Correct-by-construction Microarchitectural Pipelining

Timothy Kam Michael Kishinevsky
Strategic CAD Labs, Intel Corp., Hillsboro, Oregon, USA

Jordi Cortadella Marc Galceran-Oms
Universitat Politècnica de Catalunya, Barcelona, Spain

***Abstract*— This paper presents a method for correct-by-construction microarchitectural pipelining that handles cyclic systems with dependencies between iterations. Our method combines previously known bypass and retiming transformations with a few transformations valid only for elastic systems with early evaluation (namely, empty FIFO insertion, FIFO capacity sizing, insertion of anti-tokens, and introducing early evaluation multiplexors). By converting the design to a synchronous elastic form and then applying this extended set of transformations, one can pipeline a functional specification with an automatically generated distributed controller that implements stalling logic resolving data hazards off the critical path of the design. We have developed an interactive toolkit for exploring elastic microarchitectural transformations. The method is illustrated by pipelining a few simple examples of instruction set architecture ISA specifications.**

I. INTRODUCTION

Pipelining is a powerful microarchitecture transformation for enhancing system throughput without requiring massive replication of hardware [9]. The key impediment to processor performance is pipeline stalling due to data dependency between instructions. The concept of bypass was used as early as [2] and was leveraged for CAD in [8], [20], [15], [19]. Beyond bypassing, dynamic stalling via pipeline interlocks is crucial for improving throughput.

Standard bypass and retiming cannot pipeline cyclic systems. A cyclic system writes to a storage array (e.g. register file or memory) with values functionally dependent on previous reads from the array. Specifications for ISAs, embedded systems, and most RTLs are cyclic in nature. To pipeline cyclic systems, we extend known transformations to synchronous elastic systems and add new correct-by-construction transformations valid for such systems. Synchronous Elastic Systems (aka Latency Tolerant Systems) [3], [4], [12], [6], [5] can tolerate latency changes in computations and communications. In such systems it is possible to insert empty FIFOs, increase capacity of FIFOs, introduce early enabling of multiplexor nodes, and insert anti-tokens to kill irrelevant activities.

Our contribution. This paper presents a novel method for correct-by-construction pipelining of cyclic systems with dependencies between loop iterations. Pipelining is achieved by applying the above listed elastic transformations in addition to standard bypass and retiming. While our method of pipelining relies on previously published elastic transformations (inserting empty FIFOs and early enabling), we demonstrated for the first time how to combine those transformations to achieve optimal pipelining with low overhead distributed controller. For enabling this pipelining method, we also introduced a new transformation, an insertion of anti-tokens onto an elastic channel, and showed that this transformation is required to enable optimal retiming. Without this transformation, the resulting delay-optimal pipelined structure would contain a lot of redundancy in the data-path. Furthermore we demonstrated that, somewhat unexpectedly, capacity sizing of FIFOs is required during retiming moves in elastic systems: without up-sizing of elastic FIFOs, some retiming moves may lead to deadlocks or performance degradation. We show that elastic microarchitectural pipelining can resolve all data hazards found in in-order processors [9]. Instead of implementing a monolithic stall controller that often resides on a critical path, the generated elastic control is fully distributed alongside the datapath

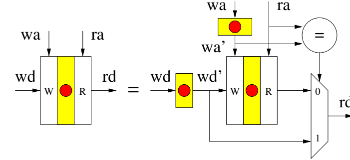

Fig. 1. Bypass transformation for RF or memory

pipeline. While manual crafting of an interlocked pipeline with bypass network and stall controller can be laborious and error prone, our formal method enables user-guided pipeline construction which is provably-correct and as compact as a manual design for our testcases.

In this section we review the classical concepts of pipelining and bypassing, and motivate our work through an example. Dynamic stalling is important for implementing efficient pipelines and correct handling of data hazards. Pipeline stalling is implemented through synchronous elastic systems in Section II. Different pipelines can be naturally constructed by applying trusted transformations, including new elastic transformations proposed in Section III and elastic versions of standard transforms in Section IV. Beside establishing behavioral equivalence between synchronous elastic systems, Section V shows how their performance can be analyzed and optimized. Two examples of elastic pipelining are given in Section VI. Pipelining results are presented with our experimental assumptions and environment. Related work is referenced in Section VII before we conclude.

A. Facts about pipelining and bypass

For acyclic systems, combinational and sequential logic can be pipelined via retiming. Flip-flops for bit signals or registers for data vectors can be introduced at the primary outputs, and retimed backward to pipeline the datapath. For cyclic systems, naive insertion of registers may not be correct. E.g. in Figure 3(a) if a register is inserted between $F1$ and $F2$, the new cyclic system will take one extra cycle to generate results before write-back. So the new system is not sequential equivalent to Figure 3(a) unless back-to-back instructions are not data dependent.

The shaded boxes in subsequent figures represent registers. Tokens inside registers denote valid data. A register with no token represents a bubble (i.e. invalid data). For control abstraction, a register file denoted by RF can be viewed as a monolithic register always having a token. Write select logic W writes data wd at address wa. Data rd is read via read multiplexor logic R for address ra.

Bypass (Figure 1) transforms a register file (or any storage array) to another microarchitecture which though more complicated has a path between the input and the output, independent of the RF latency, when read and write occur at the same address. Specifically, if the read address ra is the same as the previous write address wd', read data rd is forwarded directly from the previous data written wd'. As initial condition, the write data register wd' should be initialized according to the initial value of the write address register wa'_{init} as $wd'_{init} = RF^{orig}_{init}[wa'_{init}]$.

