An Overview of Prediction-Based Architectural Retiming

Soha Hassoun and Carl Ebeling
Department of Computer Science and Engineering
University of Washington, Seattle, WA
{soha,ebeling}@cs.washington.edu

July, 1997
Submitted for TAU97

Abstract

Architectural retiming [4] is a technique for optimizing latency-constrained circuits. It is based on pipelining a latency-constrained path using a pair of registers: a negative register followed by a pipeline register. The negative register which speeds up its input cancels the latency of the added pipeline register. By increasing the number of clock cycles available for the computation, the circuit can run at a smaller clock period. Two implementations of the negative register are possible: precomputation and prediction. In this paper we focus on the prediction implementation of the negative register. We describe the timing characteristics and the operation of the predicting negative register. We present an algorithm for synthesizing the negative register as a finite state machine capable of the verification of the previous prediction, the correction of mispredictions if necessary, and generating new predictions if needed. We describe how don't care conditions can be utilized to minimize the frequency of mispredictions. We also briefly describe two correction strategies that enable circuits to return to correct operations after mispredictions. The concepts are demonstrated using a controller example.

Contact Author:

Soha Hassoun
Department of Computer Science and Engineering, Box 352350
University of Washington, Seattle, WA 98195-2350
Fax: (206)543-2969.
Phone: (206)543-5143.
e-mail: soha@cs.washington.edu

An Overview of Prediction-Based Architectural Retiming

Soha Hassoun and Carl Ebeling
Department of Computer Science and Engineering
University of Washington, Seattle, WA

1 Introduction

In many applications, improving the circuit's performance is of paramount importance. Improving performance usually means increasing the throughput, the rate at which the circuit completes computations, or decreasing the execution time, the total time between the start and completion of one computation. The improvement in performance is often obtained by optimizing the circuit to run at a smaller clock period, even at the expense of increasing the latency, the number of clock cycles between the start and completion of a computation. There are cases however when performance cannot be improved due to a latency constraint. A latency constraint along a path p simply fixes the number of clock cycles that are available for performing the computation. No additional pipelining is therefore allowed to reduce the clock period. Once the combinational delay along p is minimized, and the registers are optimally placed in the circuit, then it is up to the designer to change the circuit's architecture to improve performance.

Latency constraints arise frequently in practice. Some latency constraints are imposed by external performance requirements. For example, if the execution time and the clock period are specified, then only a fixed number of clock periods is available to perform the computation. Other latency constraints are caused by cycles in the circuit. Changing the number of registers on a cycle changes the functionality of the circuit. The latency of each cycle is therefore fixed. External constraints can be modeled as cyclic constraints by feeding the output of an externally constrained pipeline back to the input through a register external to the circuit. The circuit's critical cycle, the one with the maximum ratio of the delay along the cycle to number of registers, can be easily identified using efficient algorithms [2].

Consider the cyclic latency constraint in the controller example in Figure 1. The circuit's critical cycle, consists of the logic through the fetch and decode block, the read operation of the register file, the ALU, the PC (program counter) block, and through the PC register. Retiming [8] in this case does not help. Pipelining decreases the clock period; however, it will not improve the throughput as now the controller can only execute a new instruction every other cycle. The ad hoc solution here is to pipeline the critical cycle, and rely on a guessing mechanism to predict the branch taken condition one cycle earlier. If the mispredictions are kept to a minimum, then the overall throughput and thus the performance of the circuit is improved.

Architectural retiming [4] is a technique for optimizing latency-constrained circuits. It is based on pipelining the latency-constrained path using a pair of registers: a negative register followed by a pipeline register. Unlike the pipeline register which delays its input, a negative register speeds up its input. The latency of the added pipeline register is canceled by the negative register, and the two registers together act like a wire. By increasing the number of clock cycles available for the computation, the circuit can run at a smaller clock period, and the circuit's throughput is improved. Two implementations of the negative register are possible: precomputation, and prediction. In precomputation, the signal at the output of the negative register is precomputed one cycle earlier using the combinational logic in the previous pipeline stage. In prediction-

