Speculation in Elastic Systems

Marc Galceran-Oms*
Universitat Politècnica de Catalunya
Barcelona, Spain

Jordi Cortadella
Universitat Politècnica de Catalunya
Barcelona, Spain

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

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. 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.

Categories and Subject Descriptors: B.5.2 [Register-transfer-level implementation]: Design Aids.
General Terms: Design, Theory, Verification.
Keywords: Elastic designs, speculation, protocols, synthesis.

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 sufficiently accurate, speculation may potentially provide a tangible performance improvement.

Elastic systems, either synchronous or asynchronous, are characterized by their tolerance to the variability of communication and computation latencies or delays [11, 4, 6]. 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.

Recently, different schemes to handle **early evaluation** have been proposed [3, 9, 1, 5]. 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. For example, multiplexors only need the select signal and the selected data to be valid. In these cases, the dispensable data must be ignored when arriving at the computational block. *Anti-token* [5] can be used to nullify the dispensable data.

Contribution. 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 framework presented can be conceptually applied to any elastic system, either synchronous [4] or asynchronous [11], 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 [5].

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.

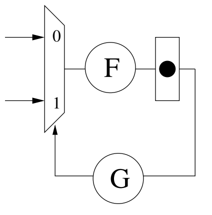
2. OVERVIEW

The need for speculation arises when there is a decision point in the datapath in which some of the required data arrives late. Fig. 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.

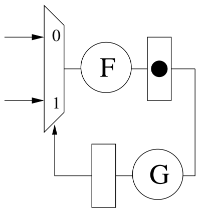
This scheme could actually be found in a real micro-architecture. For example, the two inputs might be the next PC (Program Counter) and the PC in case a branch instruction is taken. The loop through F and G could represent the computation needed to decide whether the branch is taken. Let us assume that 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. Thus, a possible way to optimize the performance of this design 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, it would also decrease the throughput, and no real gain would be achieved.

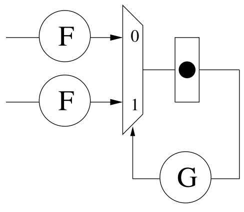
*This work was supported by grants from Intel Corp., CICYT TIN2004-07925 and FI from Generalitat de Catalunya. We thank Verific for permitting us to use their front-end tools.



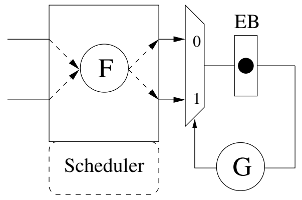
(a) Non-speculative system



(b) Bubble insertion

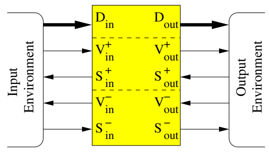


(c) After Shannon decomp.

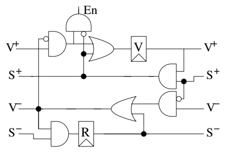


(d) Design with speculation

Figure 1: Example of speculation in elastic systems



(a) Interface of an EB



(b) EB $L_f = 1, L_b = 0$

Figure 2: EB controllers

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 token

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) [10], as shown in Fig. 1(c). After moving F from the output of the multiplexor to its inputs, F and G are executed in parallel rather than sequentially, achieving a better cycle time. Furthermore, the throughput of the system is optimal as there are no bubbles. 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*.

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. When it is not correct, the multiplexor stalls. Then, the correct token must be propagated through F.

The design in Fig. 1(c) is optimal in performance. However, if the prediction strategy in Fig. 1(d) is sufficiently accurate, the cycle clock penalty due to 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. We use 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) [6, 5] 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. This second pair of handshake bits is used to propagate *anti-tokens*.

Elastic buffers, EB, are the sequential elements in an elastic design. An EB is an unbounded FIFO which stores tokens (data items) and anti-tokens, which cancel each other at the boundaries of the EB. Figure 2(a) shows the interface of an EB with one input channel and one output channel. 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 : $C \geq L_f + L_b$. An EB similar to a flip-flop in conventional synchronous designs ($L_f = 1, L_b = 1, C = 2$ initialized with one token) can be efficiently implemented using transparent latches[6]. If an EB does not initially contain any token, it is called a *bubble*. Other implementations can sometimes be useful, Figure 2(b) shows an EB controller with $L_b = 0, C = 1$ for fast propagation of anti-tokens.

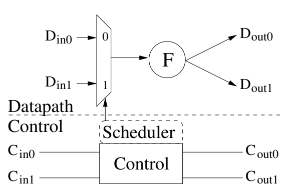
Design transformations known from conventional synchronous systems, such as retiming or bypassing, can also be applied in elastic systems. Furthermore, elastic systems support novel correct-by-construction transformations enabling new micro-architectural trade-offs. For example, a method to perform correct-by-construction micro-architectural pipelining was presented in [8].