Starting from a register file in a closed loop with a combination logic F, Figure 2 shows the result of applying bypass transformation three times recursively to the RF in Figure 3(a). The loop involving

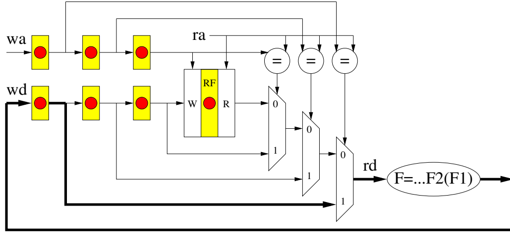

Fig. 2. Bypass of register file in loop with logic

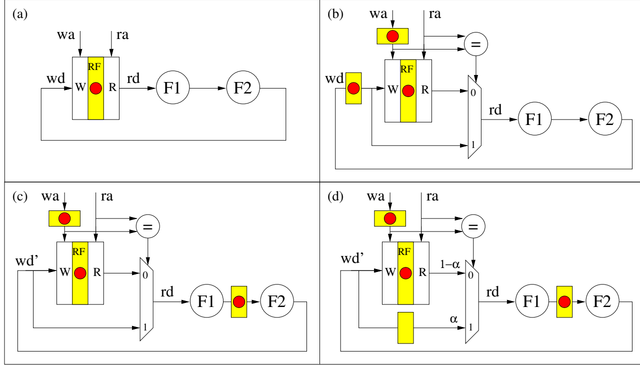

Fig. 3. (a) 1-instruction ISA model; (b) RF bypassed; (c) Retimed pipeline; (d) Bubble inserted

the register file has three additional registers on it so the access of register file has effectively been pipelined. However, Figure 2 still has a loop (marked bold) involving the F logic with only one register. Thus the function F must be computed within one clock cycle and we cannot pipeline this further. *Standard bypass cannot pipeline logic in closed loop with register file or memory.*

B. Motivating example

We illustrate our transformation techniques using simple ISA specifications because they are clear and well understandable examples. Within one cycle, an ISA model fetches and executes an arithmetic (reg-reg) instruction or a memory instruction. Figure 3(a) shows a model computing $RF[wa] = F2(F1(RF[ra]))$ at every clock cycle. A couple more complete cyclic examples will be given in Section VI.

Ignoring the delays of register read and write, the cycle time of the basic architecture (Figure 3(a)) is determined by the sum of delays of $F1$ and $F2$. Assuming the two delays are the same (D), the cycle time is $2D$. The bypass transformation (given by Figure 1) incorporates new registers in the circuit and extra logic (multiplexor and comparator) to determine whether the data must be read from the register file or from the bypass. By retiming the new register, the circuit can be transformed from Figure 3(b) to Figure 3(c). Still, the cycle time is determined by a critical path that starts from the input of $F2$ and ends at the output of $F1$ through the bypass. If we assume that the multiplexor delay is negligible, the cycle time is still $2D$.

Here is where *elasticity* comes into play. A bubble can be inserted at the bypass input of the multiplexor, as shown in Figure 3(d). Now the cycle time is D. However, the bubble also has a negative impact on the performance. The throughput (processed data per cycle) is determined by the cycle containing two registers and one token. This results in processing only one data item every $2D$ time units. Hence the effective performance does not improve.

By using *early evaluation* [5], elastic pipelines attain optimal average-case performance by dynamically selecting bypasses using early enabling multiplexors. When no data hazards are present (i.e.

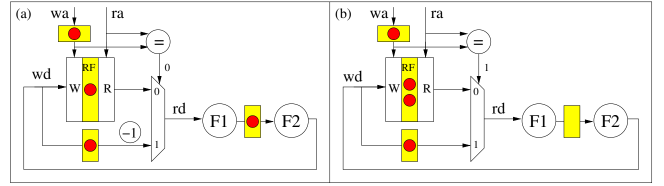

Fig. 4. Simulation cases: (a) $ra \neq wa'$; (b) $ra = wa'$

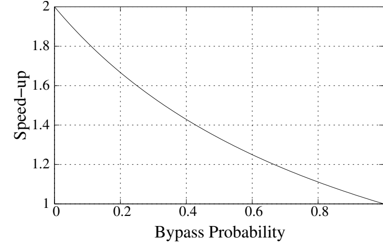

Fig. 5. Speed-up in Figure 3(d)

$ra \neq wa'$), the multiplexor propagates the data read from RF and an anti-token is created in the bypass path to kill the forwarded data, as shown in Figure 4(a). However, when a bypass must be selected, the multiplexor cannot be enabled, and the system will stall to wait for the bypass data, as shown in Figure 4(b).

The exact throughput of the system (computed using Markov chains) is $1/(1+\alpha)$, where α is the bypass probability (data hazard). The speed-up of the pipelined microarchitecture is $2/(1+\alpha)$. Figure 5 shows the relationship between the bypass probability and the speed-up in this example.

II. SYNCHRONOUS ELASTIC SYSTEMS

Specification of synchronous systems typically rely on precise knowledge on latencies (i.e. delays as measured in number of clock cycles) of different computations. Such knowledge that is often required from early stages in design specifications may make the design process highly inflexible to possible changes in communication and computation latencies or delays. In addition, it makes design specifications highly non-modular, complicates the system description, and changes to the specification.

Latency insensitive (aka synchronous elastic) systems [3], [4], [12], [6], [5] are tolerant to changes in latency of computation and communication components. Such systems enable design of functional units optimized for the most frequent cases offering new design trade-offs, simplifies layout convergence, and provides a practical method to separate timing and functionality.