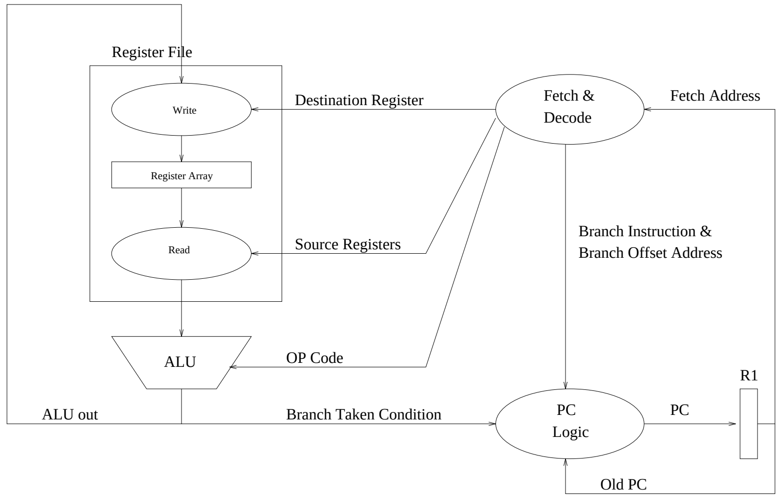

Figure 1: A controller with a critical path from the R1 register, through the fetch and decode logic, the read operation in the register file, the ALU, the logic to compute the program counter, and back to the R1 register

based implementations, the output of the negative register is predicted one cycle before the data actually arrives at the input. The previous prediction is verified each cycle, and the misprediction is corrected if needed.

In this paper we focus on prediction-based architectural retiming. We begin by introducing our model and notation. We then examine the timing characteristics of the predicting negative registers and discuss the effects of mispredictions, both of which form the ground work for the sections that follow. After that, we present an algorithm for synthesizing the negative register as a finite state machine capable of verification, correction, and prediction. We also describe how to utilize don't cares to reduce the frequency of mispredictions. Next, we briefly describe two strategies for returning to correct operation after mispredictions. We conclude with a discussion of work in progress.

2 Circuit Graph

All registers in a circuit are edge-triggered registers clocked by the same clock. Time is measured in terms of clock cycles and the notation x^t denotes the value of a signal x during clock cycle t, where t is an integer clock cycle. Values of signals are referenced after all signals have stabilized and it is assumed that the clock period is sufficiently long for this to happen. A register delays its input signal y by one cycle. Thus, $z^{t+1} = y^t$, where z is the register's output.

Each function f in the circuit has an output signal y computed using N input variables (or signals) $x_0, x_1, \ldots, x_{N-1}$. In a specific cycle t, the variable y is assigned the value computed by the function f using specific values for $x_0, x_1, \ldots, x_{N-1}$, that is, $y^t = f(x_0^t, x_1^t, \ldots, x_{N-1}^t)$.

A synchronous circuit, $\mathcal{C}$, is modeled as a directed graph, $G = (V, E)$. The terms circuit and graph are used interchangeably. The vertex set $V = V^C \cup V^R \cup V^I \cup V^O$ is partitioned into combinational logic, register, primary input, and primary output sets. Each combinational logic

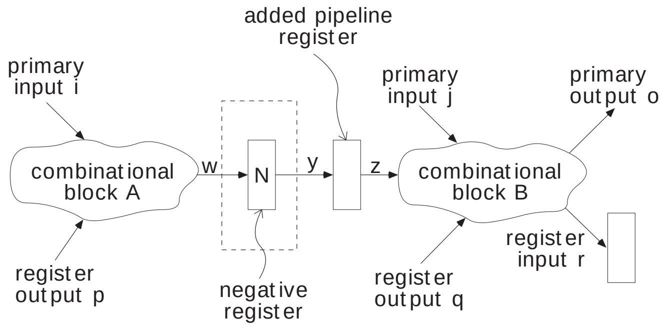

Figure 2: The negative register is followed by the pipeline register, and they are both added along a latency-constrained path containing blocks A and B.

component in the circuit is modeled as a vertex $v \in V^C$, and each register is modeled by a vertex $v \in V^R$. Each primary input is represented with a vertex $v \in V^I$. Similarly, each primary output is represented with a vertex $v \in V^O$. An edge, e, in the set of edges, E, from vertex u to vertex v in V represents an interconnection between the two corresponding circuit components.

