* has been divided into three nodes. The breaking up of logic to allow pipelining is a design decision that is typically considered in concert with pipelining decisions. Thus, the user may try to divide a functional block into several nodes and let the optimization algorithm decide the best edges to place the EBs.

In Fig. 13(b), the bypass transform has been applied three times on *RF* to build a bypass network. The node *DD* receives all previous write addresses and the current read address in order to detect any dependencies and determine which of the inputs of the bypass multiplexor must be selected. The conventional use of bypasses is to forward data already computed to the read port of the bypassed memory element. In addition, this bypass network can be used as a data hazard controller, taking advantage of the underlying elastic handshake protocol with early evaluation to handle stalls.

The right-most multiplexor and the bypass EBs must be duplicated to feed each bypass path independently, enabling new forwarding paths, as shown in Fig. 13(c). Once the forwarding paths have been created, the design can be pipelined by applying retiming and anti-token insertion, achieving the system in Fig. 13(d). The final elastic pipeline is optimal in the sense that its distributed elastic controller inserts the minimum number of stalls. Furthermore the pipeline structure is not redundant since there are no duplicated nodes. Therefore, this is as good as a manually designed pipeline.

Fast instructions that require few cycles to compute, like *ADD* in this example, use the bypass network to forward data avoiding extra stalls, while long instructions use the bypass network as a stall structure that handles data hazards. In this example, the only possible stalls occur when the paths with anti-token counters are selected by the early evaluation multiplexors. This situation corresponds to a read after write

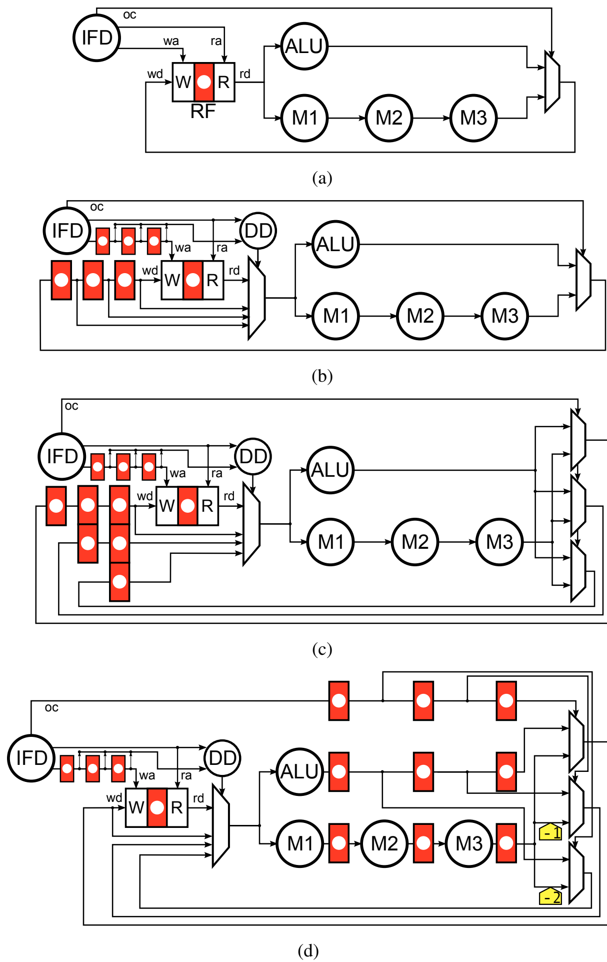

Fig. 13. (a) Graph model of a simple design, (b) After 3 bypasses, (c) Duplicate mux, enable forwarding, (d) Final pipeline after transformations

(RAW) dependency involving a result computed by *M*, which needs three cycles to complete.

B. Exploration Algorithm

The previous example is small enough to allow a manual exploration, but if the microarchitectural graph grows, manual exploration becomes complicated and error-prone.

