SELF: Specification and design of synchronous elastic circuits

Jordi Cortadella
Universitat Politècnica de Catalunya
Barcelona, Spain

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

Bill Grundmann
Strategic CAD Lab, Intel Corp.
Hillsboro, OR, USA

***Abstract*— A simple protocol for latency-insensitive design is presented. The main features of the protocol are the efficient implementation of elastic communication channels and the automatable design methodology. A latch-based implementation with no storage overhead is also proposed. With this approach, fine-granularity elasticity can be introduced at the level of functional units (e.g. ALUs, memories). A formal specification of the protocol is defined and several schemes for the implementation of elasticity are discussed. The opportunities that this protocol opens for microarchitectural design are illustrated with several examples.**

I. Motivation

The time discretization imposed by synchronicity forces to take early decisions that often complicate changes at the latest stages of the design or efficient migrations to scaled technologies. In DSM technologies, calculating the number of cycles required to transmit an event from a sender to a receiver is a problem that cannot be solved until the final layout has been generated.

Some researchers advocate for the modularity and efficiency of asynchronous circuits to devise some kind of object-oriented methodology for complex systems. However, the CAD support for asynchronous circuits is still in its pre-history.

The question we want to answer in this paper is: can we find an *efficient* scheme that combines the modularity of asynchronous systems with the simplicity of synchronous implementations?.

Other authors have been working into this direction. Latency-insensitive (LI) schemes [CMSV01] were proposed to separate communication from computation and make the systems insensitive to the latencies of the computational units and channels. The implementation of LI systems is synchronous [CSV02], [CN01] and uses *relay stations* at the interfaces between computational units.

In a different scenario, *synchronous interlocked pipelines* [JKB$^+$02] were proposed to achieve fine grained local handshaking at the level of stages. The implementation is conceptually similar to a discretized version of traditional asynchronous pipelines with req/ack handshake signals.

A *de-synchronization* [HDGC04], [BCK$^+$04] approach automatically transforms synchronous specifications into asynchronous implementations by replacing the clock network with an asynchronous controller. The success of this paradigm will mainly depend on the attitude of designers towards accepting asynchrony in their design flow.

A. Contributions of the paper

The main contributions of the paper are as follows:

- A simple and efficient protocol for latency-insensitive design and an abstract model for elastic channels and buffers.
- Demonstration of several architectures and control schemes for the implementation of elastic buffers and channels. Implementations of LI systems proposed in [CSV02] and interlock pipelines in [JKB$^+$02] are two particular solutions in this design space.
- An efficient latch-based implementation with no storage overhead, clock-gating of all sequential elements and eager forks.
- We demonstrate that the proposed scheme can be applied on different levels of system granularity and both in the white-box (e.g. microprocessor design) and black-box scenarios (SoC IPs). Contrary to [CMSV01] the elastic system before inserting additional delays has the same sequential latency as the original synchronous design.
- A design methodology with the automatic correct-by-construction transformation of a synchronous system into an elastic one and the analytical performance analysis.
- Sequential optimization of the controllers.

II. The structure of an elastic system

Intuitively, an elastic design is a collection of elastic modules and elastic channels. Every channel can propagate data from one module to another. As it will be discussed in Section III, channels have control wires implementing a handshake between the sender and the receiver.

For simplicity in the explanation, we will initially assume that elastic modules are partitioned into combinational blocks, to do computations, and sequential elements, to store and propagate the results of the computations. In section VII we will show the generalization to modules with fixed and variable sequential latencies.

In the low granularity elastic design, all flip-flops are replaced with *Elastic Buffers* (EB). EBs can be composed of *Elastic Half-Buffers*, (EHB), in the same fashion as flip-flops can be implemented as a pair of two transparent latches with opposite polarity (master and slave).Thus, a designer of an elastic system has a choice between using edge-triggered or transparent elements.

Depending on the state of the associated control wires, a channel can carry valid or invalid data items. For simplicity, we will talk about *tokens* and *bubbles*, respectively.

III. Specification of the SELF protocol

This section describes an elastic protocol called SELF (*Synchronous ELastic Flow*) that can be implemented with the following features:

- Small control overhead that can be effectively used at the level of medium-grain blocks (ALUs, shifters, register files, etc).
- Scalable in such a way that the delay overhead of the protocol is independent from the size of the system.
- A method for design automation to transform conventional synchronous designs into elastic systems.

