Speculation in Elastic Systems *

Marc Galceran-Oms
Universitat Politècnica
de Catalunya

Jordi Cortadella
Universitat Politècnica
de Catalunya

Mike Kishinevsky
Strategic CAD Lab.
Intel Corp.

Abstract

Speculation is a well-known technique for increasing parallelism of the microprocessor pipelines and hence their performance. While implementing speculation in modern design practice is error-prone and mostly ad-hoc, this paper proposes a correct-by-construction method for implementing speculation in Elastic Systems. The technique is based on applying provably correct transformations such as early evaluation, insertion of anti-tokens and bubbles, retiming, and sharing. It allows to explore different micro-architectural solutions for better design trade-offs. The benefits of speculation are illustrated with two examples in which these transformations are systematically applied. The method proposed in this paper is amenable for automation in a synthesis flow.

1 Introduction

Speculation is a well-known technique to increase the instruction level parallelism in pipelined microprocessors. When the outcome of an operation is unknown during some cycle, but is required to perform another operation, two schemes can be considered for a correct behavior: (1) stall the pipeline until the first operation has completed, or (2) predict the outcome of the operation and continue the computations without stalling the pipeline. In the second case, the predicted result must be checked for correctness after the first operation has completed and, in case of *misprediction*, the speculated computations must be invalidated. If the predictions are highly accurate, speculation may potentially provide a tangible performance improvement.

The most typical example of speculation is in the execution of branch instructions when the target address is predicted without knowing the outcome of the branch. Once the branch condition is calculated and a misprediction is detected, the speculated instructions are invalidated and the correct flow is started.

Elastic systems

Elastic systems, either synchronous or asynchronous, are characterized by their tolerance to the variability of communication and computation latencies or delays [15, 5, 8]. This tolerance enables the exploration of new micro-architectural trade-offs aimed at the optimization for the average case rather that the worst case.

Elastic systems use distributed handshake controllers to control the flow of data (*tokens*) along the datapath. These controllers implicitly take care of stalling early data items when they need to be synchronized with other late data items. A pair of handshake signals (usually called req/ack in asynchronous or valid/stop in synchronous systems) control the activity at each communication channel.

*This is an extended version of a paper published at the Design Automation Conference, San Diego, July 2009. This work has been partially supported by a grant from Intel Corp., CICYT TIN2004-07925 and FI from AGAUR (Generalitat de Catalunya and European Funds). We thank Verific for permitting us to use their front-end tools.

A synchronous system can be transformed into elastic by generating a set of distributed controllers that guide the flow of data inside the datapath. To incorporate elasticity, the storage units (registers) must be transformed into elastic buffers supervised by the distributed controllers.

Early evaluation

In conventional elastic systems, all inputs must be available in order to start a computation. However, this requirement can be too strict in some cases. For example, multiplexors only need the select signal and the selected data to be valid in order to compute its output.

Recently, different schemes to handle *early evaluation* have been proposed [4, 13, 1, 7]. By relaxing the condition that requires *"all inputs to be valid"*, certain operations can be initiated when sufficient information is available to perform the computation. In these cases, the dispensable data must be ignored when arriving at the computational block. The concept of *anti-token* has been used to define the phase of the protocol that nullifies the dispensable data.

In [7], anti-tokens circulate (or are stored) in a dual network of handshake controllers in such a way that tokens and anti-tokens cancel each other when they meet in a pair of dual controllers.

Contribution of the paper

This paper presents a novel method to add speculation into elastic systems. Speculative designs are obtained by applying a sequence of provably-correct-by-construction transformations to a non-speculative micro-architecture. Thus, it is guaranteed that the speculative design is functionally equivalent to the original one, regardless the prediction strategy used for speculation. The study of prediction schemes for speculation are out of the scope of this paper, even though they have a crucial impact on the performance of the system.

The method presented in this paper can be conceptually applied to any elastic system, either synchronous [5] or asynchronous [15], and customized for any specific elastic protocol. In this paper, we will focus on one specific protocol for synchronous elastic systems for which early evaluation has been incorporated and formally proved to be correct [7].

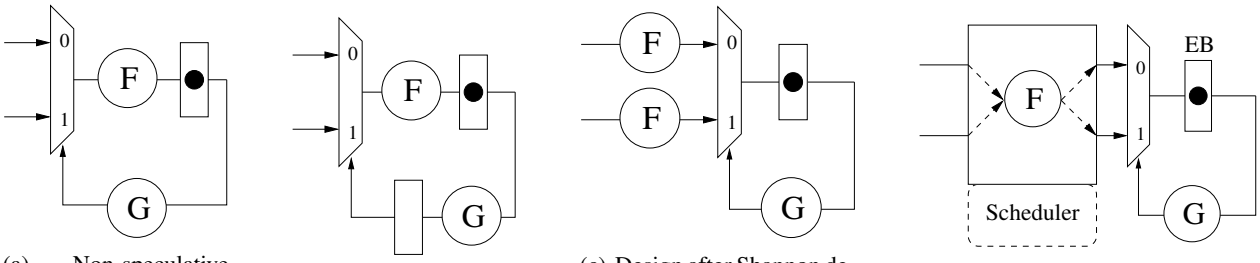
The paper is structured as follows. Section 2 describes the speculation method by means of a simple example. Section 3 introduces synchronous elastic systems. Section 4 presents the implementation details and verification of the controllers for speculation. Section 5 studies two examples that illustrate the benefits of speculation. Finally, some conclusions are drawn in Section 6.