A synchronous elastic system is similar to a conventional circuit, but every data item in it has an associated valid bit. Every functional unit can also issue a stop bit to stall the activity of the preceding units when it is not ready to receive information. These control bits implement a synchronous version of the handshake protocol optimized in comparison with an asynchronous request/acknowledge protocol thanks to the presence of the clock reference.

For the purpose of this paper an elastic system can be viewed as a composition of combinational blocks, register files or memories, and elastic FIFOs connected by channels (a bundle of data wires with a (valid,stop) handshake pair) [1]. The basic case of an elastic FIFO, called elastic buffer, `EB`, has a capacity to store two pieces of information and a latency of one clock cycle. `EB`s in an elastic

[1]The pipelining technique we present in this paper is valid for any implementation of latency-tolerant systems that support early evaluation.

system replace standard synchronous registers. In the absence of a backpressure stop request, an EB acts like a normal register. In the presence of stop, however it can store the second data item, unlike a regular register. [6] demonstrates that an EB can be implemented as a standard master-slave register with a negligible overhead of adding a simple handshake controller.

The latency of propagating data and valid bit between two blocks of the system in the forward direction is equal to the latency of propagating stop backpressure signals in the backward direction [2]. This makes the systems highly scalable as no global stall is required to propagate combinationally within the large design area.

[6] describes an algorithm for converting any synchronous system into a transfer equivalent elastic system [3]. This conversion is done by replacing registers with EBs and augmenting the original design with a handshake controller whose structure corresponds to a data-flow between registers of the original design. The handshake controller is partitioned into small basic blocks: valid-stop control, joins, and forks. A combination of a join, followed by a valid-stop controller, followed by a fork can be associated with each EB.

In this paper we will only draw the datapath of the elastic systems. E.g. Figure 9(a) represents an elastic version of the synchronous bypass system shown in Figure 2 after removing two redundant bypasses (because of the simplifying assumption that only back-to-back dependencies can occur in the design) and splitting the functional block F into three sub-blocks. We do not draw the details of the control (e.g. no valid and stop wires are shown). However we project some control information on the datapath. In particular, we draw EBs as boxes. The box is empty if the EB contains no valid information. The box is marked with a dot, if the corresponding EB contains one token of valid information. In the hardware implementation, presence and absence of a token of information is encoded as a value of the valid bit inside the corresponding to the EB handshake controller.

A. Early evaluation

[3], [4], [6] rely on lazy evaluation: the computation is initiated only when all input data are available. This requirement is often too strict. Consider a multiplexor with the following behavior:

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

Early evaluation can be applied if, for instance, s and a are available and the value of s is *true*. In that case, the result $z = a$ can be produced without waiting for b to arrive and the value of b can be discarded when it arrives at the multiplexor.

In implementing early evaluation, care must be taken in preventing the spurious enabling of functional units when the *unrequired* inputs arrive later than the completion of the computation. A possible technique is the use of *negative tokens*, also called *anti-tokens*. Each time an early evaluation occurs, an anti-token is generated at every unrequired input so that when it meets the positive token they annihilate. The anti-tokens can be *passive*, waiting for the positive token to arrive and accumulated by the up-down counters on the input channels of the early evaluation nodes, or *active*, traveling in the backward direction to meet the positive token. Based on this idea, [5] describes a method of implementing elastic systems with early evaluation (both in the active and the passive form). A similar technique has been proposed for implementing preemption in asynchronous pipelines [1]. [18] suggests a method for implementing early evaluation based on the analysis of the don't care conditions of the datapath logic. Their implementation is of a passive form.

The pipelining described in this paper strongly rely on the idea of the early evaluation (in either passive or active form). All bypass multiplexors (e.g. in Figure 9(a)) are implemented as early evaluation nodes. As a result of early evaluation, the bypass multiplexor in the datapath (even if the latency of the bypass network is long) in absence of dependencies initiates the next iteration of the computation immediately without waiting for the delayed bypass network. In the presence of dependencies as will be explained in Section VI the multiplexor waits until the result of the previous instruction reaches the end of the pipeline before issuing the next instruction. This implements a fully distributed stall controller.

Note that the choice of system nodes to be implemented as early evaluation can be done based on system performance analysis. If there is no significant benefit in implementing a particular multiplexor as an early evaluation node, it can be implemented as a simpler late evaluation node.

III. New elastic transforms

Figure 6 shows some of the correct-by-construction transformations that are valid for synchronous elastic systems (but not for ordinary synchronous systems) that are used in this paper. The left box shows four kernel transformations. The box on the right illustrates two derivative transformations that can be obtained by applying sequences of kernel transformations. The transformation rules from top to bottom are as follows:

BI: Bubble insertion. A characteristic property of elastic system is tolerance to latency changes (for formal proof see [16]). It is therefore possible to insert and remove an empty EB on any channel of an elastic system.

AI: Anti-token insertion. An empty EB (and hence a channel) is equivalent to an EB with one token of information followed by an anti-token, indeed $0 = 1 - 1$. The anti-token injector denoted by -1 is implemented by a modulo-2 up-down counter placed on the elastic channel (without changing the latency of the channel). When the first valid token of information flows across the channel the anti-token kills it by setting a valid bit to false. The anti-token simultaneously disappears (the counter resets to 0). The counter sets back to -1 each time an anti-token is issued by the receiver of the channel and the following two conditions hold: (a) there is no token to kill and (b) no backward propagation possible (either because this is a "passive" implementation without backward propagation, or no free capacity for accommodating the anti-token exists upfront, or blocking occurred due to synchronizing anti-tokens at multiple channels).