Fig. 1 depicts an example of an elastic implementation for transmitting data between two units. Each register has an associated *valid* bit (V) that keeps track of the validity of the stored data. The clock signal is not explicitly shown and the enable signal (*En*) indicates when new data is stored into the register. The chain of AND gates manages the *back-pressure* generated by the receiver when it is not able to accept data (*Stop* $= 1$).

The scheme in Fig 1 is not scalable due to the long combinational path from the receiver to the sender. When the pipeline is full, i.e. all V's are at 1, the delay of the *Stop* chain becomes critical.

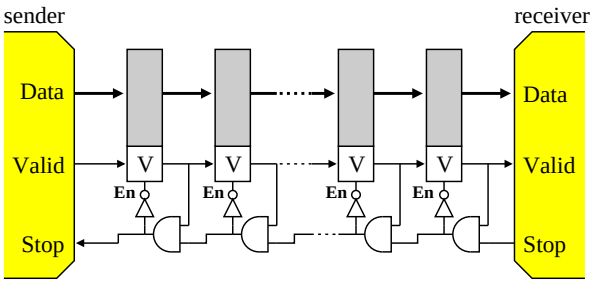

Fig. 1. A simple control for elastic data transmission.

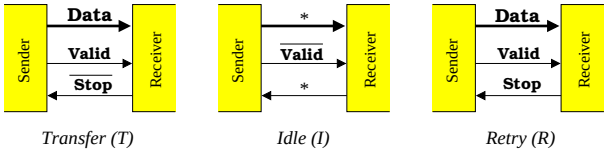

Fig. 2. The SELF protocol.

A. The SELF protocol

Data transfer between a sender and a receiver is performed by using the two control signals, *Valid* (V) and *Stop* (S), that determine three possible states in the channel (see Fig. 2):

- *Transfer (T)*, when $V = 1$ and $S = 0$, indicating that the sender is providing valid data and the receiver is accepting it.
- *Idle (I)*, when $V = 0$, indicating that the sender is not providing any valid data.
- *Retry (R)*, when $V = 1$ and $S = 1$, indicating that the sender is providing data but the receiver is not able to 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.

The language observed at a SELF channel can be described by the following regular expression:

$$(I^* R^* T)^* \tag{1}$$

Absence of a subtrace RI implies the persistency of the behavior. Table I shows an example of a trace committing the SELF protocol and transmitting values $A - D$. When $V = 0$, the value at the data bus is irrelevant (cycles 0, 6 and 7). The receiver can issue a *Stop* even when the sender does not send valid data (cycle 7). The sender persistently maintains the same valid data as in the previous cycle during cycle 3, 4 and 9.

B. Specification of elastic buffers

Figure 3 shows the interface of an elastic buffer (EB) with one input and one output channels. The extension to multi-input/output channels will be discussed in Section V-A. There can be different architectures to implement an EB trading-off area, delay and power

Cycle	0	1	2	3	4	5	6	7	8	9
Data	$*$	A	B	B	B	C	$*$	$*$	D	D
Valid	0	1	1	1	1	1	0	0	1	1
Stop	0	0	1	1	0	0	0	1	1	0
State	I	T	R	R	T	T	I	I	R	T

TABLE I
A TRACE COMMITTING THE SELF PROTOCOL.

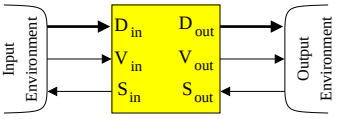

Fig. 3. Interface of an EB with one input and one output channels.