A *path* in the circuit between two nodes u and v refers to a sequence of vertices such that each pair of successive vertices has an edge between them. A path is *combinational* if all the vertices along the path are combinational vertices. A *cycle* is a path whose first and last vertices coincide. A cycle is *simple* if each node along the path is distinct, except for the first and last vertices which coincide. When referring to a cycle in the graph, the term cycle refers to a simple cycle unless stated otherwise. All components around a cycle are either register or combinational logic vertices. A combinational cone $\mathcal{CC}_i$ that computes signal i is a subgraph in $G = (E, V)$. It is defined recursively to include the following vertices and edges:

1. if $v_i \in V^R \cup V^I$, then v_i is in $\mathcal{CC}_i$.
2. if $v_i \in V^C \cup V^O$, then v_i and all its in-edges are in $\mathcal{CC}_i$, and the combinational cone of each input to v_i is also in $\mathcal{CC}_i$.

3 The Predicting Negative Register

To perform prediction-based architectural retiming, the register pair, the negative register followed by the pipeline register, is added along the latency-constrained path as illustrated in Figure 2. An instance of the operation of the negative register is shown in Table 1. During each cycle, the negative register verifies the prediction from the previous cycle. If correct, then the negative register generates a new prediction; otherwise, the negative register corrects the previous prediction. The negative register then is responsible for three tasks: prediction, verification, and correction. Figure 3(a) illustrates an implementation of the negative register. In Section 4 we provide an algorithm for synthesizing the negative register as a finite state machine based on transition probabilities provided by the designer.

The number of mispredictions can be decreased by utilizing don't care conditions in the circuit. If the circuit computes correctly (primary output o and register input r in Figure 2) in the cycle

Cycle	w	y	z	Verification	Correct?
$t-2$		v_j^*			
$t-1$	v_j	$v_k^\flat$	v_j^*	$v_j =? v_j^*$	yes
t	v_k	v_k	$v_k^\flat$	$v_k =? v_k^\flat$	no
$t+1$		v_l^*	v_k	No verification necessary	

Table 1: *An instance of the operation of the negative register in Figure 2. A correct prediction is indicated by v_m^*. A misprediction is indicated by $v_m^\flat$. A correct value is refered to as v_m.*

of the misprediction detection regardless of the misprediction, then there is no need to declare a misprediction. Section 4.3 describes how to utilize don't care conditions when synthesizing the predicting finite state machine to decrease the frequency of mispredictions.

As seen in Table 1, when a misprediction occurs, two cycles are required to compute a new value. How does a misprediction then affect the operation and I/O behavior of the circuit? By mispredicting the negative register creates invalid data. This invalid data must be managed correctly, as it may affect the next iterations of the computation once the invalid data propagates through the circuit through feedback loops. To ensure the correct operation of the circuit, we must devise a correction mechanism that would allow the circuit to return to correct operation after each misprediction. In Section 5 we provide two mechanisms for nullifying the effects of mispredictions.

Externally, architectural retiming may change the I/O behavior of the circuit. Because the negative register takes an extra cycle to compute the correct value after a misprediction, there is one cycle during which the circuit's primary inputs and outputs are idle: the circuit cannot consume new data, and the primary outputs are not capable of producing a new correct value.

The effect of applying architectural retiming is to create an *elastic* interface, which provides (consumes) output (input) data at varying rates. Elastic interfaces are implemented by adding handshaking mechanisms to the circuit. The circuit's primary outputs must be paired with validation signals, *data_valid*, to inform the interface whether or not the circuit is providing data. The circuit's primary inputs must be stalled for one cycle for each misprediction to ensure that the circuit is not provided with more data than what it can consume. *Data_not_taken* signals are paired with the primary inputs to indicate when the input data is not taken. The two correction strategies in Section 5 include a description of how handshaking signals are used to ensure the correct operation of the circuit.

4 Oracle Synthesis

We realize the negative register as a finite state machine capable of verifying the previous guess, correcting a misprediction, and providing a new prediction when the previous prediction is verified to be correct. Let's assume that signal w, the input to the negative register, can have any of the following values $v_0, v_1, \ldots, v_{n-1}$. The user is responsible for providing transition probabilities between the different values of w, i.e. $p(v_i \rightarrow v_j)$, for all $0 \leq i, j \leq n-1$[1].