2 Overview

The need for speculation arises when there is a decision point in the datapath in which some of the required data arrives late. Figure 1(a) shows a simple elastic circuit in which speculation can boost up its performance. In this figure, the box is an *Elastic Buffer* (EB), initially containing one valid data item (represented as a token). The circles represent functional blocks. The multiplexor can handle early evaluation when data at the non-selected channel has not arrived yet. Control details are not explicitly displayed, only the data dependencies are drawn.

This scheme could actually be found in a real micro-architecture. For example, the two inputs might be the next PC (Program Counter) and the PC in case a branch instruction is taken. The loop through F and G could represent the computation needed to decide whether the branch is taken. Let us assume that there is a critical path starting at the EB, going through G, the multiplexor, F and arriving at the EB again.

In elastic systems, it is always possible to insert empty EBs (*bubble insertion* [10]) in any channel keeping the same design functionality. Thus, a possible way to optimize the performance of this design



(a) Non-speculative system (b) Bubble insertion (c) Design after Shannon decomposition (d) Design with speculation

Figure 1: Example of speculation in elastic systems

would be to insert an empty EB in the critical path, as shown in Fig. 1(b). While this transformation would improve the cycle time of the design, the throughput of the system would decrease to $1/2$ because there is a loop with one token and two cycles of latency. The token would reach the multiplexor once every two cycles, and during the other cycle there would be no real computation. Therefore, bubble insertion brings no real gain 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) [14], as shown in Fig. 1(c). After moving F from the output of the multiplexor to its inputs, F and G can be executed in parallel rather than sequentially, achieving a better cycle time. Furthermore, the throughput of the system is optimal as there are no bubbles in any cycle. However, this speed-up comes at a price: duplication of logic. Here is where speculation can be effectively used. In order to reduce area, all copies of F can be merged into a single one as shown in Fig. 1(d). Thus, the system can now *speculate* which input channel of the multiplexor should first use the *shared functional module*. A scheduler inside the shared module is the one that must select which input channel can propagate a token through the shared module, thus implicitly predicting the value of the select signal of the multiplexor.

When the speculation is correct, the early evaluation multiplexor receives the required data and computes its output. At the same time, an anti-token is propagated backwards through the channel that was not selected, invalidating the unneeded data. After misprediction, the prediction must be corrected in order to allow the transfer of the correct token.

The design in Fig. 1(c) is optimal in performance. However, if the prediction strategy in Fig. 1(d) is sufficiently accurate, the penalty of speculation will be rarely paid, thus achieving a similar performance with smaller power and area.

A manual implementation of all the stalling/cancelling mechanisms for speculation is complicated and error-prone. This paper presents a set of verified control primitives that can automatically take care of these mechanisms in a distributed control fashion using local handshake protocols. Thus, an automatable and scalable scheme for speculation is provided.

3 Synchronous Elastic Systems

An elastic system can be defined as a collection of blocks and FIFOs connected by channels. A channel is a set of data wires with a tuple of control signals associated: (V^+, S^+, V^-, S^-). Synchronous ELastic Flow (SELF) [8, 7] defines a formal protocol and a set of control circuit primitives for creating an elastic system. The *valid* (V^+) and *stop* (S^+) bits implement a handshake protocol between the sender and the receiver of the channel. The valid bit, going in the forward direction, is set by the sender when some piece

of data (a *token*) is being sent. The stop bit, going in the backward direction, is used for stalling the sender by propagating *back-pressure* when the receiver is not ready.

Analogously, V^- and S^- bits implement the same protocol on the opposite direction, where the pairs (V^+, S^+) and (V^+, S^-) have same semantics but different direction. This second pair of handshake bits is used to propagate *anti-tokens* in the backward direction. When a token and an anti-token meet, they cancel each other, creating a *bubble* ($V^+ = 0$) in the channel.

3.1 SELF Protocol

The protocol used by each one of the (V,S) pairs is based on the one presented in [8], where there are three possible states :

- **(T) Transfer**, ($V \wedge \overline{S}$): the sender provides valid data and the receiver accepts it.
- **(I) Idle**, ($\overline{V}$): the sender does not provide valid data.
- **(R) Retry**, ($V \wedge S$), the sender provides valid data but the receiver does not accept it.

The language protocol observed by a pair (V, S) is described by the regular expression (I*R*T)*. In order to verify the protocol, the following properties expressed in *linear temporal logic* [12] (LTL) are verified in each channel using model checking.

$$\mathsf{G}\,((V^+ \wedge S^+) \implies \mathsf{X}\,V^+) \qquad \text{(Retry}^+\text{)}$$

$$\mathsf{G}\,((V^- \wedge S^-) \implies \mathsf{X}\,V^-) \qquad \text{(Retry}^-\text{)}$$

$$\mathsf{G}\,\mathsf{F}\,((V^+ \wedge \overline{S^+}) \vee (V^- \wedge \overline{S^-})) \qquad \text{(Liveness)}$$

$$\mathsf{G}\,(\overline{V^- \wedge S^+} \wedge \overline{V^+ \wedge S^-}) \qquad \text{(Invariant)}$$

The first and second properties state that the sender of a token or an anti-token has a *persistent* behavior when a *Retry* cycle is produced: valid data is held until transfer occurs. The third one states that every channel will eventually see a token or an anti-token. Finally, as described by the last property, a token cannot be killed and stopped at the same time. The symmetric property must hold for anti-tokens.