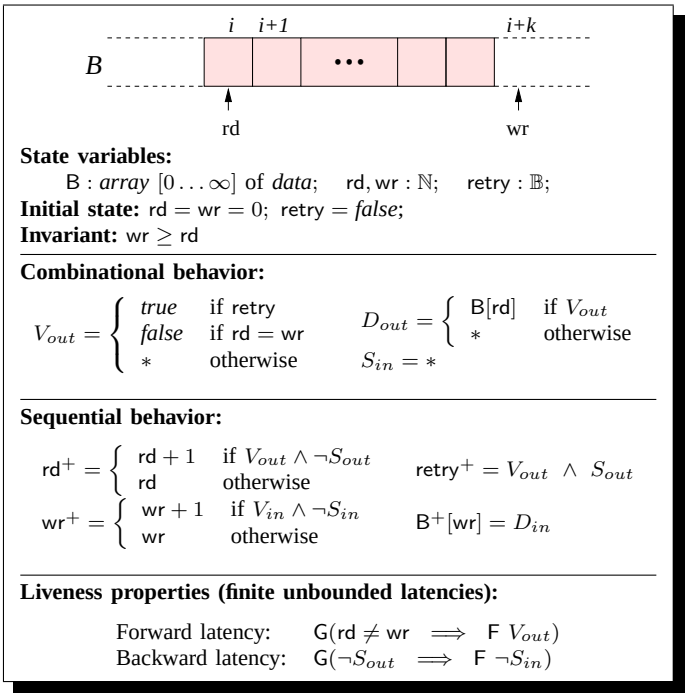

Fig. 4. Abstract model for an EB.

consumption. For this reason, it is convenient to have a formal specification that can be used to verify that a particular implementation is valid refinement of the specification and can safely substitute another implementation.

The abstract model for an EB is described in Fig. 4. Briefly, an EB is modeled as an unbounded FIFO that commits the SELF protocol at the input and output channels. The notation X^+ is used to represent the *next-state value* of variable X. The symbol '$*$' represents a non-deterministic value (don't care).

B is an infinite array that stores the items written into the buffer, but not sent to the output yet. The variables wr and rd are the write and read indices of the array, respectively. The value $k = \mathsf{wr} - \mathsf{rd}$ is the current number of items stored in the buffer.

The retry variable remembers whether a transfer was attempted in the previous cycle. If retry is true, the same data item as in the previous cycle is issued and $V_{out} = \mathit{true}$. On the other hand, if the buffer does not contain any item ($\mathsf{rd} = \mathsf{wr}$), no transfer can be performed ($V_{out} = \mathit{false}$). Finally, the value $V_{out} = *$ represents the non-deterministic behavior of a buffer with finite but unbounded delay: the items stored in the buffer will eventually be transferred to the output after a finite unknown delay. S_{in} can non-deterministically stop any data transfer at the input channel. The behavior of the indices of the array is modeled by the rd^+ and wr^+ equations. When a data transfer is produced at the corresponding channel ($V \wedge \neg S$), the value of the index is incremented.

Two liveness properties expressed in *linear temporal logic* [Pnu77] ensure finite response in the forward and backward directions:

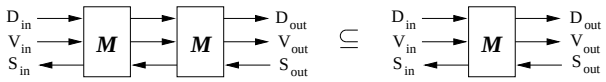

Fig. 5. The sequential composition of EBs refines an EB.

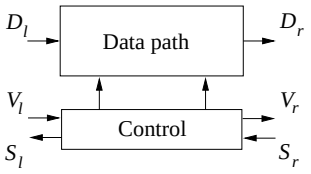

Fig. 6. EB structure.

- data from the EB will eventually be sent to the output, and
- a non-stop at the output will eventually be propagated to the input.

C. Verification of EBs

The model in Fig. 4 has been used as specification to check that every implementation presented in this paper is a refinement of this model. The proofs have been performed by using the features of refinement verification and data type reductions offered by SMV [McM99].

Let M be an abstract model of the EB. We have also verified that the sequential composition of two M is a refinement of M (see Fig. 5). This guarantees that EBs with specific depths can be built by the sequential composition of any implementations of EB that refines M presented in this paper.

IV. Implementation of elastic buffers

The implementation of an EB can be decomposed into two parts: data-path and control (see Fig. 6).

A. The data-path of the EBs

Two important parameters of an EB are the *forward*, L_f, and backward, L_b, latencies. The L_f is the latency of forward propagation of the data and the Valid bit in in case the receiver is ready. The L_b is the backward latency for the stop signals. According to the unbounded specification of Fig. 4, L_f and L_b can be any value greater than zero (including non-deterministic values)[1]. However, for performance optimization and for the ability to distribute the EBs across long communication channels and reuse them as sequential wire repeaters, the following constraint should be satisfied:

$$L_f = L_b = 1$$

The capacity of an EB defines the maximal number of data items that can be simultaneously stored inside the buffer, e.g. the capacity of a regular flip-flop is 1. The following property holds:

Property 4.1: The capacity of an elastic buffer, C, should satisfy the following constraint:

$$C \geq L_f + L_b$$

Therefore, for the case of interest ($L_f = L_b = 1$) the minimal possible capacity is $C = 2$. Hence the data path of an EB can be constructed with two storage cells and different write/read policies, e.g. the different number of read and write ports. Fig. 7 illustrates different options for the construction of an EB with respect to the numbers of write and read ports. For example, in the case of W_1R_1, writes always occur to the first cell (A), and reads always occur from the second cell (B), while in the case of W_2R_2 both cells permit writes and reads.