Most transformations presented in Fig. 7 are captured in the formal retiming and recycling method from [6], which can be solved as a mixed integer linear programming problem (MILP). Capacity sizing and bypassing are the transformations that cannot be captured.

The optimal capacity for elastic channels can be obtained at the end of the exploration by running an ILP problem, as mentioned in Section III-F. Therefore, the parameter to optimize is the number of bypasses that should be applied to every memory element before applying the retiming and recycling optimization.

A heuristic algorithm that explores different number of bypasses for each memory element, such as the one presented in [21], is the best solution in order to efficiently browse through as many configurations as possible. Since the retiming and recycling method uses an upper bound of the throughput instead of the exact throughput, the most promising designs should be simulated at the end of the exploration to identify the best one, or to study a possible trade-off between performance and area or power.

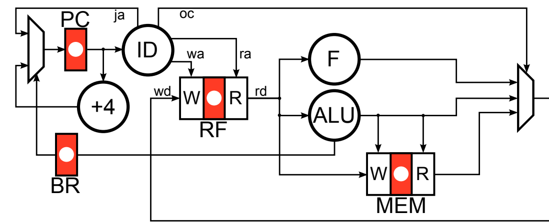

Fig. 14. DLX initial graph

The possibility of using variable-latency units can also be considered in the exploration framework. In order to do so, the retiming and recycling MILP can be extended with a boolean variable for each node which can be implemented as a variable-latency node, to choose whether the node is used in variable-latency mode or not. Depending on the value of the boolean variable, the node is assigned its regular latency and combinational delay, or it is assigned the latency and combinational delay of its variable-latency implementation.

Furthermore, speculation can also be added to the exploration framework. The heuristic to solve the retiming and recycling MILP proposed in [6] provides a set of designs with different Pareto-point trade-offs between cycle time and throughput. For the designs with a small cycle time but also a small throughput, it is easy to determine whether the graph has a cycle going through the control port of a multiplexor which is critical in the throughput of the system. If this is the case, it is possible to try to apply the speculation method presented in [20] and possibly improve the throughput with a small cycle time and area overhead.

C. Pipelining of a DLX

We illustrate this exploration method on a simple microarchitecture similar to a DLX, shown in Fig. 14 before pipelining. The execution part of the pipeline has an integer ALU and a long operation F. The instruction decoder ID produces the opcode, *oc*, that goes to the write-back multiplexor and a target instruction address, ja, that is taken in function of the previous ALU operation, as stored in the register BR. Table I shows approximate delays and area of the functional blocks of the example, taking NAND2 with FO3 as unit delay and unit area. In order to obtain these parameters, some of the blocks have been synthesized in a 65nm technology library using commercial tools (ALU, RF, mux2, EB and +4), and the rest of the values have been estimated. EB and mux2 delay and area numbers were taken for single bit units. The delay of bit-vector multiplexors and EBs is assumed to be the one shown in the table, while area is scaled linearly w.r.t. the number of bits. Multiplexors with a fan-in larger than two are assumed to be formed by a tree of 2-input multiplexors.

The register file is 64 bits wide, with 16 entries, 1 write and 2 read ports. The total footprint of the RF is 6000 units, (including both cell and wire area). To account for wiring of other blocks, we assume that 40% space is reserved for their wiring. Furthermore, we also need to consider the area overhead of elastic controllers. Based on experiments with multiple design points, we assume a 5% area is reserved for the controllers.

TABLE I
DELAY, AREA AND LATENCY NUMBERS FOR DLX EXAMPLE

Block	Delay	Area	Lat.	Block	Delay	Area	Lat.
mux2	1.5	1.5	1	EB	3.15	4.5	1
ID	6.0	72	1	+4	3.75	24	1
ALU	13.0	1600	1	F	80.0	8000	1
RF W	6	6000	1	RF R	11	-	1
MEM W	-	-	1	MEM R	-	-	10

The memory has a read latency of L_{MEM} cycles, which is set to 10 in Table I (corresponds to a realistic L2 read latency). Memory reads are assumed to be non-blocking, i.e., a few reads can be pipelined into a memory subsystem. We do not account for area of the memory subsystem (as it is roughly constant regardless of pipelining).