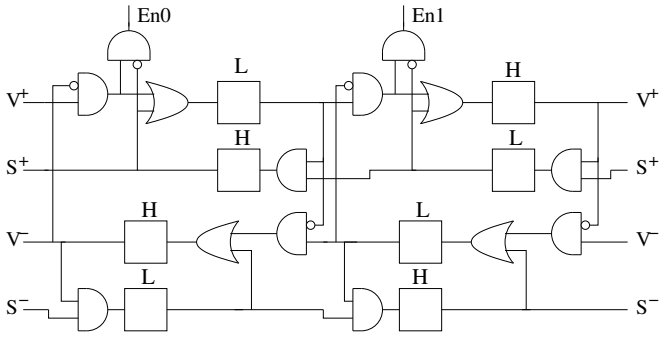
Anti-tokens do not propagate data values, they just propagate the control bit indicating that the next token coming in the channel is not needed. Anti-tokens can be *active*, traveling backwards through the controller; or they can be *passive*, waiting for a token to arrive.

In order to check behavioral equivalence of two conventional synchronous designs, it must be checked that they produce the same output stream when they receive identical input streams. In elastic designs, data transfer count is decoupled from cycle count, and thus, in order to test whether two elastic systems are behaviorally equivalent, it is not necessary to compare output streams cycle-by-cycle. Two elastic systems are *transfer equivalent* [10] if, given identical input streams, the output *transfer streams* match, i.e. the output streams considering only transfer cycles.

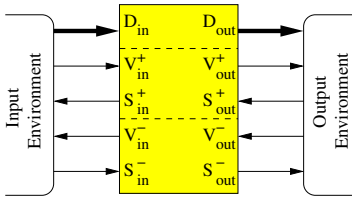
3.2 Elastic Buffers

Figure 2(b) shows the interface of an elastic buffer, a sequential element similar to a conventional synchronous register.

The abstract model for an EB is described in Figure 3. It is an extension of the abstract model for an EB presented in [8] made in order to handle anti-tokens. An EB is an unbounded FIFO which stores tokens (data items) and anti-tokens, which cancel each other at the boundaries of the EB. The notation X_{nxt} represents next-state value of variable X, and the symbol '*' represents a non-deterministic value.



(a) Implementation of an EB controller [7]



(b) Interface of an EB

Figure 2: EB controllers

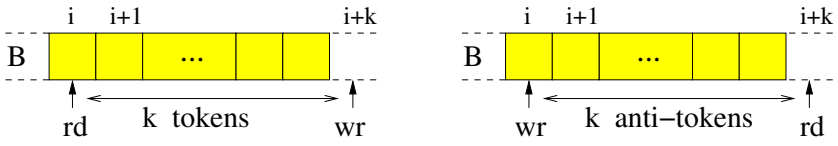
B is an infinite array that stores tokens written to the buffer. Variables wr and rd are the write and read pointers respectively. When wr $>$ rd, the buffer contains $k =$ wr $-$ rd tokens of data. On the other hand, when wr $<$ rd, the buffer contains $k =$ rd $-$ wr anti-tokens. Note that when the buffer stores tokens, each incoming anti-token decrements the amount of tokens, while each incoming token increments the number of tokens by one. When the buffer stores anti-tokens it works in the opposite way. If wr $=$ rd the buffer is empty.

The R^+ and R^- "retry" variables ensure that transmissions are persistent, as required by the protocol. If the buffer contains no tokens (anti-tokens), no transfer can be performed, $V^+_{out} = false$ ($V^-_{in} = false$). Cases of $V^+_{out} = *$ and $V^-_{in} = *$ model the non-deterministic latency of the buffer for propagating tokens and anti-tokens. The stop bits are asserted non-deterministically satisfying the constraint that both tokens and anti-tokens cannot be canceled and stopped at the same time.

The liveness properties expressed in LTL ensure a finite but unbounded latencies in the forward and backward directions for both tokens and anti-tokens. The first property states that each token stored in the buffer must eventually be sent or killed by an anti-token (finite forward response). The second property ensures a finite response time in the backward direction: each time the receiver environment is ready to receive a new token, the buffer must eventually unblock and become ready to receive a new token from the sender. The two symmetric properties must hold for anti-tokens.

Two important parameters of an EB are the forward latency L_f, which is the number of clock cycles needed to propagate tokens through the EB, and the backward latency L_b, which is the number of clock cycles needed to propagate anti-tokens and the stop bit backwards. It is known that the capacity C of an EB must satisfy the following constraint : $C \geq L_f + L_b$. Otherwise, tokens could get lost. E.g. consider $L_f = L_b = 1$ and $C = 1$. If a buffer has a token, but the receiver is blocked, then the sender will learn about it only one cycle later (since $L_b = 1$). In the meantime the sender may want to transmit the next token in which case the transfer will occur (as $V_{in} = 1, S_{in} = 0$) and the previous token inside the buffer will get overwritten and hence lost.