The finite state machine to be synthesized is a Mealy machine, where the new prediction and the misprediction signal is generated in the same cycle when the actual signal arrives for verification (see Figure 3(b)). For simplicity however, we synthesize a Moore machine which provides the

[1]The algorithm can be extended to accommodate more complex probability transition models, including the correlation of signal w with other inputs to the combinational block driving w. The trade-off to the increased accuracy however is the increase in the complexity of the resulting FSM.

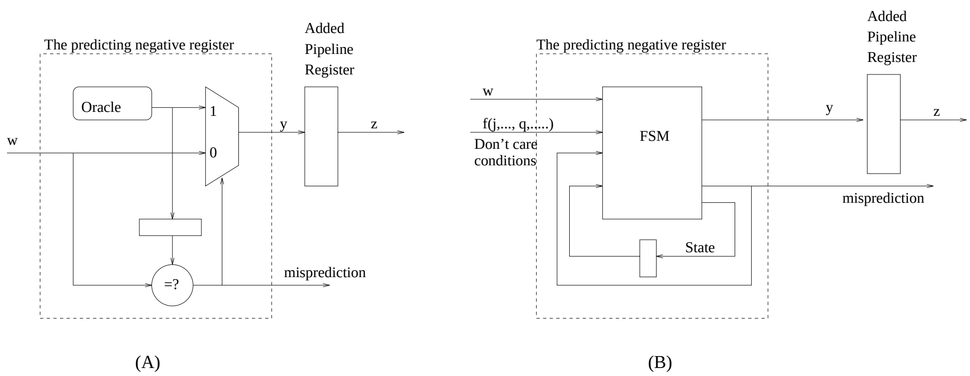

Figure 3: Realizations of the negative register. (a) During each cycle, previous guess is verified. The oracle provides a new prediction if needed. The misprediction signal selects between the new prediction or the correction. (b) A Mealy finite state machine that implements the negative register.

new prediction and misprediction signal one cycle later than needed, as the outputs of a Moore machine are associated with states and not transitions. We then convert the Moore machine to a Mealy machine which allows generating the new prediction and the misprediction signal once cycle earlier than in the Moore machine [7].

4.1 Finite State Machine Construction

A Moore machine is a 6-tuple defined as $M = (Q, \Sigma, \Delta, \delta, \lambda, q_0)$. The set of states in M is Q. The input alphabet is Σ, and the output alphabet is Δ. Both Σ and Δ are equal to $\{v_0, v_1, \ldots, v_{n-1}\}$. The transition function δ is a mapping from $Q \times \Sigma$ to Q. The output function is a mapping from Q to Δ, and it is the value of y at the output of the negative register (either a new prediction or a correction). We describe how to generate the states and the transitions for the Moore machine.

To create the Moore predicting finite state machine, we first create the set of states Q. For each possible value, v_i, we create a correct state,c_{v_i}, and an error state e_{v_i}. The set $Q_c \subset Q$ is the set of correct states, and has the states $c_{v_0}, \ldots, c_{v_{n-1}}$. The output function $\lambda(c_{v_i})$, $0 \leq i \leq n-1$, is equal to v_i. Similarly, the set of errors states, Q_e, is a subset of Q, and consists of the states $e_{v_0}, \ldots, e_{v_{n-1}}$. The output function $\lambda(e_{v_i})$, for $0 \leq i \leq n-1$, is equal to v_i. When in a correct state, then the FSM predicted correctly in the previous cycle. Transitioning into an error state corresponds to declaring a misprediction. The correct state associated with the value that has the highest probability of being correct is chosen as the initial state, q_0. The designer may provide that information, or it can be computed directly from the values of the transition probabilities.

We next create the transitions in the Moore state machine as follows:

1. Create transitions from correct states to correct states.

$$\forall c_{v_i} \in Q_c, p(v_i \rightarrow v_j) = \forall_{0 \leq k \leq n-1} max(p(v_i \rightarrow v_k)) \Rightarrow \delta(c_{v_i}, v_i) = c_{v_j}$$

   From a correct state, we transition to the correct state with the output that has the highest probability of being correct.