AG: Anti-token grouping. Summing up anti-tokens is done by combining modulo-(i+1) and modulo-(j+1) counters into a single modulo-(i+j+1) counter.

AC: Adding capacity. Similar to the **BI** rule, one can always insert an empty elastic FIFO with capacity to hold k items. Moreover, if the latency of the FIFO is equal to 0 (implementable as a FIFO with a bypass from write to read), the performance of the design as measured by the throughput stays the same. Even the effective cycle, a realistic measure of the performance for an elastic design (see Section V-A), does not change, provided that the combinational bypass logic of the FIFO is off the critical combinational path. The rhombus in Figure 6 stands for a 0-latency FIFO with capacity k. If the channel already contained some FIFO (perhaps with some tokens of information) one can always add an extra capacity by appending to the FIFO's input another 0-latency FIFO. These two FIFOs can of course be merged into a single FIFO with bigger capacity. This

[2]This is not a requirement: elastic systems can also be built with different latencies between the forward and backward propagations.

[3]Transfer equivalence is discussed in Section V-C. Intuitively it is an equivalence of sequences of valid data.

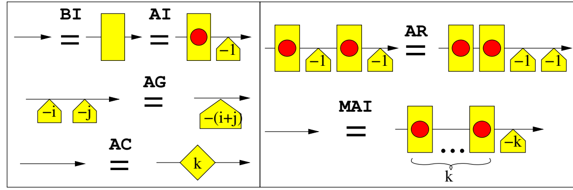

Fig. 6. Elastic transforms

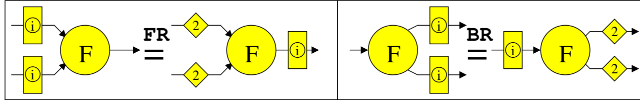

Fig. 7. Elastic retiming moves

operation therefore can be viewed as sizing of elastic FIFOs. Note that increasing capacity of the FIFOs preserves correctness of the system, while increasing capacity without increasing the latency also preserves system performance.

AR: Anti-token retiming. In general retiming of anti-tokens should be performed with care, else it may kill an information token incorrectly after retiming. However, in this situation the left anti-token can be safely retimed forward across the EB with a token, since this token is intended to be killed by the second anti-token anyway. This transformation (left to right) can be derived by first applying **AI** * **BI** right to left to remove the second pair of (token, anti-token) and then applying **BI** * **AI** left to right to the intermediate wire between the token and the anti-token.

MAI: Multiple anti-token insertion. This rule is obtained by applying **BI** * **AI** * **AR** * **AG** recursively k times.

IV. Elastic versions of standard transforms

Well known transformations valid for synchronous systems such as block splitting and merging, bypass and retiming can also be applied to elastic systems, although some of them require modifications. We will call them: **SP: split**, **MR: merge**, **EBP: elastic bypass**, **FR: forward retiming**, and **BR: backward retiming**.

Partitioning of a combinational block into a few sub-blocks is valid for synchronous elastic systems. Of course, a reverse operation - block concatenate - works the same way as well.

A. Elastic bypass

A register file or memory block in an elastic system can be seen by the elastic controller as a block with a token of information (as indicated by the dot inside RF in Figure 9(a)). Elastic bypass transform works the same way as a standard bypass shown in Figure 1 with two exceptions: (a) Registers in the write data and write address channels are replaced with EBs containing a single token of information. (b) The bypass multiplexors are implemented as early evaluation nodes in elastic control.

B. Retiming of elastic buffers

Retiming [17] is a traditional technique for sequential optimization for area or delay. It moves registers across combinational blocks preserving functionality. If the registers are initialized, care must be taken to preserve the correct initial state of the system. If initial values are ignored, then initial behavior of the system may not be preserved. In this case retiming is called steady state retiming. There are efficient methods to separate the steady state retiming from the initial state computation: first retiming is done ignoring the initial state, and then the initial state is computed after retiming. In rare cases, some extra logic must be inserted to compute the initial state correctly. In this paper we take the same view on retiming of datapath registers assuming that, if needed, the correct initial state can be computed after the steady state retiming.

Figure 7 illustrates forward and backward retiming moves for elastic systems. Two differences with respect to regular synchronous systems should be noticed.

- Elastic version of retiming moves EBs instead of registers. In addition to a datapath register an EB also contains an initialized controller that marks the number of tokens of information that fills the EB. For a legal retiming move of a pair of EBs across a combinational node, it is required that both EBs be marked with the same number of tokens, i, where $i \in \{0, 1, 2\}$ for the primitive EB with capacity 2.
- Moving EB's implies moving capacity. In general, reducing channel capacity may lead to performance degradation and even to deadlocks as there might be lack of room in elastic buffers to make progress. E.g., after the **FR** move the capacity of input channels are reduced from 2 to 0 (similarly for a **BR**). As Theorem 1.(A) states this may lead to a deadlock. To guarantee that the deadlock does not occur, following Theorem 1.(B) we conservatively add capacity of 2 to the channels where the EBs have been placed before the retiming. This corresponds to applying a correct-by-construction transformation **AC(2)** before the retiming move and then performing a move.

Theorem 1: Let S be an elastic system without deadlocks, and S' be another elastic system obtained from S by a forward or backward retiming move.

(A) If capacity of channels from which the EBs are moved out is not changed as part of the retiming move, than S' may deadlock.

(B) If capacity of channels from which the EBs are moved out are increased by the capacity of EBs (by 2 in case of simple EBs) as shown in Figure 7, then S' has no deadlocks.

