Automatic Microarchitectural Pipelining

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

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

***Abstract*—This paper presents a method for automatic microarchitectural pipelining of systems with loops. The original specification is pipelined by performing provably-correct transformations including conversion to a synchronous elastic form, early evaluation, inserting empty buffers, anti-tokens, and retiming. The design exploration is done by solving an optimization problem followed by simulation of solutions. The method is explained on a DLX microprocessor example. The impact of different microarchitectural parameters on the performance is analyzed.**

I. Introduction

The structure of the pipeline is one of the key decisions in the early design stages of a system. However, microarchitectural pipelining is often done ad hoc, due to the significant computational costs of simulation during the design space exploration and the lack of analytical optimization methods capable of pipelining in the presence of dependencies between iterations.

For a given workload, there is an optimum pipeline depth that delivers the best possible performance [1]. Thus, specialization of the CPU cores and IP blocks can significantly increase their performance. Such specialization would benefit from automation during early design stages.

This paper presents a method that given a microarchitectural graph with delay annotation of functional nodes *automatically pipelines* this graph such that a near optimal performance is achieved. The algorithm can also output a set of Pareto-points with different clock cycles and throughputs such that a designer or an architect can select best suited for the application. Starting from a functional unpipelined specification as shown in Fig. 1(a), our algorithm produces a pipelined specification as in Fig. 1(b). Design space exploration is driven by certain probabilities at the decision points of the microarchitecture, which must be given as an input to the optimization procedure. Using our method a designer can quickly analyze the optimal pipeline depth for a given microarchitecture conducting pipelining studies similar to proposed in [1], [2].

Our method relies on the capabilities of Synchronous Elastic (aka Latency Tolerant) Systems [3], [4], [5]. Such systems can tolerate latency changes in computations and communications. This elasticity enables new microarchitectural trade-offs aiming at average-case optimization rather than worst case. As shown in [6], it is possible to pipeline elastic systems (even in presence of cycles and dependencies between iterations) using a set of correct-by-construction local transformations.

II. Background

An elastic system can be defined as a collection of blocks and FIFOs connected by channels. A channel is comprised from a set of data wires and a few control signals implementing a synchronous handshake. Synchronous ELastic Flow (SELF) [7], [4] defines a formal protocol and a set of control circuit primitives for creating an elastic system. A pair of control signals bits (*valid* and *stop*) implements a handshake protocol between the sender and the receiver of an elastic 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, implements *back-pressure* and is used for stalling the sender when the receiver is not ready.

This work was supported by grants from Intel Corp., CICYT TIN2004-07925, FPU grant AP2005-4866 and FI from Generalitat de Catalunya. We thank Verific for permitting us to use their front-end tools.

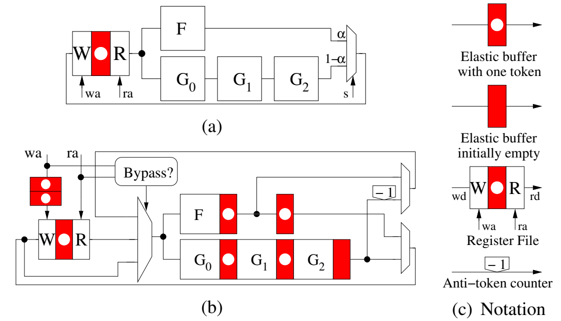

Fig. 1. Example (a) input and (b) output graphs for our method

Elastic buffers (EB) store and transmit tokens through the elastic network. They are replacing registers in conventional synchronous designs, and can be efficiently implemented using transparent latches [4]. Throughout the paper EBs are represented as shown in Figure 1(c). If the EB initially contains some data, it is marked with a dot. Otherwise, it is called a *bubble* and is initially empty. Lines in the figures represent elastic channels: each line represents a set of datapath wires and the associated handshake wires. We assume that EBs have a latency of one clock cycle in absence of back-pressure.

Register files are considered as a special case of EB. The array of memory elements in the register file can be viewed as a buffer. The write logic (denoted "W" in our figures) at the input of the buffer requires the write data wd and address wa channels to carry valid tokens to store a new token inside the array. The read function (denoted "R") requires the read address ra and the previous write operation to be available in order to perform a read and propagate a token to the output channel rd.