2. Create transitions from correct states to error states.

$$\forall_{v_i \neq v_j}, \forall_{c_{v_i} \in Q_c}, \delta(c_{v_i}, v_j) = e_{v_j}$$

Regardless of the probability, we transition on a misprediction to the error state that provides correction.

3. Create transitions from error states to correct states.

$$\forall e_{v_i} \in Q_e, p(v_i \rightarrow v_j) = \forall_{0 \leq k \leq n-1} max(p(v_i \rightarrow v_k)) \Rightarrow \delta(e_{v_i}, -) = c_{v_j}.$$

Here regardless of the input, we transition to the correct state with with the output that has the highest probability of being correct.

Once the transitions are created, we can eliminate all non-reachable states.

4.2 Converting the Moore machine to a Mealy machine

Given a Moore machine $M_1 = (Q, \Sigma, \Delta, \delta, \lambda, q_0)$, we create an equivalent Mealy machine $M_2 = (Q', \Sigma, \Delta, \delta', \lambda', q_r)$ as follows:

- For each state $s \in Q$, we create an equivalent state $s' \in Q'$. In addition, we add a new initial state q_r to Q'. Therefore $Q' = Q \cup \{q_r\}$.
- For each transition $\delta(s_i, v_j) = s_k$ in M_1, we define a transition a similar transition in M_2, $\delta'(s'_i, v_j) = s'_k$. We also define a new transition from the new initial state q_r in M_2 to the state equivalent to q_0. That is, $\delta'(q_r, -) = q'_0$.
- Finally, to generate the output transition function λ', for each $\delta(s_i, v_j) = s_k$, we create a output function $\lambda'(s'_i, v_j) = \lambda(\delta(s_i, v_j))$. FSM M_1 generates an output o_k in state s_k as $\lambda(s_k) = o_k$, and we wish to produce that output as we transition from s'_i to s'_k in M_2.

The derived Mealy machine has the same exact output sequence as the original Moore machine; however, it produces the output sequence one cycle earlier.

4.3 Reducing Misprediction Frequency by Using Don't Care Conditions

There are two important questions to ask when utilizing don't care conditions to reduce the frequency of mispredictions. First, how do we determine if there is a don't care condition which we can take advantage of? More specifically, can we use available signals to decide if the outputs of combinational block B in Figure 2 are insensitive to the fact that the previous prediction does not match the actual value of w? The second question is how do we modify the Moore machine described in the previous section to utilize don't care conditions. We answer both questions.

The True Misprediction Function

Consider the illustration in Figure 3(a). Let's label the inputs to each primary output or register that can be reached from z via a combinational path as h_j, for $0 \leq j \leq m-1$. We also label the primary inputs and register outputs that are inputs to combinational block B, excluding the signal z if needed, as u_k, for $0 \leq k \leq l-1$. We label input z into combinational block B as u_l. A mismatch between the previous prediction value, v_g, and the value of the current input to the negative register, v_t, can be ignored if the correct evaluation of each h_j is not affected by the misprediction. The function that computes true misprediction, f_{TM}, is a function of u_k, for $0 \leq k \leq l-1$, and evaluated based v_g and v_t. It can be computed as follows:

$$f_{TM}(u_0, u_1, \ldots, u_{l-1}) \mid_{v_t, vg} = \bigvee_{\forall 0 \leq i \leq m-1} (h_i(u_0, u_1, \ldots, u_l) \mid_{u_l = v_t} \oplus h_i(u_0, u_1, \ldots, u_l) \mid_{u_l = v_g})$$

If any of the signals h_j evaluated based on v_g differs from the value evaluated based on v_t, we have a true misprediction.

The combinational circuit graph that computes f_{TM} can be easily constructed by first identifying all signals h_j, for $0 \leq j \leq m-1$, and then constructing the combinational cones that drive these signals as a function of the inputs u_k, for $0 \leq k \leq l$. We note here that the true prediction function f_{TM} is a symmetric function, that is $f_{TM} \mid_{v_t, v_g} = f_{TM} \mid_{v_g, v_t}$.