To prove the theorem we first prove that the elastic system with early evaluation deadlocks if and only if the corresponding elastic system with late evaluation does (this follows from [13]). We then construct marked graphs that precisely model behavior of the elastic systems before and after the retiming move and analyze the critical cycles in the graphs. To avoid deadlocks the sum of marking on every cycle should be positive. The construction of marked graphs follows [6], but needs to be extended to handle negative marking and combinational blocks. The detailed proof is omitted due to the lack of space.

V. Analysis and optimization

In this section we will informally discuss a few theoretical topics that underlines the pipelining method. Section V-A explains by example how to measure the performance of elastic systems. Section V-B explains the method of capacity sizing that as a postprocessing minimizes sizes of the conservatively upsized during retiming elastic FIFOs preserving performance. Finally, Section V-C presents the notion of behavioral equivalence in the context of elastic circuits. This equelence is preserved during all the transformations described in this paper.

A. Performance analysis

To introduce metrics used for performance analysis of elastic systems, let us consider a simple example shown in Figure 8. A retiming graph of the original system is shown in Figure 8(a). The weight on a node represents the propagation delay of the combinational block while the edge weight indicates the number of registers along the interconnection between two blocks. The system computes one value

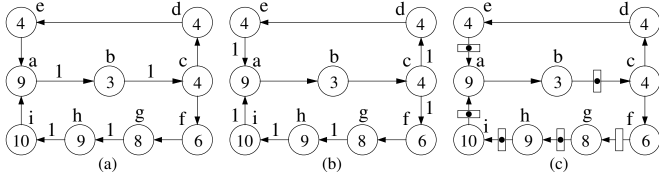

Fig. 8. (a) Initial retiming graph, (b) Min-delay retiming configuration, (c) Retiming and **BI**.

every clock cycle. Hence its throughput is equal to 1. The critical path goes through the combinational nodes c, d, e, a and is equal to 21 time units. This value of 21 is called a system cycle time. After minimal delay retiming as shown in Figure 8(b) the system cycle time is reduced to 16.

Figure 8(c) shows an elastic version of the system, in which all registers are replaced with EBs holding one token of information and the EBs are appropriately retimed. In addition one empty EB is inserted between nodes f and g. The *cycle time* of the graph is equal to 12 time units while the *throughput*, i.e the number of valid data items processed on average at every cycle, is equal to $\frac{4}{5}$.

In our example, throughput is determined by the bottom cycle in which a bubble has been inserted (4 dots in 5 registers). The *effective cycle time* of this system configuration is equal to 15 time units ($15 = 12 \cdot \frac{5}{4}$). It means that a dot item is processed every 15 time units on average, compared to the 16 time units of the min delay retimed original design.

The *effective throughput* = 1/*effective cycle time* corresponds to the number of instructions per time unit as used in computer architecture. The goal of pipelining is to minimize the *effective cycle time* or equivalently to maximize the *effective throughput*.

B. Capacity sizing

Retiming moves with capacity sizing guarantees that after every move a correct elastic system is obtained. However, as a result of applying conservative increase in channel capacity some channels can get oversized. To optimize their sizes we formulate and solve a buffer sizing problem by encoding it as an ILP optimization problem.

As discussed above an elastic system can be modeled by a Timed Marked Graph (TMG) extended to handle negative tokens and early evaluation. Arcs of this graph model both forward flow of information and the backpressure. The forward arcs model forward propagation of valid tokens of information through the system, while the backpressure arcs model the availability of buffers to store more tokens.

The sketch of the buffer sizing algorithm is as follows.

Find best possible throughput for late evaluation. Drop the backward arcs from the TMG, which corresponds to a system with infinite capacities on all channels. View all nodes as late evaluation. Find optimal throughput, Θ_L, of the late evaluation version of the system by one of the well known fast algorithms (e.g. [14]).

Find minimal buffer sizes for late evaluation. Given an original TMG with feedback arcs formulate a buffer sizing problem. This problem is similar to the one described for regular MGs in [13] (for finding an optimal throughput) with the following changes: the throughput is assumed to be fixed to Θ_L, new integer variables are added to model capacity of channels. The objective function is to minimize the sum of capacities. This problem returns the minimal buffer sizes to achieve fastest possible throughput Θ_L assuming the system has no early evaluation nodes. These sizes also guarantees absence of a deadlock in the system with early evaluation.

Tune sizes for early evaluation. Add back early evaluation nodes and formulate a problem similar to the problem in step 2, with the following differences: for early evaluation nodes use formulation from [13]. The objective function is a weighted sum of the maximal throughput and minimal buffer sizes. Additional constraints added on buffer sizes to insure that they would not go below the sizes found in step 2 (this guarantees no deadlock in the solution). These extra constraints are required since the formulation from [13] computes the upper bound of the throughput for systems with early evaluation, not an exact throughput value.

C. Behavioral equivalence

Two sequential designs are said to be behaviorally-equivalent if they produce the same output stream when they receive identical input streams. In classical sequential designs, this equivalence holds cycle-by-cycle, i.e. two designs are equivalent if after receiving the same stream $x_0 x_1 \ldots x_k$, they produce an identical output stream $z_0 z_1 \ldots z_k$, where x_i and z_i represent the input and output values at cycle i, respectively.

With elastic designs, the time dimension is decoupled from the calculations. The cycle-by-cycle equivalence may not hold. Instead, sparse streams of data with void cycles (bubbles) are produced.