Figure 2(a) shows a possible implementation of an EB for $L_f = 1, L_b = 1, C = 2$, using transparent latches. Boxes labeled with H and (L) indicate active high (low) transparent latches. If the EB is initialized to contain one token, then it is similar to a flip-flop in conventional synchronous designs. If it is initially empty, it is called a *bubble*.



State vars: B : $array\ [0 \ldots \infty]$ of data; rd,wr : $\mathbb{N}$; $R^+, R^- : \mathbb{B}$
Initial state: wr ≥ 0; rd ≥ 0; $R^+ = R^- = false$

Combinational behavior:

$$V^+_{out} = \begin{cases} true & \text{if } R^+ \\ false & \text{if rd} \geq \text{wr} \\ * & \text{otherwise} \end{cases} \qquad D_{out} = \begin{cases} \text{B[rd]} & \text{if } V_{out} \\ * & \text{otherwise} \end{cases}$$

$$V^-_{in} = \begin{cases} true & \text{if } R^- \\ false & \text{if rd} \leq \text{wr} \\ * & \text{otherwise} \end{cases} \qquad S^+_{in} = \begin{cases} false & \text{if } V^-_{in} \\ * & \text{otherwise} \end{cases}$$

$$S^-_{out} = \begin{cases} false & \text{if } V^+_{out} \\ * & \text{otherwise} \end{cases}$$

Sequential behavior:

$$\text{rd}_{nxt} = \begin{cases} \text{rd} + 1 & \text{if } (V^+_{out} \wedge \neg S^+_{out}) \vee (V^-_{out} \wedge \neg S^-_{out}) \\ \text{rd} & \text{otherwise} \end{cases}$$

$$\text{wr}_{nxt} = \begin{cases} \text{wr} + 1 & \text{if } (V^+_{in} \wedge \neg S^+_{in}) \vee (V^-_{in} \wedge \neg S^-_{in}) \\ \text{wr} & \text{otherwise} \end{cases}$$

$$R^+_{nxt} = V^+_{out} \wedge S^+_{out} \quad R^-_{nxt} = V^-_{in} \wedge S^-_{in} \quad \text{B[wr]}_{nxt} = D_{in}$$

Liveness properties (finite response latencies):

$$\mathsf{G}\,(rd < wr \implies \mathsf{F}\,(V^+_{out} \vee V^-_{out})) \quad \mathsf{G}\,(\neg S^+_{out} \implies \mathsf{F}\,\neg S^+_{in})$$
$$\mathsf{G}\,(rd > wr \implies \mathsf{F}\,(V^-_{in} \vee V^+_{in})) \quad \mathsf{G}\,(\neg S^-_{in} \implies \mathsf{F}\,\neg S^-_{out})$$

Figure 3: Abstract model for an elastic FIFO

3.3 Correct-by-construction Transformations

Design transformations known from the conventional synchronous systems, such as retiming or bypassing, can also be applied in elastic systems. Furthermore, elastic systems support novel correct-by-construction transformations enabling new micro-architectural trade-offs. For example, a method to perform correct-by-construction micro-architectural pipelining was presented in [9].

A characteristic property of elastic systems is that they tolerate changes in the latency of computations and communications. Therefore, it is possible to insert or remove an empty EB on any channel while keeping the same performance. An empty EB is equivalent to an EB with one token followed by an anti-token ($0 = 1 - 1$). This rule can be often applied to enable retiming of EBs that have been initialized with a different number of tokens [9].

Simple elastic blocks wait for all inputs before they can start the computation. However, as explained in Section 1, in some cases this requirement can be too strict and *Early evaluation* can be applied instead. Early evaluation is a provably correct transformation that is based on anti-tokens [7].

4 Speculation in Elastic Systems

In this section we will present a method for introducing speculation into an elastic design based on a sequence of provably-correct transformations. This method can be completely automated. As was discussed in section

Cycle	0	1	2	3	4	5	6
F_{in_0}	A	-	C	-	E	F	F
F_{out_0}	A	-	C	-	E	*	F
F_{in_1}	-	B	D	D	-	G	-
F_{out_1}	-	B	*	D	-	G	-
Sel	0	1	1	1	0	0	0
$Sched$	0	1	0	1	0	1	0
EB_{in}	A	B	*	D	E	*	G

Table 1: Example trace from Figure 1(d). '*' = bubble in the channel, '-' = anti-token in the channel, otherwise there is valid data in the channel

2, speculation can be achieved following these steps:

1. Find a critical cycle from an output of an early evaluation multiplexor to its select input. If such cycles exist, speculation is the transformation of choice for increasing the performance. This is because other known transformations such as EB retiming, bubble insertion or early evaluation are useless, and multiplexor retiming alone typically introduces large area overhead.
2. Apply Shannon decomposition to move a logic block backward, out of the critical cycle.
3. Apply early evaluation to the moved multiplexor (these transformations modify the elastic controller, while the datapath stays the same).
4. Share the duplicated logic, introducing the speculation control that instantiates some prediction logic.