[1] $L_f = 0$ and $L_b = 0$ would reduce the EB to an elastic channel. Having one of the latencies equal to 0 is possible in parts of the design, but does not scale, due to the long combinational delays and combinational cycles.

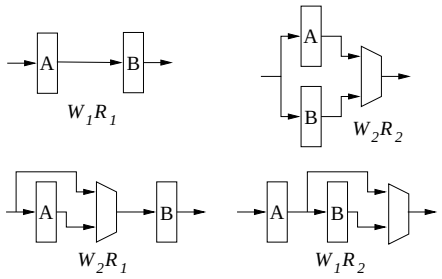

Fig. 7. Architectures for the data-path of EBs.

The following property holds:

Property 4.2: A W_1R_1 EB with $L_f = L_b = 1$ cannot be implemented using single-edge flip-flops controlled by the same frequency clock as used by the environment.

Intuitively, this is because such a structure would have a latency of two between the write operation through the input channel and the read operation from the output channel. However, the construction of W_1R_1 is possible with *double-pumping*, which would ensure that two data moves can occur during a clock cycle of the environment. Several implementation strategies can be considered: (1) two flops acting on different edges of the clock, (2) two flops acting on a single edge of a double frequency clock, and most notably, (3) simply by using two transparent latches of different polarity, similar to a master-slave structure, but with the *independent control* of enable signals for two latches.

Obviously, the W_1R_1 structure based on transparent latches is preferable according to most design metrics (area, delay, power). Other structures, that are based on single edge flops, can be used for high-level and performance modeling, formal verification (which often do not support transparent latches) and further be replaced with the equivalent W_1R_1 structures in the later design stages. Flop-based structures can also be used in the design methodologies, which would not allow transparent latches (e.g., FPGAs).

B. The control of the EBs

As an example, Fig. 8 depicts the FSM specifications for two of the EBs shown in Fig. 7. The transparent latches are shown with single boxes, labelled with the phase of the clock, L (active low) or H (active high). The flip-flops are drawn as two transparent latches back-to-back. To simplify the drawing the clock lines are not shown. The signals going from the control units to the flip-flops and latches are the enable signals. Finally, 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 specifications are similar to the specification of a 2-slot FIFO. The two versions of the FSMs are identical with respect to the protocol of the control channels, but perform different data exchanges in the data path. Let us consider, as an example, the W_1R_1 buffer. In the *Empty* state no valid data is captured in the data-path. In the *Half-full* state, an output slave latch keeps valid data. In the *Full* state, both latches keep valid data and the EB requests the sender to stop.

Figure 10 shows a linear elastic system with the two types of EBs from Fig. 8. Different types of EBs and controllers can be freely