Synchronous behavior:	$a_1\ a_2\ a_3\ a_4 \cdots a_k \cdots$ $b_1\ b_2\ b_3\ b_4 \cdots b_k \cdots$
Elastic behavior:	$a_1 \circ a_2 \circ \circ \circ a_3 \circ a_4 \cdots a_k \cdots$ $b_1 \circ \circ b_2 \circ b_3\ b_4 \cdots b_k \cdots$

The above diagram illustrates the behavior of a design with two internal registers, a and b. At each cycle, a different value is stored at the registers. In the elastic behavior, the symbol $\circ$ denotes those cycles with non-valid data (bubbles).

For elastic designs, more general notions of equivalence have been defined in a few slightly different frameworks: latency equivalence [3], flow equivalence [7], or transfer equivalence [16]. These equivalences guarantee that for every output of the design, the order of *valid* data items is the same as in the conventional synchronous design when equivalent input streams are applied. I.e., the elastic and non-elastic behaviors are indistinguishable after hiding the bubbles in the traces of values. This notion of equivalence enables a larger spectrum of transformations for design optimization.

E.g., the sequence of data items at the rd wire of the functional specification in Figure 3(a) is behaviorally equivalent to the sequence of valid data items at rd of Figure 9(c).

VI. Pipelining examples

Continuing on our single instruction example from Figure 2, let us assume for illustrative purpose that operands are either forwarded from the previous instruction (i.e. 1-cycle dependency), or read from the register file after write-backs. As a result, its two innermost bypass paths in Figure 2 (not in bold) are never mux-selected by the middle 2 comparators. So they can be removed to give Figure 9(a).

Our objective is to structurally pipeline the function F by register retiming. However no register is available on the (bold) bypass path of Figure 9(a) for retiming. As argued from Section I-A, standard bypass cannot introduce registers inside a loop. Though the theory of elastic systems allows insertion of empty EBs on this bypass, this won't help either because legal retiming requires EBs with identical number of tokens.

By applying **BI** * **AI** twice, two EBs and two anti-tokens can be inserted on the bypass path as shown in Figure 9(b). Now, we can retime 2 EBs from each fanout path of the fork node. As a result, we have successfully isolated three EBs for backward retiming to pipeline

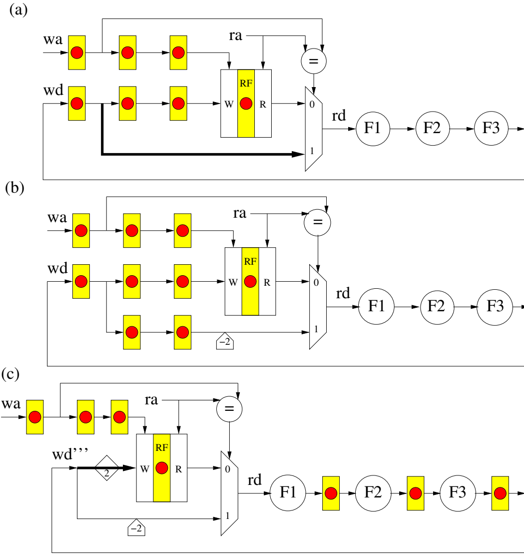

Fig. 9. (a) Bypass assuming 1-cycle dependency; (b) Insertion of anti-tokens and EBs; (c) After retiming and capacity sizing

F into three stages. These retiming moves result in the pipeline shown in Figure 9(c).

With Figure 9(c), every instruction executes in three cycles. Without data dependency (i.e. $ra \neq wa'$), the mux will read operand rd from the register file and the next instruction can start right away. For the case of back-to-back dependent instructions, the bypass will be selected because $ra = wa'$. In this case 2 anti-tokens injected will cancel with 2 tokens from the data pipe, thus stalling the pipeline for 2 cycles (bubbles). Stalling by the minimum number of bubbles is accomplished by the distributed elastic controller. Similar to Figure 3(d), our cycle time can be further decreased by inserting an empty EB on the bypass if F is further partitioned. This step will be omitted from our figures.

Elastic version of retiming increases the capacity of the channel in bold on Figure 9(c). This ensures system liveness by allowing tokens to propagate pass the fork node. Global capacity sizing can also be performed for optimal capacities.

A. From ISA to pipelined microarchitecture

Figure 10(a) shows an ISA spec of four instructions (ADD, MUL, LD and ST). The dual-port register file RF and the memory M are the only state holding blocks. *IFD* fetches instructions and decodes opcode and register addresses. *ADD* and *MUL* are 2-input arithmetic functions. AG generates memory addresses for LD and ST instructions. The results are selected by the bottom-right mux for RF write-back on the next cycle. This model is more complicated than previous ones in that opcode drives not only the mux select signal but also controls the enabling of individual read and write ports on RF and M. The definition of bypass transformation is extended for multi-ported storage arrays with read and write enable signals, which is detailed as the logical expression written next to each bypass mux.

Elastic bypass transform is applied three times on RF to construct a complete bypass network in Figure 10(b), to supply register operands for any given data dependent instructions. Furthermore three EBs are retimed backwards through the mux to form three pipeline stages along the execution datapaths.

Figure 10(b) is not throughput efficient because every instruction still takes exactly three cycles to execute. Thus the right mux is duplicated three times to feed each bypass path independently so that

Fig. 10. Top to bottom: (a) Elastic model of reduced instruction set; (b) After 3 elastic bypassing and retiming; (c) Duplicate mux, move anti-tokens across; (d) Final pipelined after transformations

anti-tokens, representing stalls, can move upstream across the muxes as shown in Figure 10(c). As a result, anti-tokens can cancel tokens in some EBs which improves the throughput of the pipeline. Extra capacities are assigned to some channel as part of retime transforms.