Figure 16 shows one of the best design points found by the method under the following design parameters: the F unit has been divided into three blocks, the memory data dependency probability is 0.5, and register file data dependency probability is 0.2, the instruction probabilities are: ($p_{\text{ALU}} = 0.35$, $p_{\text{F}} = 0.2$, $p_{\text{LOAD}} = 0.25$, $p_{\text{STORE}} = 0.075$, $p_{\text{BR}} = 0.125$). Data dependency probabilities decrease exponentially as the depth of the dependency increases. Finally, the probability of a taken branch is 0.5. These values are based on experiments found in [23], and they are used in the early-evaluated multiplexors.

In Fig. 16, the cycle time is 29.817 time units, due to the F_0, F_1 and F_2 functional blocks. 3 bypasses have been applied to RF and then EBs have been retimed to pipeline F. Note that an extra bubble has been inserted at the output of F_2: the reduction in the throughput due to this bubble, is compensated by a larger improvement in the cycle time (without this bubble the critical path would include the delay of the multiplexors after F_2). If F operation was used more frequently, the design without this bubble might be better in performance, since the bubble would have a higher impact on the throughput. Such decisions are made automatically based on the expected frequencies of instructions and data dependencies.

The bypasses in the memory MEM are used to hide the memory latency via a *load-store buffer*, as shown in Fig. 16. Such structure can be substituted by a more efficient implementation: an *associative memory*. The algorithm automatically detects the need for a load-store buffer and its optimal size.

Figure 15 shows the effective cycle time and area of the best design point found on different partitions of F, forming a Pareto-point curve. As the depth of F increases, more bypasses are needed on the register file in order to completely pipeline F. The area of the design increases with more bypasses. The best effective cycle time is achieved with F divided into 6 stages, 8 bypasses applied to RF and 9 to MEM. Design points (4,5) and (3,4) (circled in the figure) for 4 and 3 stages are simpler and overall might deliver a better design compromise. This figure illustrates how more parallelism must be introduced in order to achieve better performance, and how increasing parallelism has a significant area overhead. Furthermore, there is some point where the performance cannot be improved by simply increasing the instruction parallelism in the design.

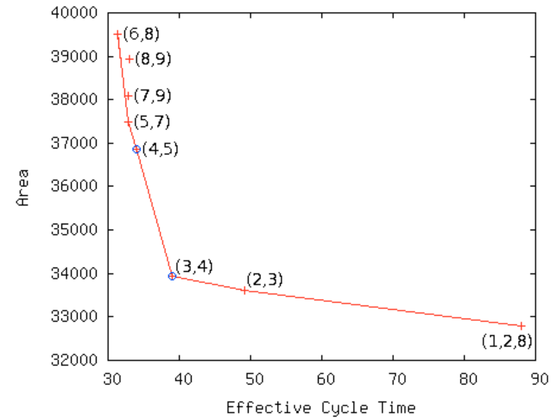

Fig. 15. Effective cycle time and area of the best pipelined design for different depths of F. (x,y) and (x,y,z) tuples represent the depth of F, the number of bypasses applied to RF and to MEM ($z = 9$ if omitted)

V. CONCLUSIONS

Elasticity enables new opportunities for system optimization that target the average case performance. By using early evaluation, the designer can focus on optimizing frequently used and hence critical parts of the design. Those blocks that are not critical for the overall performance, can have a relaxed timing with some extra latency cost. The obtained performance advantage can also be used for saving power.

This paper has presented a framework for exploration of microarchitectural designs. This framework uses correct-by-construction transformations many of which can only be applied to elastic systems. In particular the optimal pipelining of a system can be automatically obtained by using these transformations.