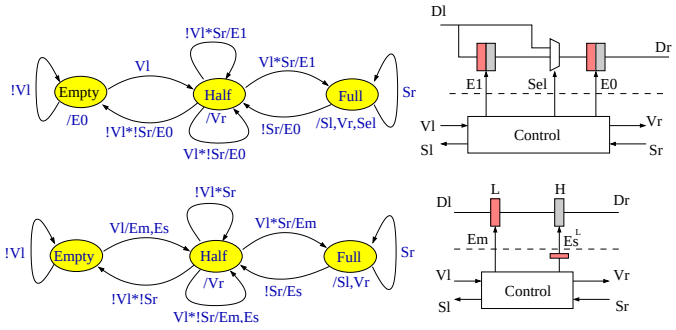

Fig. 8. Control specification for the W_2R_1 and W_1R_1 EBs

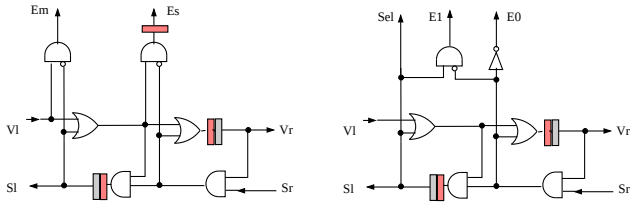

Fig. 9. W_1R_1 (left) and W_2R_1 (right) control implementation

mixed since their channel interfaces and protocols are identical and commit the SELF protocol. The EB on the left uses the W_2R_1 data-path. An implementation for the W_1R_1 control shown in Fig. 9 (left) can be obtained directly from the FSM specification in Fig. 8 (bottom). By further splitting the two flip-flops into transparent latches and retiming, one can obtain a controller composed of two VS cells with latches of the opposite phase. The first of these VS cells is shown in the middle of Fig. 10. For the second cell, a modified equivalent implementation, VS', is shown with the clock gating of both control latches.

V. Advanced Elastic Control Structures

A. 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 11(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 to compose multiple Joins together in a tree like structure.

Figure 11(b) depicts a *lazy fork*. The controller waits for both receivers to be ready (*stop* = 0) before sending the data[2]. A more efficient structure shown in Fig. 11(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" which output channels already received the data. This structure offers performance advantages when the two output channels have different backpressure.

[2]This implementation is identical to the one presented in [JKB+02].

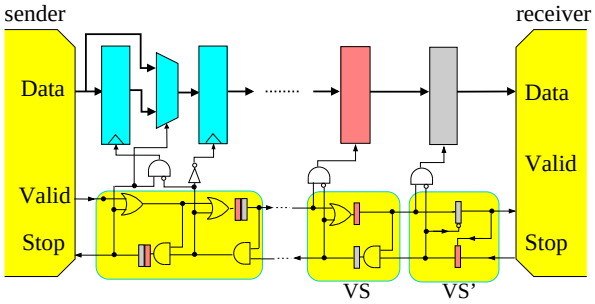

Fig. 10. Linear elastic pipeline with three versions of controllers

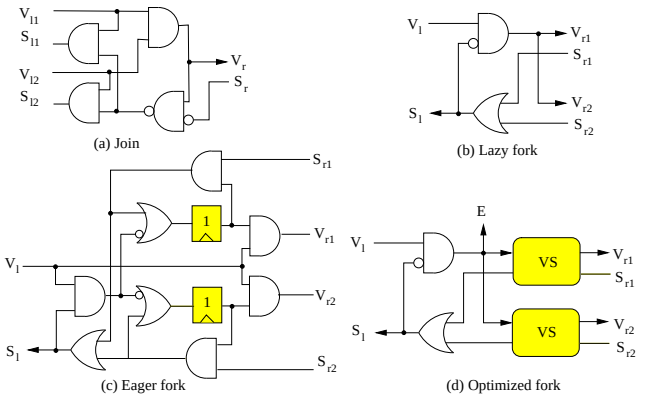

Fig. 11. Controllers for elastic Join and Fork structures.

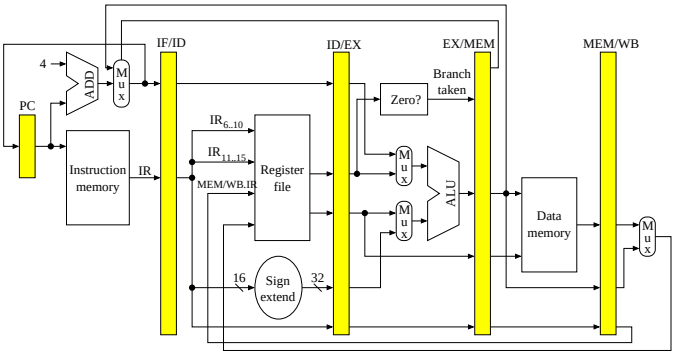

Fig. 12. The DLX pipeline.

B. Optimization of elastic controllers

Many forms of optimizing the elastic control structures are possible and some have been implemented by the authors. Local transformation of controllers have been illustrated in Fig. 10.

Combining adjacent controllers opens up other possibilities for optimization. For example, an EHB followed by a fork is equivalent to another eager implementation of the fork shown in Fig. 11(d). Furthermore, constraints on the environment or the controller topology may enable the removal of redundant latches and gates. As an example, if the stop signal is never emitted by the receiver, the latches and combinational gates in the backward path of Fig. 10 can be removed and enable and forward logic can be simplified accordingly. We have developed a tool for automatic removal of the redundant transparent latches and flip-flops and logic gates from the elastic control layer using the SIS logic synthesis system [SSL+92].