Modifying the Finite State Machine

Now we know how to define f_{TM}. We modify the Moore finite state machine presented in Section 4.1 to use the true misprediction function, f_{TM}, to reduce the misprediction frequency. First we create additional transitions from correct states to other correct states as follows:

$$\forall_{v_t \neq v_{g1}}, \forall_{c_{g1} \in Q_c}, p(v_t \rightarrow v_{g2}) = \forall_{0 \leq k \leq n-1} max(p(v_t \rightarrow v_k)) \Rightarrow \delta(c_{g1}, (w = v_t) \wedge \neg f_{TM} \mid_{v_{g1}, v_t}) = c_{g2}$$

The idea here is that if the previous misprediction was v_{g1}, and the current value of w is v_t and there was no true misprediction, then we should create a transition from $c_{v_{g1}}$ to the state with the output that has the most probability of being correct, $c_{v_{g2}}$.

We also modify the transition function from correct states to error states to only occur when there is a true misprediction. That is,

$$\forall_{v_t \neq v_{g1}}, \forall_{c_{v_g} \in Q_c}, \delta(c_{v_g}, (w = v_t) \wedge f_{TM} \mid_{v_g, v_t}) = e_{v_t}$$

4.4 The Predicting FSM for the Controller Example

We illustrate the synthesis of the FSM that predicts the condition code for the controller in Figure 1. We add the negative register followed by the pipeline register on the edge between the ALU and the PC logic block. The branch taken condition, BTC, can have two values, 0 and 1. We assume that the $p(0 \Rightarrow 0)$ is greater than $p(1 \Rightarrow 0)$, and that $p(1 \Rightarrow 0)$ is greater than $p(1 \Rightarrow 1)$. We also assume that the probability of the branch taken condition being high is less than the probability of it being low.

The synthesized FSM machine is shown in Figure 4. The Moore machine in Figure 4(a-c) has two correct states, c_0 and c_1. The two error states, Q_e are e_0 and e_1. The initial state is c_0. Because the probability of the transition to the value 0 is higher than the value 1, the transition from each correct and error state terminates at c_0.

We now add the don't care conditions. We start by computing f_{TM}. In the original controller, the program counter PC is computed each cycle as follows:

$$PC = (BTC \wedge BI)?(old\_PC + Branch\_Offset) : (old\_PC + 1)$$

where BI is the branch instruction, and BTC is the branch taken condition. Thus, the f_{TM} is:

$$\begin{aligned} f_{TM}(BI, old\_pc, Branch\_Offset) &= PC(BI, old\_pc, Branch\_Offset) \mid_{BTC=0} \oplus \\ &\quad PC(BI, old\_pc, Branch\_Offset) \mid_{BTC=1} \end{aligned}$$

The function can be simplified to:

$$f_{TM}(BI, old\_pc, Branch\_Offset) = BI \wedge (Branch\_Offset \neq 1)$$

It is only when the instruction is a branch and the branch offset address is not equal to one that it is necessary to declare a misprediction. Note that, when simplified, the true misprediction function is not a function of the *old_PC*. The modified Moore machine is in Figure 4(b). The Moore FSM with only the reachable states is shown in Figure 4(c). State c_1 and thus e_0 are non-reachable, and both can be eliminated. The prediction mechanism here is very simple. Initially, upon reset, the FSM predicts BTC to be 0. If the prediction was correct or the PC logic computes correctly (instruction is not a branch or it is a branch but the offset address is 1), then the FSM predicts 0 again and again, until a misprediction is declared. Once a misprediction is declared, the FSM generates a 1, and returns to predicting 0 in the cycle that follows.

The final Mealy machine is shown in Figure 4(d). Misprediction is asserted whenever the FSM transitions into state $s2$. We are currently working on an algorithm to trade off the size of the resulting finite state machine and the prediction accuracy.