Acknowledgments. This work has been supported by grants from Intel Corp., CICYT TIN2004-07925 and FI from Generalitat de Catalunya. We want to thank Alexander Gotmanov and Timothy Kam for applying ideas presented in this paper in design experiments and Dmitry Bufistov for developing methods for optimization elastic systems.

REFERENCES

[1] M. Ampalam and M. Singh, "Counterflow pipelining: Architectural support for preemption in asynchronous systems using anti-tokens," in *Proc. International Conf. Computer-Aided Design (ICCAD)*, 2006, pp. 611–618.

[2] P. Beerel, A. Lines, M. Davies, and N. Kim, "Slack matching asynchronous designs," in *Proc. Int. Symposium on Asynchronous Circuits and Systems*, 2006, p. 11.

[3] L. Benini, G. D. Micheli, A. Lioy, E. Macii, G. Odasso, and M. Poncino, "Automatic synthesis of large telescopic units based on near-minimum timed supersetting," *IEEE Transactions on Computers*, vol. 48, no. 8, pp. 769–779, 1999.

[4] E. Bloch, "The engineering design of the stretch computer," in *Proc. IRE/AIEE/ACM Eastern Joint Computer Conference*, Dec. 1959, pp. 48–58.

[5] C. Brej and J. Garside, "Early output logic using anti-tokens," in *Proc. International Workshop on Logic Synthesis*, May 2003, pp. 302–309.

[6] D. Bufistov *et al.*, "Retiming and recycling for elastic systems with early evaluation," in *Proc. ACM/IEEE Design Automation Conference*, Jul. 2009, pp. 288–291.

[7] D. Bufistov, J. Júlvez, and J. Cortadella, "Performance optimization of elastic systems using buffer resizing and buffer insertion," in *Proc. International Conf. Computer-Aided Design (ICCAD)*, Nov. 2008, pp. 442–448.

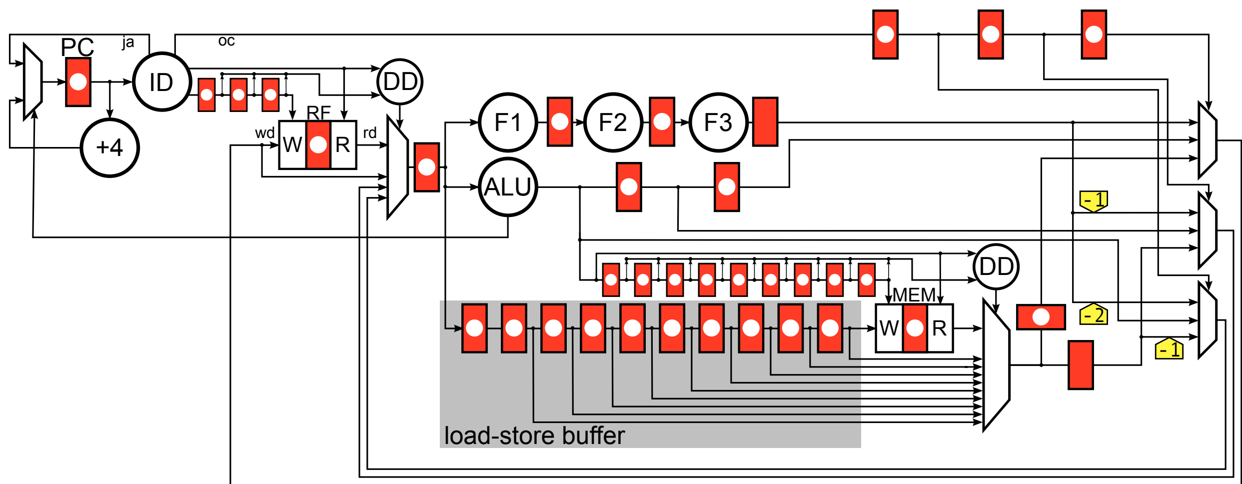

Fig. 16. Pipelined DLX graph (F divided into 3 blocks. RF has 3 bypasses and M 9)

[8] D. Bufistov, J. Cortadella, M. Kishinevsky, and S. Sapatnekar, "A general model for performance optimization of sequential systems," in *Proc. Int. Conf. Computer-Aided Design*, 2007, pp. 362–369.