VI. A Design Example: The DLX Pipeline

We use the DLX pipeline [HP90] to illustrate how elasticity can be used in microarchitectural design and to briefly review the design flow. Figure 12 depicts a block diagram of a possible implementation. It is assumed, for simplicity, that delay slots are used for handling data hazards and no forwarding is used. The shadowed boxes represent the registers that store the state of the microprocessor. From the control point of view, the register file and the instruction and data memories can be considered as combinational units.

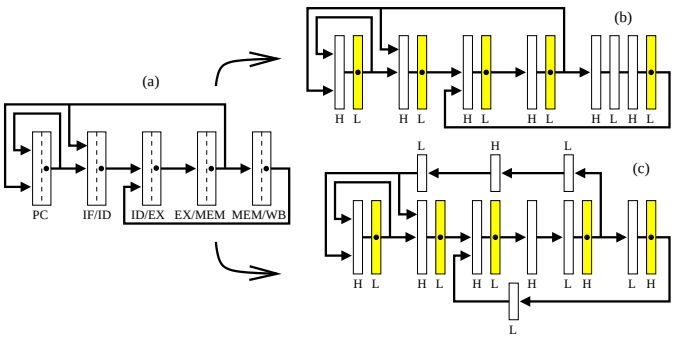

Fig. 13. The control layer for the DLX pipeline.

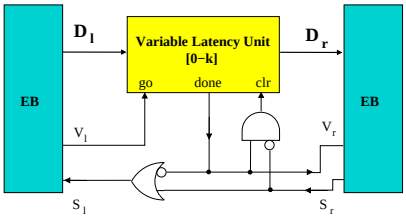

Fig. 14. Handling variable latency.

Figure 13(a) shows an abstraction of the pipeline in which only the channels among units are shown. The boxes represent the registers composed of two latches (master/slave). The diagram is equivalent for the datapath and control layers. The join and fork SELF controllers must be used when one unit has more than one input or output channel, respectively.

Figures 13(b) and 13(c) show a latch-based diagram of the same pipeline after the insertion of "bubbles". The shadowed latches are the ones initialized with valid data. This example illustrates that the insertion of bubbles does not affect the functionality of the system, although it may affect performance. Thus, the elastic architecture is *correct-by-construction* with respect to inserting empty EBs. This feature is even more interesting when the bubbles can be inserted dynamically, as it will be discussed in the next section.

The criteria for bubble insertion can be different: break long wires, cycle time optimization, power reduction, etc. Some of these criteria have been studied in [CSV03], [LK03] and the discussion on how to apply them is out of the scope of this paper.

VII. Handling variable latency

It is possible to insert bubbles dynamically by temporarily injecting $valid = 0$ and $stop = 1$ within any channel. Under the control of some supervisor algorithm the dynamic bubble insertion can lead to temporal and gradual shut-down/wake-up of computation and can be used for trading power vs. performance at different levels of granularity.

Since the bubble insertion preserves correctness of the behavior it is also possible to convert some units of the system from fixed to variable latency. For instance, one could replace the 1-cycle ALU from the DLX pipeline by a variable-latency ALU optimized for the typical data case (e.g. short carry propagation). Such a telescopic ALU (cf. [BML+99]) calculating with latency 1 for the typical data mix, and with latency 2 for the rare data mix, can lead to performance improvement by designing the whole pipeline for a faster clock cycle, and to area reduction by reducing the number of logic gates per pipeline stage.

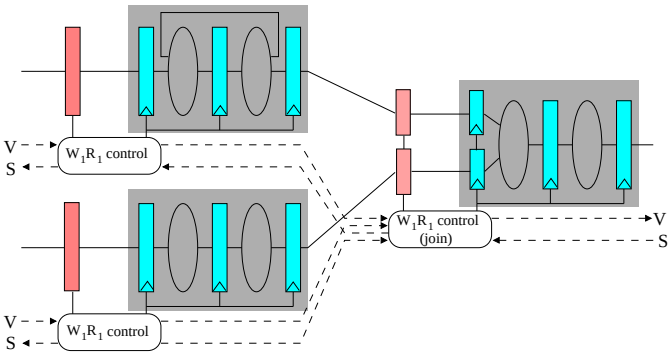

Fig. 15. Coarse-level control of functional blocks.