Anti-tokens can be propagated backwards in order to nullify irrelevant information [7]. When a token and an anti-token meet, they cancel each other, creating a *bubble* in the channel. Channels may initially store anti-tokens using counters (drawn as pentagons) for canceling a few of the next arriving tokens, as shown in Figure 1(b).

Early evaluation nodes [7] wait only for a required subset of input tokens to start a computation, instead of waiting for all of them. For example, a multiplexor only needs to wait for the select channel and one of data channels (that corresponds to the value of the select bit) to be valid. Once enabled, early evaluation nodes insert an anti-token into the input channels which are irrelevant for this computation. Anti-tokens may stay in place (in the counters) or travel in the backward direction.

Besides standard transformations, such as retiming [8] and bypass, elastic systems with early evaluation enable transformations, shown in Figure 2(a), that allow richer exploration of the design space, preserving system behavior. It is always correct to insert a bubble in an elastic channel (see bubble insertion (BI), transformation also called recycling) and to replace this bubble by an EB with a token followed by an anti-token (AI). The anti-token insertion can be extended to an arbitrary number of anti-tokens. Furthermore, anti-token counters in channels can be grouped (AG) and retimed (AR), as long as the capacity of the counters and the initial number of

978-3-9810801-6-2/DATE10 © 2010 EDAA

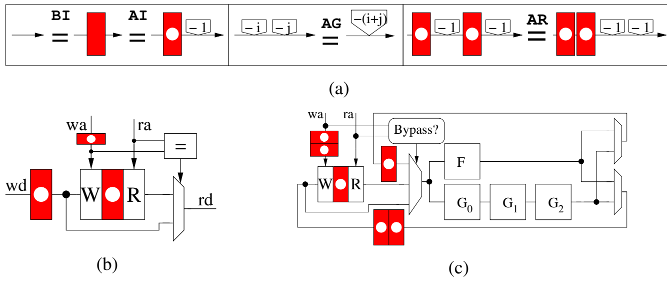

Fig. 2. (a) Elastic transformations [6], (b) bypass, (c) write data forwarding

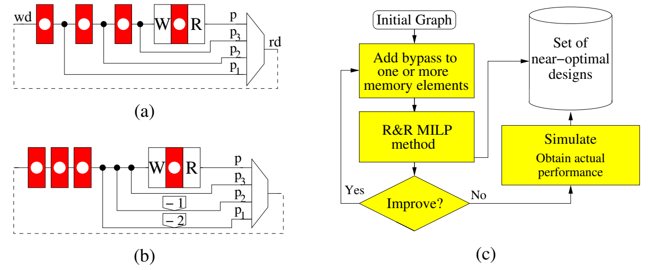

Fig. 3. 3 bypasses (a) before retiming, (b) after retiming, (c) algorithm overview

anti-tokens are preserved.

The *throughput* of an elastic channel is the average number of tokens processed during a cycle. The *effective cycle time*, the cycle time divided by the throughput, is the average time elapsed between two token transfers in a channel and is similar to an average time per instruction in processors. The effective cycle time is the main optimization target of this paper.

The retiming and recycling method [9] captures all transformations from Figure 2(a) in a formal model. A *retiming and recycling configuration*, RC, assigns an initial number of tokens (possibly negative) and a number of EBs to each edge. Since analysis of throughput in elastic systems with early evaluation only provides an upper bound, the heuristic method based on mixed integer linear programming presented in [9] finds a set of Pareto-point RCs/ with different trade-offs between cycle time and analytical throughput.

III. Automatic Pipelining

A. Overview

Starting with a functional specification graph of the design, our method automatically pipelines it by using elastic transformations. First, bypasses are inserted around register files and memories of the functional model. Then, the graph is modified to enable forwarding to the bypass multiplexors. Finally, the EBs inserted with the bypasses are moved to pipeline the design by applying automatic retiming and recycling optimization.

To determine how many bypasses to apply to each memory element, our method uses a combination of a few greedy algorithms, e.g., inserting one bypass at a time, favoring the one that leads to the maximal performance improvement (more details in Section III-E).