Our final elastic pipeline in Figure 10(d) is optimal in the sense that its distributed elastic controller automatically inserts the minimum number of stall bubbles for every combinations of instructions and data dependencies. Furthermore the pipeline structure is irredundant in that no execution unit nor register is duplicated.

B. Tool kit and flow

We have developed a toolkit to explore elastic transformations. This toolkit, named MAREX, can perform any of the described transformations on a microarchitectural graph.

MAREX represents microarchitectural graphs as a set of nodes that perform computations in the system, and a set of edges that represent elastic channels connecting elastic modules. Edges and nodes can be labelled with attributes, e.g., an edge may contain an EB, and a node has an associated delay and can be marked as an early evaluation node. Input channels of the early-evaluation nodes are labelled with probabilities. MAREX core is implemented in C++ to ensure high performance of the toolkit. The front-end is implemented in python to provide more flexibile and interactive environment.

Transformations are performed one at a time via MAREX python front-end. Thus, pipelining can be performed either through an interactive environment or through a python script. Transformations can be efficiently applied to a microarchitectural graph of arbitrary size, as they affect only a small sub-graph, typically a single channel or a set of channels connected to a single node. It is possible to undo and redo any sequence of transformations. These features enable better exploration of the design space. Furthermore, it is possible to test the performance of a system after every transformations and to display the corresponding graph, such that the user can see the effects of the transformation.

A synchronous elastic system may not perform meaningful computation during every cycle. Hence, an important metric is the amount of computation actually done: the average number of instructions done per cycle (aka *throughput*). To compute the throughput of a graph, we generate an elastic controller in Verilog, and then simulate with random stimuli.

Throughput of our models are dependent on the instruction stream, generated according to probabilities specified. Specifically, the four instructions are assumed an equal probability of occurrence and data-dependencies between them are assume to be evenly distributed between 1 to 16 cycles.

Performance (i.e. *effective throughput*) defined as the number of instructions executed per unit time is given by $\frac{instructions/cycle}{time/cycle} = \frac{throughput}{cycle\ time}$. Cycle time is dependent on the technology library and the microarchitecture. For our experiments, the relative delays of mux, EB, RF, M, AG, ADD, and MUL are assumed to be $1:2:4:10:10:10:100$. Based on these component delays, the cycle time can be computed for a specific pipeline microarchitecture.

C. Experimental results

The original ISA model in Figure 10(a) represents our base case with pipeline $depth = 0$ along the x-axis of Figure 11. On the y-axis, performance of different pipelines are compared against this normalized non-pipelined case. The bottommost curve is obtained by naive pipelining of Figure 10(a) by inserting empty EBs into its elastic model. Every instruction is stalled by the same $depth$ number of cycles. Though correct, its performance decreases monotonically with pipeline $depth$ due to pipeline overhead.

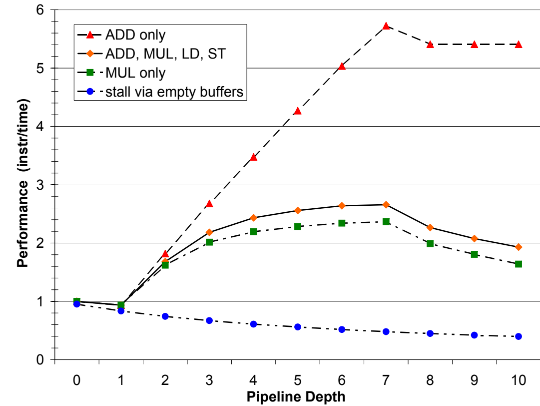

Fig. 11. Performance results of elastic pipelines

Depth :	0	2	4	6	8	10
Control :	78	224	360	496	644	780
Datapath :	4224	5824	7616	9920	12736	16064
Percentage :	1.85%	3.85%	4.73%	5%	5.06%	4.86%

TABLE I

AREA SIZE (IN NO. OF LATCHES) OF ELASTIC CONTROLLER COMPARED TO THE AREA OF A 64-BIT DATAPATH

The upper three curves correspond to the elastic bypassed pipeline of Figure 10(d) under different mixes of instructions. The top curve assumes a stream of ADDs while the third assumes MUL instructions only. As the adder has one tenth the delay of the multiplier, ADD can be computed within one stage for short pipelines. Thus its performance improves linearly with pipeline $depth$ for $depth \leq 7$. In contrast, MUL is the slowest instruction. Leveraging the bypass network and early evaluation of multiplexors, results from most instructions are available sooner for subsequent instructions, instead of waiting for MUL. The second curve assumes equal probability between the four instructions, and its performance lies in between the fastest case for ADDs and the slowest case for MULs.

Distributed controller for elastic bypass stalls the elastic pipeline only when necessary and by the minimum number of bubbles, thus attains high performance. As tradeoff in deciding pipeline depth, on the one hand the deeper pipeline partitions delay into more stages resulting in shorter cycle time, but on the other hand it is penalized more severely by data hazards. Under our experimental assumptions the optimal pipeline depth of 7 was found delivering 2.4 to 5.7 performance speedup for different benchmarks.

The implementation of the control part of an EB, which manages the valid/stop handshakes and generates the enabling signals for the latches in the datapath, has a negligible delay overhead. The number of gates from the output of an EB to the input of another EB is tipically small. Thus, the arrival time of these enabling signals should not have an impact on the cycle time. Furthermore, the area overhead for the elastic controller is not very large as shown in Table I. Given a 64-bit datapath, the number of latches used for the elastic controller is around 5% of the number of latches used in the datapath. The growth of both the size of the datapath and the size of the control is close to linear as a function of the pipeline depth, although the sizes of 0-latency FIFOs and the anti-token counters increase as the number of stages times the number of bypasses. For example for the depth of 3 both sizes are equal to $1 + 2 = 3$.