An exact construction of the variable latency controller depends on the unit type (e.g. a multi-cycle combinational unit vs. a pipelined unit) as well as on the exact protocol that the unit follows. An example is shown in Fig. 14. The unit is assumed to have variable latency within 0 to k cycles. The valid signal sent at the input channel, V_l, is connected to the *go* input of the unit that starts the operation. The valid output on the right channel, V_r, is asserted when the unit completes the operation (*done*). Until then, the input channel is stopped, which guarantees that the valid data value on the input channel is not changing (cf. the construction of the VS block). When the output EB is enabled the unit receives a clear signal (*clr*). This acknowledgement can be used to reset the internal state of the unit and to prepare it for the next operation.

VIII. Formal methods and CAD tool support

A. Re-using existing CAD flows

A crucial aspect in the adoption of elasticity by designers is the capability of using existing CAD flows. As shown in Sect. VI, the transformation of a synchronous system into an elastic system is straightforward by simply adding the control layer committing the protocol. The datapath remains intact, except for the fact that flip-flops are replaced with slightly different flip-flop cells with independent enabling for the master and slave latches (assuming the W_1R_1 EBs are selected). Since the resulting design is still synchronous, conventional CAD tools for synthesis, analysis and simulation can be used.

B. Coarse-level elasticity

Figure 15 depicts an example on how elasticity can be inserted between blocks that can contain any kind of sequential behavior. In the example, the global clock is substituted by the gated clock generated by the elastic controllers (W_1R_1 in this particular case). Thus, the elastic controller commands all the registers of the block with only one enable signal.

To handle the backpressure, "ghost" latches must be added at the inputs of the blocks. These latches have the same polarity as the internal input latches of the block and do not introduce any extra latency, since they are redundant during the normal operation of the system. In the case that the blocks can be "opened" and optimized, the internal input latches can be simply removed.

C. Correctness by construction

In Sect. III, a formal model for the protocol and the EBs has been presented. This model is an abstraction that can be used for compositional verification of complex systems to guarantee correct system behavior. In particular, Fig. 13 illustrated the insertion of

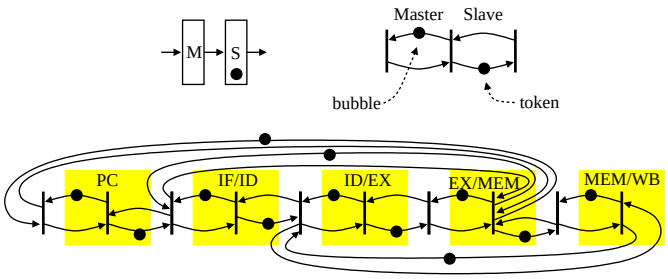

Fig. 16. A concurrent model for elastic designs.

empty EBs into the DLX pipeline. The following property that guarantees soundness of this transformation holds:

Property 8.1: The insertion of empty EBs in an elastic design preserves *flow equivalence*.

Informally, *flow equivalence* [GTL03] between two designs guarantees that for every output the order of *valid* data items is the same[3]. An example on what flow equivalence means is illustrated below.

Synchronous behavior:	$a_1\ a_2\ a_3\ a_4\ \cdots\ a_i\ \cdots$ $b_1\ b_2\ b_3\ b_4\ \cdots\ b_i\ \cdots$
Elastic behavior:	$a_1\ \tau\ a_2\ \tau\ \tau\ \tau\ a_3\ \tau\ a_4\ \cdots\ a_i\ \cdots$ $b_1\ \tau\ \tau\ b_2\ \tau\ b_3\ b_4\ \cdots\ b_i\ \cdots$

The synchronous behavior shows the trace of values observed at two registers, a and b, at every cycle. After making the design elastic, some don't' care values marked as invalid may appear (denoted as τ). However, the order of valid data is preserved. It is important that the values at different registers may be shifted in time with respect to the pure synchronous behavior without affecting the functional correctness of the system, since any observer of both values would synchronize the corresponding valid tokens.

While bubble insertion (empty EBs) is correct-by-construction, token insertion (non-empty EBs) requires care and possibly partial re-design similar to the standard pipelining.