During the exploration, the effective cycle time of the design is analytically estimated using TGMG analysis[10]. After the exploration, the most promising points are simulated in order to determine which one is the optimum.

B. Bypasses and Forwarding

Bypasses are widely used to resolve data hazards in processors [11]. Figure 2(b) shows a register file after one bypass. Write of input data is delayed by one EB, and a forwarding path is added, so that if the read address is equal to the write address of the previous instruction (RAW dependency), the correct data value can be propagated, even though it has not yet been written in the register file.

Figure 3(a) shows a register file with 3 bypasses. Write address and read address are omitted for simplicity. Even though three EBs have been inserted, only the leftmost one can be retimed backwards. By using the AI transformation multiple times, EBs and anti-tokens can be inserted on the bypass channel. Then, all the EBs can be retimed out of the bypass structure, as shown in Figure 3(b), and used to pipeline the design along the dotted line. However, the inserted anti-tokens will stall the system on data hazards, waiting for the correct token to arrive.

In addition to stalling, data hazards can also be solved by forwarding. In order to enable forwarding to the bypass multiplexor, some EBs and multiplexors must be duplicated. For example, after bypassing the register file twice in the graph in Figure 1(a), some nodes in the graph are duplicated to achieve the system in Figure 2(c). In this figure, each bypass is fed independently, creating new forwarding paths and retiming opportunities that can lead to the retimed design in Figure 1(b).

C. Two-phase Exploration

Simulations of controllers can take significant time. Thus, it is not feasible to simulate each of the RCs found by the retiming and recycling (RR) solver. In order to prune the design space, bypasses are greedily applied instead of trying all combinations, and the throughput is estimated, rather than simulated, during a first stage of the exploration.

Hence, design space is explored using a two-phase exploration strategy, similar to [12]. During the first phase, bypasses are applied incrementally on the memory elements and RR optimization executed on each step, as shown in Figure 3(c). Within RR, performance for each point is estimated using analytical TGMG analysis [10], and the most promising points are stored. At the end of the exploration, a set of design points with near-optimal performance are simulated in order to determine the overall optimum. Since the relative error between TGMG analysis and simulation is small in most cases, we can safely assume that design points pruned during the first phase are not optimal.

D. Data Hazard Probabilities

In order to perform the TGMG analysis it is necessary to assign probabilities to the inputs of early evaluation multiplexors. For bypass structures, these probabilities should be determined by the expected frequencies of data hazards in the class of workloads for which optimization is done.

We model data hazard probability on bypass multiplexors with a single probability, γ. The register file reads the value written at the previous clock cycle (back-to-back dependency) with probability γ. Then, dependencies are assumed to decrease geometrically. Thus, the probability of p_1 (from Figure 3(a)) to be selected by the multiplexor is $p(p_1) = \gamma$; the register file reads the data value written two clock cycles before with probability $p(p_2) = (1-\gamma)\gamma$; and in general, the probability of distance i dependency is $p(p_i) = (1-\gamma)^{i-1}\gamma$.

E. Exploration Algorithm

During the exploration phase, the Algorithm 1 tries different number of bypasses for each of the memory nodes of the graph. If there is a single register file, the algorithm adds one bypass at each iteration, and then calls the retiming and recycling function (RR) for performance optimization.

The more bypasses the algorithm adds, the less recycling is needed for achieving small cycle time through pipelining. Therefore, the throughput (and the effective cycle time) keeps improving at each iteration, while the cycle times can be kept constant. At some point, either the number of bypasses is enough to fully pipeline the design using retiming, or the throughput degradation due to data hazards stalls in the new bypass is larger than the cycle time improvement due to pipelining. At this point, adding more bypasses does not improve the performance, and the exploration is completed.

If the graph has more than one memory element, the exploration can be performed in a similar way, but the algorithm selects which element to bypass next based on sensitivity analysis. Given a graph G with a set of elements that can be bypassed (memories(G)), Algorithm 1 greedily selects which element to bypass. For every memory m, H.bypass(m) applies one more bypass. Then, the algorithm calls RR optimization and estimates the effective cycle time ($\xi^{lp}(c)$) for each Pareto-point c found by RR.