4. SPECULATION IN ELASTIC SYSTEMS

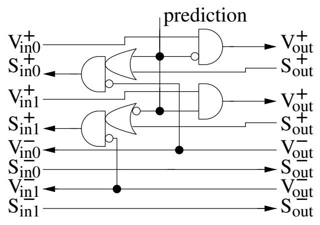
In this section we will present a method for introducing speculation into an elastic design based on a sequence of provably-correct transformations. This method can be completely automated. As was discussed in section 2, speculation can be achieved following these steps:

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.
2. Apply Shannon decomposition to move a logic block backward, out of the critical cycle.
3. Apply early evaluation to the moved multiplexor
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



(a) Shared elastic module



(b) Controller

Figure 3: Implementation of a shared module

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 ($Sel \neq Sched$). The multiplexor is not enabled, stalling and waiting for the correct token to arrive. The next clock cycle, the scheduler corrects its prediction and selects the other token, that will get propagated through the multiplexor.

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.

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 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 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 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.

[1]This consideration can be easily generalized to k blocks

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. Besides, mispredictions can be detected because of back-pressure on the predicted channel. 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 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 tokens arrive infinitely often, then they must eventually be 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 3(a) shows the datapath logic for a combinational block shared by two channels, and Figure 3(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 on 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.

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 implementation of the controller can be trivially extended to handle more than two channels.

Figure 2(b) depicts a variant of an EB with $L_b = 0$ and $L_f = 1$. This implementation of EB can be used to reduce latency overhead of speculation since stalls and anti-tokens can be detected faster. However, a care must be taken not to chain too many of such controllers to avoid potentially long combinational delays in the control.

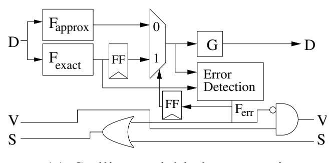
4.1.3 Verification

The absence of deadlocks has been verified for *any scheduler* that complies with the leads-to property (1). 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. More details about the verification strategy can be found in [7].

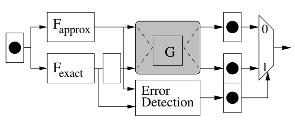
5. EXAMPLES

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



(a) Stalling variable latency unit



(b) Speculative variable latency unit

Figure 4: Speculation used for variable latency

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, and then it is connected to the datapath of the examples. The cycle time is obtained by synthesis, using commercial tools with a 65nm technology library. Finally, the throughput is computed by simulating the whole design.

5.1 Variable Latency Unit

Variable latency units, such as telescopic units [2], 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 4(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 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 4(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.

An alternative implementation that takes the critical path out of the controller can be based on using speculation with replay in case of an error. The system in Figure 4(b) always speculates that the approximate computation is always correct (in which case the computation is finished in one clock cycle). If prediction is incorrect, the next clock cycle, the speculation controller will use the result of the exact computation while the early evaluation multiplexor stalls waiting for the correct data. The shaded box G is shared between the channel coming from F_{approx} and the channel coming from F_{exact} through the bubble.

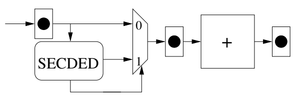
We have implemented a variable latency ALU using a simple pipeline with an 8-bit datapath. In this pipeline, F_{err} has become critical in the stalling unit like in Figure 4(a), but not in the speculative design. Moreover, the speculative design (Figure 4(b)) improves the effective cycle time by 9% with a 12% area overhead. The area overhead is due to extra EBs storing the results after the shared unit.

5.2 Resilient Designs

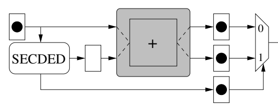
Speculation can be used to add soft-error detection and correction in a pipeline without changing the performance of the system in case of error-free behavior. As an example, we have used the single error correction and double error detection mechanism (SECDED)[12]. For each 64 bits of data, 8 extra 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 [13].

Figure 5(a) shows an adder where soft-error checking is done on each input [2]. SECDED needs a whole pipeline stage, and thus, the pipeline is deeper compared to a design with no error checking.

[2]For simplicity, only one of the two inputs of the adder is drawn.



(a) Non-speculative resilient system



(b) Speculative resilient system

Figure 5: Speculation used for error correction and detection

Speculation can be used to start the addition without waiting for SECDED to finish, as shown if Figure 5(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.

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 5(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.

7. 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] 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.

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

[4] 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.

[5] 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.

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

[7] M. Galceran-Oms, J. Cortadella, and M. Kishinevsky. Speculation in elastic systems. Technical Report LSI-09-15-R, 2009. www.lsi.upc.edu/~techreps/files/R09-15.zip.

[8] 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.

[9] 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.

[10] 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.

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

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

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