[9] L. Carloni, K. McMillan, and A. Sangiovanni-Vincentelli, "Theory of latency-insensitive design," *IEEE Trans. on Computer-Aided Design*, vol. 20, no. 9, pp. 1059–1076, Sep. 2001.

[10] L. Carloni and A. Sangiovanni-Vincentelli, "Coping with latency in SoC design," *IEEE Micro, Special Issue on Systems on Chip*, vol. 22, no. 5, p. 12, October 2002.

[11] ——, "Combining retiming and recycling to optimize the performance of synchronous circuits," in *16th Symp. on Integrated Circuits and System Design (SBCCI)*, Sep. 2003, pp. 47–52.

[12] L. Carloni, K. McMillan, A. Saldanha, and A. Sangiovanni-Vincentelli, "A methodology for correct-by-construction latency insensitive design," in *Proc. Int. Conf. Computer-Aided Design*. IEEE Press, Nov. 1999, pp. 309–315.

[13] L. Carloni and A. Sangiovanni-Vincentelli, "Performance analysis and optimization of latency insensitive systems," in *Proceedings of the 37th Annual Design Automation Conference*. ACM, 2000, p. 367.

[14] M. Casu and L. Macchiarulo, "Adaptive Latency-Insensitive Protocols," *IEEE Design & Test of Computers*, vol. 24, no. 5, pp. 442–452, 2007.

[15] M. Casu, "Improving synchronous elastic circuits: Token cages and half-buffer retiming," in *Proc. Int. Symposium on Asynchronous Circuits and Systems*. IEEE Computer Society, 2010, pp. 128–137.

[16] T. Chelcea and S. Nowick, "Robust interfaces for mixed-timing systems," *IEEE Trans. VLSI Syst.*, vol. 12, no. 8, pp. 857–873, 2004.

[17] J. Cortadella and M. Kishinevsky, "Synchronous elastic circuits with early evaluation and token counterflow," in *Proc. ACM/IEEE Design Automation Conference*, Jun. 2007, pp. 416–419.

[18] J. Cortadella, M. Kishinevsky, and B. Grundmann, "SELF: Specification and design of a synchronous elastic architecture for DSM systems," in *TAU-2006: International Workshop on Timing Issues in the Specification and Synthesis of Digital Systems*, 2006. [Online]. Available: www.lsi.upc.edu/˜jordicf/gavina/BIB/files/self_tr.pdf

[19] ——, "Synthesis of synchronous elastic architectures," in *Proc. ACM/IEEE Design Automation Conference*, Jul. 2006, pp. 657–662.

[20] M. Galceran-Oms, J. Cortadella, and M. Kishinevsky, "Speculation in elastic systems," in *Proc. ACM/IEEE Design Automation Conference*, Jul. 2009, pp. 292–295.

[21] M. Galceran-Oms, J. Cortadella, M. Kishinevsky, and D. Bufistov, "Automatic microarchitectural pipelining," in *Proc. Design, Automation and Test in Europe (DATE)*, Apr. 2010.

[22] S. Hassoun and C. Ebeling, "Architectural retiming: Pipelining latency-constrained circuits," in *Proc. ACM/IEEE Design Automation Conference*, Jun. 1996, pp. 708–713.

[23] J. Hennessy and D. Patterson, *Computer Architecture: a Quantitative Approach*. Morgan Kaufmann Publisher Inc., 1990.

[24] H. M. Jacobson, P. N. Kudva, P. Bose, P. W. Cook, S. E. Schuster, E. G. Mercer, and C. J. Myers, "Synchronous interlocked pipelines," in *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, Apr. 2002, pp. 3–12.