Table 1 shows a sample trace of the system from Figure 1(d). F_{in_0} and F_{out_0} denote the input and output channel of the shared module F that serve the first input of the multiplexor, while F_{in_1} and F_{out_1} correspond to second channel of the multiplexor. Sel is the select input of the multiplexor connected to the output of G functional block. $Sched$ is the scheduling signal that carries the channel prediction done for the shared unit F. Finally, EB_{in} is the data value at the input channel of the EB connected to the output of the multiplexor. In cycles $0, 1, 3, 4$, and 6, the correct predictions are made ($Sel = Sched$). During these cycles the early evaluation multiplexor propagates correct value from the selected channel, and the anti-token cancels the token waiting on the unused input channel, since it is not needed. In cycles 2 and 5, however, mispredictions are made ($Sel \neq Sched$). On mispredictions, the multiplexor stalls because the required data is not present. In cycle 2, channel 1 has been chosen by Sel, but F_{out_1} is not valid, since the correct token is being stalled at F_{in_1}. The stop bit on channel 1 will be set by the multiplexor, and this way the scheduler realizes a misprediction has been made, and the value of $Sched$ is corrected during clock cycle 3.

4.1 Sharing of Elastic Modules

Speculation is performed for the shared logic that has been retimed out of the critical cycle. The scheduler selects out of the valid input tokens which one to send for the execution. In other words, prediction can occur only for channels that carry valid tokens.

Let us assume that sharing occurs between two copies of the block like in the example we have considered so far [1]. There are two input data channels to the shared functional module. Let us assume that elastic buffers are inserted into the channels between the shared module and the multiplexor. Let us also assume that both buffers have identical forward latency L_f and identical backward latency L_b. A special case shown in the example in Figure 1(d) (no buffers inserted), correspond to the case of $L_f = 0$ and $L_b = 0$. Let us also

[1] The consideration below can be easily generalized for sharing of k blocks

assume (for simplicity of explanation) that there is an EB at each input of the shared module that stores tokens waiting to be served. For better understanding of the behavior of the speculation unit let us trace the processing of the i-th token, T_i, arriving at one of the input channels of the shared module. The processing goes through 3 steps:

Propagating to the input of the shared module. Since token order is preserved in elastic systems, for the i-th token, T_i to be available at the input in_1, of the shared module, the $(i-1)$-th token, T_{i-1}, in this channel must have been processed or canceled by an anti-token. Let us assume that the speculation controller of the shared module predicted channel 2 and hence did not predict channel 1 during the previous transfer. The T_{i-1} (if already arrived) is stalled at in_1 input channel of the shared module, because it needs to be used in case of misprediction. If prediction of channel 2 was correct, an anti-token is generated by the controller of the multiplexor into channel 1 (out_1). This anti-token propagates backwards reaching in_1 in L_b cycles and cancels T_{i-1}. The token T_i will remain stalled, until the previous token, T_{i-1}, is canceled. Thus, the backward latency of EBs can affect the overall system performance and become a bottleneck.

Prediction by the scheduler of the shared module. Once a token T_i gets to the input of the shared module, the scheduler may predict its channel and then T_i will get propagated through the shared module. Otherwise, the token T_i will be stalled until either the scheduler changes its prediction during one of the future cycles or an anti-token generated by the multiplexor arrives and cancels it out.

Early evaluation in the multiplexor. After a token T_i is selected by the scheduler, it is transmitted through the shared module and then, stored by the output EB, reaching the input of the multiplexor in L_f cycles. If T_i was predicted correctly, once the select signal of the multiplexor becomes available, the early evaluation multiplexor will generate a new token at its output. Otherwise, the token will be stalled at the input of the multiplexor, waiting for the correct token to arrive in the other channel.

4.1.1 Scheduler

A scheduler predicts at each clock cycle which channel can use the shared resource. The performance gain obtained by applying speculation is based on the assumption that the prediction can be done with a high probability of success. The scheduler can implement prediction algorithms of different complexity, from always predicting one of the channels to more advanced algorithms such as the state-of-the-art branch prediction in modern micro-processors.

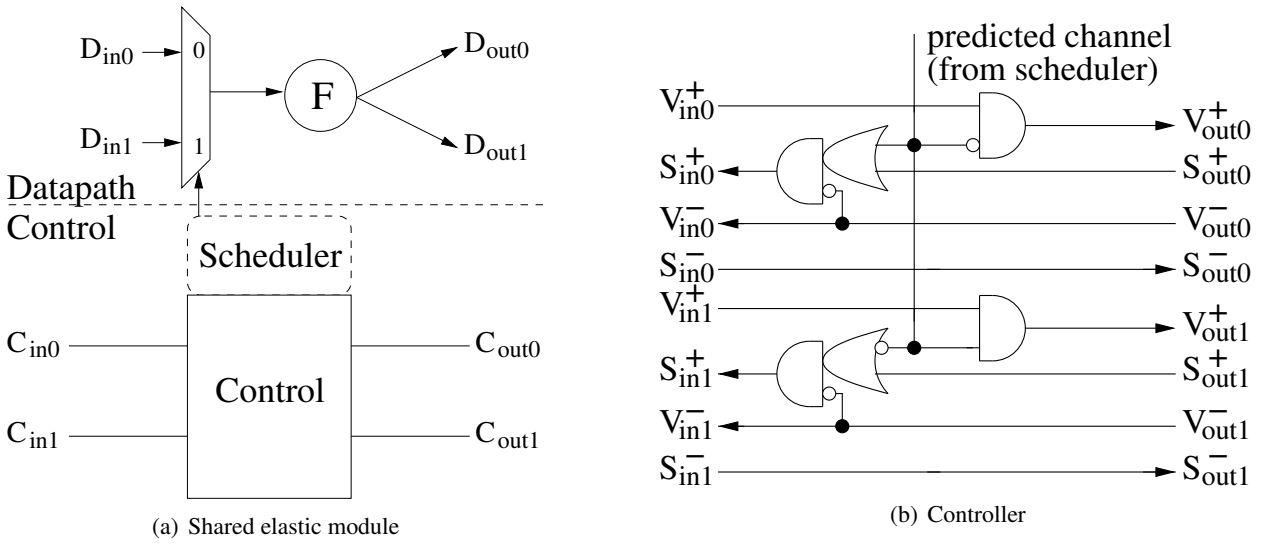
For better performance, the scheduler should take into account the elastic protocol, since a channel that is not valid, or is stalled, cannot use the shared unit even if selected. For correctness of behavior a scheduler should avoid potential scheduling deadlocks. To guarantee this a scheduler should detect and correct all mispredictions. In addition starvation of some channels must be avoided: every token that reaches the shared module must eventually be scheduled unless it is cancelled by an anti-token.