The node m that leads to the fastest configuration is chosen as the next step, and the greedy algorithm continues into the next iteration. Variable ξ_{loop} keeps track of the best effective cycle time and G_{loop} keeps the best graph found so far within the loop of the algorithm.

Algorithm 1: Bypass_One(G)

explored_points := $\emptyset$; $\xi_{\min} := \infty$; done := false; $G_{loop} := G$
while *not done* **do**
  $\xi_{loop} := \xi_0 := \xi_{\min}$
  for $m \in$ *memories*(G) **do**
    $H := G$
    H.bypass(m)
    RCs := RR(H)
    for $c \in$ RCs **do**
      explored_points.add(c)
      $\xi_{\min} := \min(\xi^{lp}(c), \xi_{\min})$
      if $\xi^{lp}(c) < \xi_{loop}$ **then**
        $G_{loop} := H$
        $\xi_{loop} := \xi^{lp}(c)$
  done := $\xi_{\min} \geq \xi_0 * (1 -$ improve_threshold)
  $G := G_{loop}$
return G, explored_points

When the estimated effective cycle time cannot be improved with a given threshold (improve_threshold, in our experiments is set to 1%), the exploration stops. The priority queue *explored_points* stores the best designs within a given performance overhead compared to the best design found by analytical estimation. At the end, all stored designs are simulated in order to find the best one. The frontier of best solutions is kept in addition to the best one with effective cycle time of $\xi_{\min}$ since analytical LP formulation for the effective cycle analysis is approximate and estimates the upper bound on performance. Therefore some other points in the frontier close to the best estimated may have the best performance (as can be checked by simulation). The ability to store a set of solutions can be also used to extend the algorithm for generating Pareto-points in the (performance, area) solution space.

Some graphs may require several memory elements to be bypassed at the same time in order to reach a performance improvement. On such designs, algorithm Bypass_One may be inefficient. Therefore, we first run an algorithm, called Bypass_All, which is similar to 1 but bypasses all memories once before calling RR.

If a memory node is not further bypassed by the Algorithm 1, it might have too many bypasses. By running Algorithm 1 again starting from the best found design but using *unbypass* transformation instead of bypass, we can further explore the design space. Algorithm 1 with unbypass has a different termination condition: instead of checking that a better design point is found, it checks that the best RR Pareto-point is not significantly worse than the best ξ found so far.

Algorithm 2: Top-level Algorithm

$G_{best}, explored_points_1 :=$ Bypass_All(G)
$G_{best}, explored_points_2 :=$ Bypass_One(G_{best})
$G_{best} := simulate(explored_points_1 + explored_points_2)$
$H, explored_points :=$ Unbypass_One(G_{best})
$G_{best_2} := simulate(explored_points)$

The top-level algorithm, shown in Algorithm 2, calls algorithm Bypass_All first and then algorithm Bypass_One. Next, the best design points are simulated in order to obtain the best one from this first exploration. Since it may be over-bypassed, algorithm Bypass_One with unbypass transformation is called. If unbypass is not applied the algorithm ends. Otherwise, it will explore new design points. At the end, this new explored points are simulated. The performance of G_{best_2} can be at most equal to G_{best}. In our experiments we have never observed a large performance difference between them. Depending on the area gain and the performance loss of G_{best_2}, the user can decide which one to take as the best one.

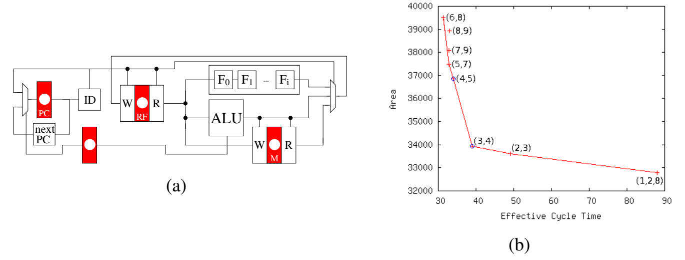

Fig. 4. (a) DLX initial graph, (b) Effective cycle time and area of the best pipelined design for different depths of F. (x,y) and (x,y,z) tuples represent the depth of F, the number of bypasses applied to RF and to MEM ($z = 9$ if omitted)