[25] J. Júlvez, J. Cortadella, and M. Kishinevsky, "On the performance evaluation of multi-guarded marked graphs with single-server semantics," *Discrete Event Dynamic Systems*, Aug. 2009.

[26] T. Kam, M. Kishinevsky, J. Cortadella, and M. Galceran-Oms, "Correct-by-construction microarchitectural pipelining," in *Proc. Int. Conf. Computer-Aided Design*, 2008, pp. 434–441.

[27] S. Krstić, J. Cortadella, M. Kishinevsky, and J. O'Leary, "Synchronous elastic networks," in *Int. Conf. on Formal Methods in Computer-Aided Design (FMCAD)*, Nov. 2006.

[28] C. E. Leiserson and J. B. Saxe, "Retiming synchronous circuitry," *Algorithmica*, vol. 6, no. 1, pp. 5–35, 1991.

[29] C.-H. Li and L. Carloni, "Using functional independence conditions to optimize the performance of latency-insensitive systems," in *Proc. Int. Conf. Computer-Aided Design*, Nov. 2007.

[30] R. Lu and C.-K. Koh, "Performance optimization of latency insensitive systems through buffer queue sizing of communication channels," in *Proc. Int. Conf. Computer-Aided Design*, Nov. 2003, pp. 227–231.

[31] R. Manohar and A. Martin, "Slack elasticity in concurrent computing," in *Mathematics of Program Construction*. Springer, 1998, pp. 272–285.

[32] J. Monteiro, S. Devadas, and A. Ghosh, "Retiming sequential circuits for low power," in *Proc. Int. Conf. Computer-Aided Design*, 1993, pp. 398–402.

[33] T. Murata, "Petri Nets: Properties, analysis and applications," *Proceedings of the IEEE*, pp. 541–580, Apr. 1989.

[34] A. Peeters and K. van Berkel, "Synchronous handshake circuits," in *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, Mar. 2001, pp. 86–95.

[35] P. Prakash and A. Martin, "Slack matching quasi delay-insensitive circuits," in *Proc. Int. Symposium on Asynchronous Circuits and Systems*, 2006, p. 10.

[36] R. Reese, M. Thornton, C. Traver, and D. Hemmendinger, "Early evaluation for performance enhancement in phased logic," *IEEE Trans. on Computer-Aided Design*, vol. 24, no. 4, pp. 532–550, Apr. 2005.

[37] C. Soviani, O. Tardieu, and S. Edwards, "Optimizing sequential cycles through Shannon decomposition and retiming," in *Proceedings of the conference on Design, automation and test in Europe.* European Design and Automation Association 3001 Leuven, Belgium, Belgium, 2006, pp. 1085–1090.

[38] S. Srinivasan, K. Sarker, and R. Katti, "Token-Aware Completion Functions for Elastic Processor Verification," *Research Letters in Electronics*, 2009.

[39] Y.-S. Su, D.-C. Wang, S.-C. Chang, and M. Marek-Sadowska, "An efficient mechanism for performance optimization of variable-latency designs," in *Proc. ACM/IEEE Design Automation Conf.*, 2007, pp. 976–981.

[40] I. E. Sutherland, "Micropipelines," *Communications of the ACM*, vol. 32, no. 6, pp. 720–738, Jun. 1989.

[41] M. Vijayaraghavan and Arvind, "Bounded dataflow networks and latency-insensitive circuits," in *Proceedings of the 7th International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, July 2009.

[42] M. Vijayaraghavan and A. Arvind, "Bounded dataflow networks and latency-insensitive circuits," in *MEMOCODE'09: Proceedings of the 7th IEEE/ACM international conference on Formal Methods and Models for Codesign*, 2009, pp. 171–180.

[43] T. Williams, "Performance of iterative computation in self-timed rings," *The Journal of VLSI Signal Processing*, vol. 7, no. 1, pp. 17–31, 1994.