This property can be formalized as a leads-to constraint : if property p is true infinitely often, then property q has to be true infinitely often. The scheduler must satisfy the following leads-to constraint: every arrived token must be eventually served by the shared unit or killed. Formally, for every user of the shared unit, i:

$$\mathsf{G}\ (V^+_{in_i} \Rightarrow \mathsf{F}\ (V^-_{out_i} \vee (sel = i \wedge \overline{S^+_{out_i}}))) \tag{1}$$

4.1.2 Design

Figure 4(a) shows the datapath logic for a combinational block shared by two channels, and Figure 4(b) shows its control logic. C_{in_i} and C_{out_i} represent the handshake control bits of the elastic channels; and D_{in_i} and D_{out_i} represent the datapath wires associated with these control bits. The delay overhead added to the datapath is one multiplexor plus the delay in the scheduling decision. One should make sure that the scheduler is out of the critical path.



(a) Shared elastic module

(b) Controller

Figure 4: Implementation of a shared module

The controller sets the valid signal of the selected channel as long as its input is valid and keeps the valid signal of the other channel at 0. It also stops the other channel (unless it is killed). The kill signal and the stop signal of a channel are mutually exclusive. The implementation of the controller can be trivially extended to handle more than two channels.

4.2 Verification

Since controller design is error-prone, all elastic controllers have been verified with NuSMV [6]. Using a collection of netlists with speculation designed to test different controller combinations, the absence of deadlocks has been verified for *any scheduler* that complies with the leads-to property. In addition, it has been verified that all controllers comply with the SELF protocol and the interaction between the datapath and the controller is correct.

It was verified that even when shared modules add extra latency to the channels, the SELF protocol is still preserved and all tokens are eventually sent. SELF protocol is verified by making sure properties defined in section 3.1 are held on elastic channels. In order to prove that leads-to of schedulers is a sufficient condition for liveness of an elastic system, we applied refinement verification. It has been proven that a shared module sequentially composed with an EB specification with a finite response latency (as defined in Figure 3) is a refinement of an EB specification itself, provided that the shared module has a non-deterministic scheduler specification defined by the leads-to property. The proof has been performed by using SMV [11].

Notice that after a retry cycle on an output channel of a shared module, the scheduler (specified by the leads-to constraints) is allowed to change its prediction. Hence, the output channels of the shared modules are not required to be persistent. However, persistence is maintained at the inputs of the shared module and at the outputs of all EBs after the shared module, which is enough to ensure that no tokens are lost or reordered.

4.3 EBs with Zero Backward Latency

In case of correct prediction, an anti-token is injected on the input channels of the multiplexor without a token. This anti-token needs to rush to the corresponding input channel of the shared module, that stalls data for the misprediction case. Since the prediction was correct, this data is no longer necessary. As it has been

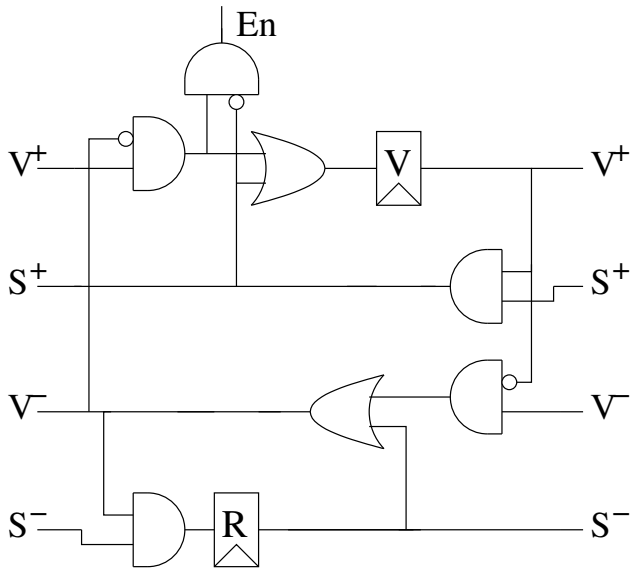


Figure 5: Elastic Buffer with no backward latency

explained in section 4.1, if anti-tokens do not go backwards fast enough, the token to be cancelled delays progress of the subsequent tokens.