D. Performance evaluation

Performance evaluation is another important aspect in system design. Abstractions of elastic systems can be derived by using concurrent models. In particular, the behavior of elastic systems can be modeled by using a subclass of Petri nets called *marked graphs* [Mur89]. As an example, Fig. 16 depicts the marked graph model corresponding to the DLX abstraction shown in Fig. 13(a). Each latch is modeled as a pair of complementary arcs in which the location of the token indicates whether the latch contains valid (token) or invalid (bubble) data. The transitions of the marked graph represent the computations between latches (empty if the transition is between a master and slave latch). Modeling systems with marked graphs enables the use of an extensive set of tools for the analytical performance analysis that can be effectively used at the earliest stages of the design. The authors have implemented computation of the effective cycle time for the SELF systems based on the separation analysis method from [Cha98].

IX. Conclusions

A novel scheme for latency-insensitive design has been presented. It combines the modularity of asynchronous design with the efficiency of synchronous implementations.

[3]Other authors call this property *latency equivalence* [SBM+05].

The little overhead introduced by the implementation of the elastic buffers makes this scheme attractive for different levels of granularity. Additionally, the correct-by-construction paradigm of this method enables its applicability at the latest stages of the design, when accurate delay estimations of data transfers have been performed, without any impact on the functionality of the system.

References

[BCK+04] I. Blunno, J. Cortadella, A. Kondratyev, L. Lavagno, K. Lwin, and C. Sotiriou. Handshake protocols for de-synchronization. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 149–158. IEEE Computer Society Press, April 2004.

[BML+99] L. Benini, G. De 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*, 48(8):769–779, 1999.

[Cha98] Supratik Chakraborty. *Polynomial-Time Techniques for Approximate Timing Analysis of Asynchronous Systems*. PhD thesis, Stanford University, August 1998.

[CMSV01] L. Carloni, K.L. McMillan, and A.L. Sangiovanni-Vincentelli. Theory of latency-insensitive design. *IEEE Transactions on Computer-Aided Design*, 20(9):1059–1076, September 2001.

[CN01] Tiberiu Chelcea and Steven M. Nowick. Robust interfaces for mixed-timing systems with application to latency-insensitive protocols. In *Proc. ACM/IEEE Design Automation Conference*, June 2001.

[CSV02] L.P. Carloni and A.L. Sangiovanni-Vincentelli. Coping with latency in SoC design. *IEEE Micro, Special Issue on Systems on Chip*, 22(5):12, October 2002.

[CSV03] L. Carloni and A.L. Sangiovanni-Vincentelli. Combining retiming and recycling to optimize the performance of synchronous circuits. In *16th Symp. on Integrated Circuits and System Design (SBCCI)*, pages 47–52, September 2003.

[GTL03] P. Le Guernic, J.-P. Talpin, and J.-Ch. Le Lann. Polychrony for system design. *Journal of Circuits, Systems and Computers*, 12(3):261–304, April 2003.

[HDGC04] G.T. Hazari, M.P. Desai, A. Gupta, and S. Chakraborty. A novel technique towards eliminating the global clock in VLSI circuits. In *Int. Conf. on VLSI Design*, pages 565–570, January 2004.

[HP90] J.L. Hennessy and D. Patterson. *Computer Architecture: a Quantitative Approach*. Morgan Kaufmann Publisher Inc., 1990.

[JKB+02] Hans M. Jacobson, Prabhakar N. Kudva, Pradip Bose, Peter W. Cook, Stanley E. Schuster, Eric G. Mercer, and Chris J. Myers. Synchronous interlocked pipelines. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 3–12, April 2002.

[LK03] R. Lu and C.-K. Koh. Performance optimization of latency insensitive systems through buffer queue sizing of communication channels. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 227–231, November 2003.

[McM99] Kenneth L. McMillan. Verification of infinite state systems by compositional model checking. In *CHARME*, pages 219–234, 1999.

[Mur89] T. Murata. Petri Nets: Properties, analysis and applications. *Proceedings of the IEEE*, pages 541–580, April 1989.

[Pnu77] A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th IEEE Symposium on Foundations of Computer Science*, pages 46–57, 1977.

[SBM+05] S. Suhaib, D. Berner, D. Mathaikutty, J.-P. Talpin, and S. Shukla. Presentation and formal verification of a family of protocols for latency insensitive design. Technical Report 2005-02, FERMAT, Virginia Tech, 2005.

[SSL+92] E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton, and A. Sangiovanni-Vincentelli. SIS: A system for sequential circuit synthesis. Technical report, U.C. Berkeley, May 1992.