VII. Related work

As previous work on automatic generation of pipelines, high-level synthesis community described scheduling and resource sharing algorithms (e.g. [21]) for functional pipelining and software loop pipelining, but static schedules cannot handle dynamic dependencies.

[8] shows standard bypass can be applied to register arrays by inserting a negative/regular register pair. The output of negative register is precomputed (or predicted) based on signals and logic spanning multiple cycles. As discussed earlier, logic within a critical loop cannot be effectively pipelined.

Given a pipelined implementation, [20], [15] addressed its verification problem by deconstructing the pipeline via term rewriting to derive an equivalent ISA model. [20] proves by applying microarchitecture laws, e.g. standard bypass, without giving precise design on its timing and control.

[15] describes a method to add forwarding logic and stall engine to a specification already partitioned into stages. The manual design needs to be proved for correctness afterwards. Their global controller also relies on immediately propagation of control signals to all pipe stages whenever necessary.

[19] extracts operations from a specification in the form of term-rewriting rules [11], and schedules them into stages interconnected via FIFO queues. Speculation, stalling and forwarding are accomplished by specific styles of rewriting. A global controller is generated that needs to flush all queues or stall all dependent computations in one clock cycle.

[10] proposes a framework to specify and verify pipelines where designers follow a template to specify pipeline stages and choose from a library of control cells. As a result, verification scripts about pipeline properties can be generated automatically. Their approach doesn't use provably-correct transformations and doesn't generate optimal stalling and bypass based on data dependencies.

In contrast with the above, our method generates a distributed controller in which every local block only controls local bounded queues and local computation blocks. The propagation of control signals may then take multiple cycles without ever being on a critical path of the system. Furthermore we demonstrate that pipelining can be performed correct-by-construction by applying a few well defined structural transformation starting from the original design spec.

While this paper focuses on design refinement, our set of provably-correct transformations should be equally valuable for formal verification approaches [20], [15], [10] in proving transfer equivalence (as defined in Section V-C) between a pipeline implementation and its abstract model (e.g. ISA).

VIII. Conclusions

We have presented a formal method that can pipeline logic in closed loop with register files and memories, which is not possible using standard bypass and retiming alone. This underscores powerful transformations that can be performed in elastic systems, which enable designers to refine an untimed function model towards a cycle-accurate implementation.

By applying a series of correct-by-construction transformations to ISA models, we We also illustrated that correct-by-construction transformations can derive microarchitecture features of processor designs which are not trivial to implement correctly. These includes operand forwarding with bypass network and automated generation of stall controller. We have also used similar transformation techniques to hide memory latency by forwarding stores via load store buffer, and prefetching instructions.

Although this paper focused on pipelining of synchronous systems, these techniques can as well be applied to pipelining of asynchronous systems by implementing (or using already known) asynchronous versions of the synchronous elastic transformations presented here.

Acknowledgments

We want to thank Bill Grundmann for discussing the latency refinement problem with us and the academia. This research has been partially funded by the research projet CICYT TIN2007-66523.

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*, pages 611–618, 2006.

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

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

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

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

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

[7] P. L. Guernic, J.-P. Talpin, and J.-C. L. Lann. Polychrony for system design. *Journal of Circuits, Systems and Computers*, 12(3):261–304, Apr. 2003.

[8] S. Hassoun and C. Ebeling. Using precomputation in architecture and logic resynthesis. In *Proc. International Conf. Computer-Aided Design*, pages 316–323, 1998.

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

[10] J. T. Higgins and M. Aagaard. Simplifying the design and automating the verification of pipelines with structural hazards. *ACM Trans. Design Autom. Electr. Syst.*, 10(4):651–672, 2005.

[11] J. C. Hoe and Arvind. Synthesis of operation-centric hardware descriptions. In *Proc. International Conf. Computer-Aided Design*, pages 511–518, 2000.

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

[13] J. Júlvez, J. Cortadella, and M. Kishinevsky. Performance analysis of concurrent systems with early evaluation. In *Proc. International Conf. Computer-Aided Design*, Nov. 2006.

[14] R. Karp. A characterization of the minimum cycle mean in a digraph. *Discrete Mathematics*, 23:309–311, 1978.

[15] D. Kroening and W. Paul. Automated pipeline design. In *Proc. ACM/IEEE Design Automation Conf.*, pages 810–815, 2001.

[16] S. Krstic, J. Cortadella, M. Kishinevsky, and J. O'Leary. Synchronous elastic networks. In *Proc. Formal Methods in Computer Aided Design*, pages 19–30, 2006.

[17] C. E. Leiserson and J. B. Saxe. Retiming synchronous circuitry. *Algorithmica*, 6(1):5–35, 1991.

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

[19] M.-C. V. Marinescu and M. C. Rinard. High-level automatic pipelining for sequential circuits. In *Proc. International Symposium on Systems Synthesis*, pages 215–220, 2001.

[20] J. Matthews and J. Launchbury. Elementary microarchitecture algebra. In *11th International Conf. on Computer Aided Verification*, Lecture Notes in Computer Science. Springer-Verlag, 1999.

[21] N. Park and A. C. Parker. Sehwa: a software package for synthesis of pipelines from behavioral specifications. *IEEE Trans. on Computer-Aided Design*, 7(3):356–370, 1988.