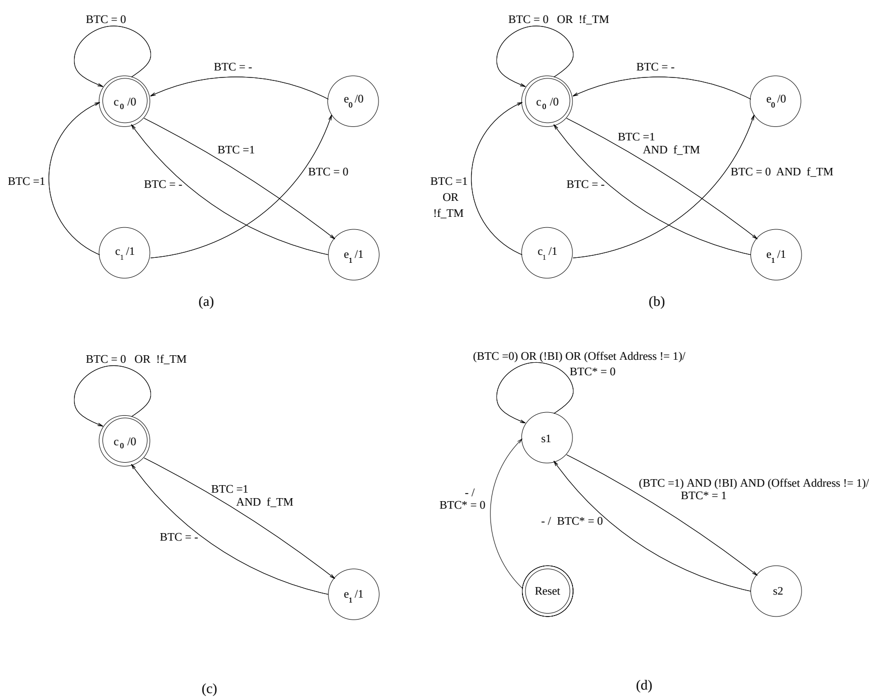

Figure 4: Synthesizing FSM for controller example. (a) Initial Moore FSM. (b) Utilizing don't care conditions to reduce frequency of misprediction. (c) Eliminating non-reachable states. (d) Final Mealy implementation of predicting negative register. f_{TM} is the true misprediction function. Signal BTC^* is the output of the negative register representing the prediction of the branch condition taken.

5 Correction Strategies

One can view recovering from a misprediction as an urgent matter that must be attended to immediately, or as a an accident that one should not worry about until necessary. Two recovery strategies then are possible. The first is *as-soon-as-possible* (ASAP) restoration. It eliminates the effects of a misprediction in the cycle that the misprediction is discovered. The second strategy is *as-late-as-possible* (ALAP) correction. The distinguishing feature of the latter is that invalid data generated by the negative register are allowed to freely propagate throughout the circuit unless a later iteration of the computation is affected. Due to space limitations, we outline the basic ideas for these two strategies and omit the implementation details. An extension of our circuit model to include register arrays as hierarchical graphs and implementation details of the correction strategies are provided in [3].

5.1 ASAP Restoration

The idea for ASAP restoration stems from the desire to turn back time, once a misprediction is detected, and not mispredict. Since that is impossible, the closest alternative is to restore the state and the evaluation conditions from the previous cycle, and correct the prediction. Restoring the state requires setting the outputs of the registers to their value in the previous cycle. Providing the same evaluation conditions requires that the primary inputs retain the same values for two consecutive cycles. The values calculated by the nodes in the circuit with the exception of the output of the negative register will then be identical for two consecutive cycles. In the same cycle during which a misprediction is detected, the primary outputs are marked as invalid to prompt the environment to ignore the duplication. In addition, the circuit will signal *data_not_taken* at the primary inputs thus accommodating the one cycle penalty associated with the misprediction.

5.2 ALAP Correction

The key concepts in ALAP correction is to permit invalid data due to mispredictions to propagate through the circuit and to modify each node to handle the invalid data when they arrive as inputs. We explain what is needed to nullify the effects of the misprediction assuming that the negative register mispredicts only once, and then extend the ideas to include multiple mispredictions.