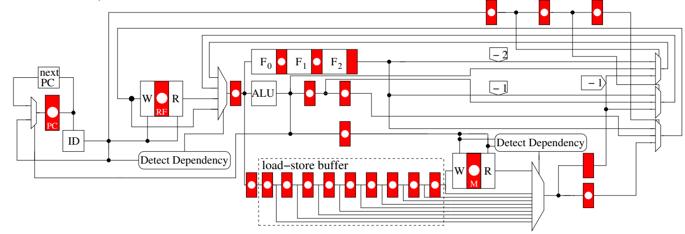

Fig. 5. pipelined DLX graph (F divided into 3 blocks. RF has 3 bypasses and M 9)

IV. Experiments

We have implemented the presented method in our toolkit for exploration of elastic systems. To obtain an accurate throughput of a system, our tool simulates a Verilog controller synthesized automatically from the microarchitectural graph.

We experimented with a large set of micro-architectural graphs to tune the optimization algorithm. In particular, if the algorithm stores only those design points which have an estimated effective cycle time within 1% from $\xi_{\min}$, then the design point with the best effective cycle time is found with a 70% success rate. When the optimal design point is missed, the performance degradation with respect to the optimal point is on average only 3%. The average number of simulations required is 4.125. The success rate of finding the optimal performance design increases to 91% if the solution frontier threshold is set to 5%. The best design point is always found if the threshold is set to 10%. The average number of simulations are 12.6 and 24.3, respectively.

A. DLX Pipeline

We illustrate our method on a simple microarchitecture similar to a DLX, shown in Figure 4(a) before pipelining. The execution part of the pipeline has an integer ALU and a long operation F. Table I shows approximate delays and area of the functional blocks of the example, taking NAND2 with FO3 as unit delay and unit area. In order to obtain these parameters, some of the blocks have been synthesized in a 65nm technology library using commercial tools (ALU, RF, mux2, EB and nextPC), and the rest of the values have been estimated. EB and mux2 delay and area numbers were taken for single bit units. The delay of bit-vector multiplexors and EBs is assumed to be the one shown in the table, while area is scaled linearly w.r.t. the number of bits. Multiplexors with a fan-in larger than two are assumed to be formed by a tree of 2-input multiplexors.

TABLE I
DELAY, AREA AND LATENCY NUMBERS FOR DLX EXAMPLE

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

The register file is 64 bits wide, with 16 entries, 1 write and 2 read ports. The total footprint of the RF is 6000 units, (including both cell and wire area). To account for wiring of other blocks we assume that 40% space is reserved for their wiring. Based on experiments with multiple design points, we assume a 5% area is reserved for the controllers. Given that $\text{Area}_{\text{Blocks}}$ is the area due to the different combinational blocks plus the area of all the EBs, the total area of the design is $\text{Area}_{\text{RF}} + (\text{Area}_{\text{Blocks}} * 1.05)/0.6)$. The area of the initial non-pipelined design shown in Figure 4(a) is 23284 units.

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

Figure 5 shows one of the best design points found by our method under the following design parameters: the F unit has been divided into three blocks, the memory data dependency probability is 0.5 ($\gamma_{\text{MEM}} = 0.5$), and register file data dependency probability is 0.2 ($\gamma_{\text{RF}} = 0.2$), the instruction probabilities are: ($p_{\text{ALU}} = 0.35, p_{\text{F}} = 0.2, p_{\text{LOAD}} = 0.25, p_{\text{STORE}} = 0.075, p_{\text{BR}} = 0.125$). Finally, the probability of a branch taken is 0.5. These values are based on the experiments found in [11], and they are mapped to the early evaluation multiplexors.

In Figure 5, the cycle time is 29.817 time units, due to the F_0, F_1 and F_2 functional blocks. 3 bypasses have been applied to RF and then EBs have been retimed to pipeline F. Note that our algorithm inserted an extra bubble at the output of F_2: the reduction in the throughput due to this bubble, is compensated by a larger improvement in the cycle time (without this bubble the critical path would include the delay of the multiplexors after F_2). Our method does such decisions automatically based on the expected frequencies of instructions and data dependencies.