Figure 5 depicts a variant of an EB with $L_b = 0$ and $L_f = 1$. Thus, the stop and kill bits travel combinationally through the controller. The capacity C of this EB is $C = L_f + L_b = 1$. Hence it can store one token. This controller uses two flip-flops for forward bits instead of a pair of latches.

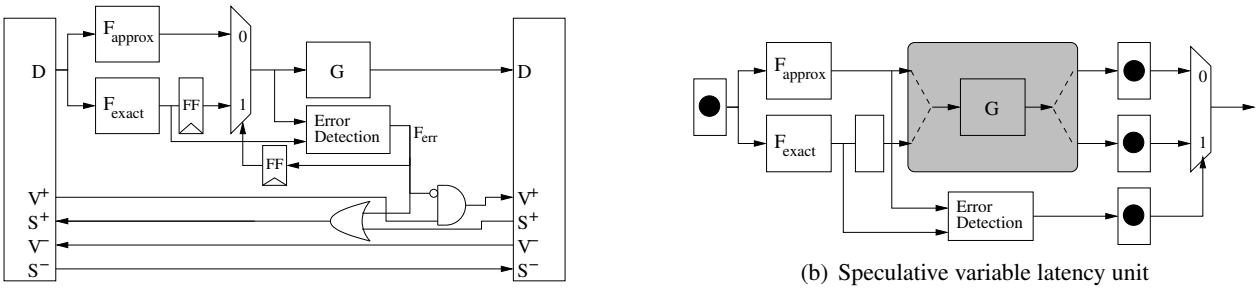
This implementation of EB can be used to reduce overhead of speculation and in other paths where fast backward propagation is beneficial. However, a care must be taken not to chain too many of such controllers to avoid potentially long combinational delays in the control.

5 Examples

In this section we demonstrate the use of speculation combined with elastic systems on two interesting examples. For performing these experiments we have developed a complete framework for exploring elastic systems. Given an abstract netlist representing an elastic system as a collection of modules and FIFOs connected by elastic channels, our toolkit can apply all of the known correct-by-construction transformations under the user guidance in the form of command scripts within an interactive shell. Since all transformations are local they are very fast to compute.

This environment enables fast exploration of the design space. The user can perform transformations, visualize the modified graph, undo and redo the transformations. At any point, it is possible to generate a Verilog netlist of the elastic controller, a blif model for logic synthesis with SIS or a NuSMV model for verification. The elastic controller is built by assembling a set of predefined parameterized control circuit primitives using Verific's front-end tools for manipulating Verilog. For calculating performance of the design point the Verilog netlist of the elastic controller and the specified model of the datapath is simulated and the throughput and the cycle time are reported.

For the experiments the actual datapath of the examples was designed in Verilog, connected to the generated Verilog of elastic controllers, and synthesized using commercial tools with a 65nm technology library.



(a) Stalling variable latency unit

(b) Speculative variable latency unit

Figure 6: Speculation used for variable latency

5.1 Variable Latency Unit

Variable latency units, such as telescopic units [3], optimize the frequent paths of design into a faster single clock cycle, and execute infrequent critical paths in two clock cycles

Variable latency in elastic systems can be handled in a natural way thanks to the handshake protocols. Figure 6(a) shows a variable latency unit which can take 1 or 2 clock cycles to compute. F_{approx} is an approximation of F_{exact} that can be obtained automatically [2], and it has a shorter critical path. Most of the computation cycles, the approximation is correct ($F_{approx} = F_{exact}$), and thus, $F_{err} = 0$. Therefore, the function can be computed within a single clock cycle with no stalling. However, when the approximation is incorrect, F_{err} inserts a bubble into the receiver channel and stalls the sender. The next cycle F_{exact} can be used to finish the computation. In Figure 6(a), F_{err} is connected directly to the controller, which handles the clock gating mechanism to govern the datapath. Since F_{exact} belongs to the critical path of the original design, it is possible that F_{exact} followed by a few gates of the controller is delay critical. Critical paths involving both datapath and control can be difficult to handle.

An alternative implementation that does not have this problem can be based on using speculation with replay in case of an error. The system in Figure 6(b) always speculates that the approximate computation is always correct (in which case the computation is finished in one clock cycle). The shaded box G is shared between the channel coming from F_{approx} and the channel coming from F_{exact} through an empty EB. Whenever F_{err} is asserted, the prediction was incorrect. Then, the next clock cycle, the speculation controller uses the result of F_{exact} computation stored in the bubble, while the early evaluation multiplexor stalls waiting for the correct data. The critical path is taken out of the elastic controller.