The inserted memory bypasses are used to hide the memory latency via a *load-store buffer*, as shown in Figure 5. Such structure can be substituted by a more efficient implementation: an *associative memory*. The algorithm automatically detects the need for a load-store buffer and its optimal size.

Figure 4(b) shows the effective cycle time and area of the best design point found by our method on different depths of F, forming a Pareto-point curve. As depth(F) increases, more bypasses are needed on the register file. The area of the design increases with more bypasses. The best effective cycle time is achieved with F divided into 6 stages, 8 bypasses applied to RF and 9 to MEM. Design points (4,5) and (3,4) (circled in the Figure) for 4 and 3 stages are simpler and overall might deliver a better design compromise.

The runtime of algorithm 2 is about 200 sec for every DLX configuration (the longest was 400 sec). The larger depth of F and memory latency increases the run time as more bypasses are needed. About 93% of run time is spent solving RR ILP problems using the CPLEX solver.

We have successfully applied our method to other micro-architectural graphs with a more heterogeneous structure and multiple register files, including a video decoding engine of the industrial SOC. The algorithm scales well up to hundreds of nodes - enough for realistic IP blocks and embedded CPUs.

V. PREVIOUS WORK

A few automatic [13] and semi-automatic [14] pipelining approaches have been discussed in the literature.They relied on use of a global controller for resolving data hazards. Global controllers that handle stalling and logic forwarding may introduce critical paths in the control of design and are generally not acceptable in the nanometer technologies. In contrast, elastic pipelines implement fully distributed and pipelined stall logic avoiding global critical signals.

It is possible to pipeline logic blocks without adding latency using a negative/positive register pair [15]. The output of the negative register must either be precomputed or predicted. This is not possible within a critical loop (unless an expensive unrolling operation is attempted). In our approach, anti-tokens of elastic design can be viewed as a physical implementation of the negative registers that leads to efficient pipelining of critical loops, as long as they contain register files or memories.

Our method is based on the pipelining method presented in [6]. However, [6] was relying on manual application of the above transformations and did not propose a technique for automatic exploration and optimization. Our method enables a better exploration of the design space since it is fully automatic and can handle large micro-architectural graphs hard or impossible to comprehend by a human.

VI. CONCLUSION

A method for automatic pipelining has been proposed. This method takes advantage of optimization techniques available for elastic systems. The method has been effectively applied to several pipeline designs. By setting different parameters on the input graph, a designer can explore different design trade-offs and decide which pipelined design is the best for a given application.

REFERENCES

[1] A. Hartstein and T. R. Puzak, "The optimum pipeline depth for a microprocessor," in *Proc. Int. Symposium on Computer Architecture*, 2002, pp. 7–13.

[2] M. Hrishikesh, N. Jouppi, K. Farkas, D. Burger, S. Keckler, and P. Shivakumar, "The optimal logic depth per pipeline stage is 6 to 8 FO4 inverter delays," *ACM SIGARCH Computer Arch.*, vol. 30, no. 2, pp. 14–24, 2002.

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

[4] J. Cortadella, M. Kishinevsky, and B. Grundmann, "Synthesis of synchronous elastic architectures," in *Proc. ACM/IEEE Design Automation Conf.*, Jul. 2006, pp. 657–662.

[5] 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. Int. Symposium on Asynchronous Circuits and Systems*, Apr. 2002, pp. 3–12.

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

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

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

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

[10] J. Júlvez, J. Cortadella, and M. Kishinevsky, "Performance analysis of concurrent systems with early evaluation," in *Proc. Int. Conf. Computer-Aided Design*, Nov. 2006.

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

[12] S. Eyerman, L. Eeckhout, and K. De Bosschere, "Efficient design space exploration of high performance embedded out-of-order processors," in *Proc. Conf. Design, Automation and Test in Europe*, 2006, pp. 351–356.

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

[14] D. Kroening and W. Paul, "Automated pipeline design," in *Proc. ACM/IEEE Design Automation Conf.*, 2001, pp. 810–815.

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