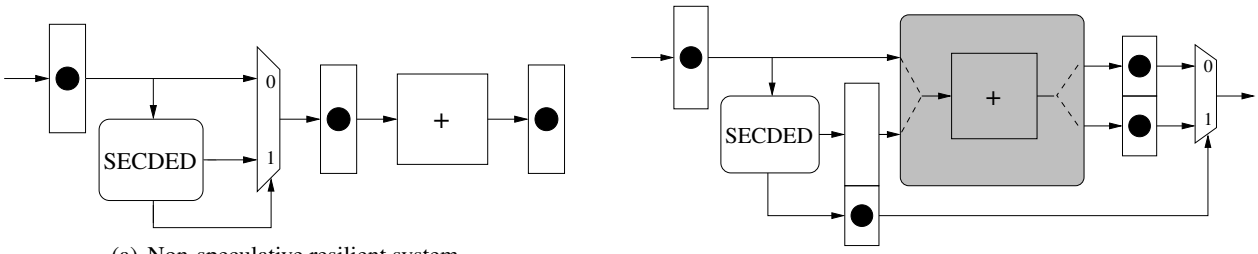
We have implemented a variable latency ALU using a simple pipeline with an 8-bit datapath. In this pipeline, F_{err} has become critical in the stalling unit like in Figure 6(a), but not in the speculative design. Moreover, the speculative design (Figure 6(b)) improves the effective cycle time by 9% with a 12% area overhead. The area overhead is due to extra EBs storing the results after the shared unit.

5.2 Resilient Designs

Speculation can be used to add soft-error detection and correction in a pipeline without changing the performance of the system in case of error-free behavior. As an example, we have used the single error correction and double error detection mechanism (SECDED)[16]. For each 64 bits of data, 8 extra bits allow to detect and correct any single bit error. Besides, double bit errors are detected as well. Some implementation details of SECDED can be found in [17].

Figure 7(a) shows an adder where soft-error checking is done on each input [2]. SECDED needs a whole

[2]To simplify the figure, only one of the two inputs of the adder is drawn.



(a) Non-speculative resilient system

(b) Speculative resilient system

Figure 7: Speculation used for error correction and detection

pipeline stage, and thus, the pipeline is deeper compared to a design with no error checking. Speculation can be used to start the addition without waiting for SECDED to finish, as shown if Figure 7(b), after applying Shannon decomposition and sharing. The system always predicts that no errors will be found and the execution of the addition starts normally. At the same time, SECDED is computed on both inputs to detect errors in the input data. Next cycle, if SECDED detected an error, the mispredicted computation is stalled at the input of the multiplexor, and the addition is restarted using the corrected values coming from the SECDED unit. Thus, the scheduler must only listen to the outcome of the SECDED unit to decide which decision to make. If there were errors last cycle, the addition is replayed with corrected values, otherwise, a new operation is started.

This design has been synthesized using a 64-bit prefix-adder, and it has been checked that there is no performance penalty during the error-free behaviors. Whenever an error is detected, a single clock cycle is lost in order to correct the data and repeat the computation. This mechanism can also be used for error-protection of memories and register files. The area overhead due to speculation in Figure 7(b) is 36%, caused mainly by the recovery EBs necessary for speculation. Notice that this overhead is paid on a single pipeline stage, and hence, it would be amortized across the whole system when implemented on a real pipeline.

6 Conclusions

A novel method for applying speculation in elastic systems has been proposed. It is performed by applying Shannon decomposition and module sharing to a non-speculative design. Since both transformations are correct-by-construction, functional equivalence is preserved when applying speculation. It has been shown that speculation can be used to enhance performance of two realistic examples involving variable latency units and resilient designs.

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)*, pages 611–618, 2006.

[2] D. Bañeres, J. Cortadella, and M. Kishinevsky. Variable-latency design by function speculation. In *Proc. Design, Automation and Test in Europe (DATE)*, Apr. 2009.

[3] 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*, pages 769–779, 1999.

[4] C. Brej. *Early Output Logic and Anti-Tokens*. PhD thesis, University of Manchester, 2005.

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

[6] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV 2: An OpenSource Tool for Symbolic Model Checking. *LECTURE NOTES IN COMPUTER SCIENCE*, pages 359–364, 2002.

[7] J. Cortadella and M. Kishinevsky. Synchronous elastic circuits with early evaluation and token counterflow. In *Proc. ACM/IEEE Design Automation Conference*, pages 416–419, June 2007.

[8] J. Cortadella, M. Kishinevsky, and B. Grundmann. Synthesis of synchronous elastic architectures. In *Proc. ACM/IEEE Design Automation Conference*, pages 657–662, July 2006.

[9] T. Kam, M. Kishinevsky, J. Cortadella, and M. Galceran-Oms. Correct-by-construction microarchitectural pipelining. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, Nov. 2008.

[10] S. Krstić, J. Cortadella, M. Kishinevsky, and J. O'Leary. Synchronous elastic networks. In *International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, Nov. 2006.

[11] K. McMillan. Verification of infinite state systems by compositional model checking. *Correct Hardware Design and Verification Methods*, 1703:219–237, 1999.

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

[13] R. Reese, M. Thornton, C. Traver, and D. Hemmendinger. Early evaluation for performance enhancement in phased logic. *IEEE Transactions on Computer-Aided Design*, 24(4):532–550, Apr. 2005.

[14] 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.*, pages 1085–1090. European Design and Automation Association 3001 Leuven, Belgium, Belgium, 2006.

[15] I. Sutherland. Micropipelines. *Communications of the ACM*, 32(6):720–738, 1989.

[16] J. Wakerly, C. Jong, and C. Chang. *Digital Design: Principles and Practices*. Prentice Hall Englewood Cliffs, NJ, 2001.

[17] Xilinx. Single Error Correction and Double Error Detection (SECDED) with CoolRunner-II CPLDs. *Application Note XAPP383*, 1:1–4, 2003.