If the negative register mispredicts in some cycle t, then the misprediction is detected in cycle $t+1$. The output of the added pipeline register is marked as invalid. The data at the primary inputs are marked as invalid, and the circuit's interface is notified (using signal *data_not_taken*) to provide the same data again in cycle $t+2$. Consider any multi-input node, v, in the circuit. Depending on the circuit topology, the invalid data due to the misprediction may arrive at v in any cycle after t. The invalid data may also arrive along the different input edges, E_{in}, to v in different cycles. There is a set of one or more edges, $E_e \subset E_{in}$, along which invalid data first arrives; let's call that cycle t_e. Once the invalid data arrives along E_e, node v is forced to compute invalid data. Meanwhile, the valid data along the input edges $e_l \notin (E_{in} - E_e)$ must not be ignored. Node v is then responsible for temporarily buffering the valid inputs until cycle $t_e + 1$ when valid inputs arrive along the edges in E_{in} and v can compute valid data. In cycle $t_e + 1$, if valid data arrives along an edge e_l, then the data must be buffered again while v is computing based on the valid data that arrived in the previous cycle. The buffering and consuming of the one-cycle old data continues until invalid data finally arrives along edge e_l. Then, node v uses the data in the temporary buffer, and no new data is stored in the buffer. In the following cycle, node v uses the valid data that arrives along e_l. Since each node in the circuit is reachable from either a primary input or the negative register, it is guaranteed that invalid data due to the same misprediction will eventually reach every node in the circuit.

The temporary storage for one valid data is not always sufficient. If invalid data due to a new misprediction arrives along the edges in E_e before the invalid data due to the previous misprediction arrives along e_l, then another valid data must be also temporarily buffered. Moreover, the valid data must be used by node v in the same order that it arrives in. Thus, a first-in first-out queue is needed along the input edges in $E_{in} - E_e$.

An intuitive way to view the invalid data caused by a misprediction is as a *wavefront* that propagates through the circuit. The wavefront first originates at the output of the added pipeline register and at the primary inputs. Each clock cycle, the wavefront moves to the following pipeline stages. With the addition of the FIFO queues along the proper edges, each node in the circuit meets the following criteria:

C1. Upon the arrival of a new wavefront, a node is forced to compute invalid data, as it cannot possibly compute valid data.

C2. For each misprediction that occurs, each node generates only one invalid data.

Because we require that each modified node propagates each wavefront only once (C1), which occurs at the earliest time the wavefront arrives along any of the node's inputs (C2), the movement of the wavefronts is predictable regardless of the circuit topology. We can determine the earliest time that a wavefront reaches a circuit node using a single-source shortest path algorithm. The earliest arrival times can be used to determine the exact sizes of the queues

5.3 The Controller Example: Correction

We give a brief summary of how the controller example is conceptually modified under both correction strategies. We assume that the negative/pipeline register pair are added along the edge between the ALU and the PC logic. When applying ASAP restoration to the controller in Figure 1, we must restore the state of the PC register, as well as the state of the register file. Thus, in the cycle of misprediction detection, each circuit component with the exception of the negative register reproduces the value from the previous cycle. There is a one-cycle misprediction penalty associated with each misprediction.

Applying ALAP correction, we first calculate the earliest arrival times of invalid data generated at the output of the added pipeline register at each node in the circuit. All the nodes have an earliest arrival time of one, with the exception of the PC logic and the PC register which have an earliest arrival time of zero. A FIFO of depth one is then needed along the edge from the fetch and decode logic to the PC logic to hold the valid branch instruction and branch offset address while the PC logic is calculating invalid data. To enforce condition C2, the register file must also be modified to not update it's state in the cycle when the wavefront arrives at the register file's write node.

6 Related Work

Holtmann and Ernst present a scheduling algorithm for high-level synthesis that applies a technique that is modeled after multiple branch prediction in a processor [5]. The ASAP correction is conceptually similar to their approach even though theirs is more general, as they allow more than one cycle between predicting and the verification. They add more than one register set to restore the state in case of a misprediction.

Recently, Benini et al. described a technique similar in its goals to architectural retiming, that is reducing the clock period and increasing throughput [1]. The technique constructs a signal which is asserted whenever an input vector to a combinational block requires more than the reduced target clock period to evaluate. Once that signal is asserted, the combinational block takes two cycles, instead of one, to complete the computation. Radivojević et al. describe a scheduling technique that employs pre-execution [9]. All operations possible after a branch point are precomputed before the branch condition is determined. Once the branch condition is known, one of the pre-executed operations is selected.

7 Conclusion and Status Report

This paper introduces some of the algorithms needed to automate prediction-based architectural retiming. We implement the predicting negative register as a FSM capable of predicting, verification, and correction. We outlined how to use don't care conditions to minimize the frequency of mispredictions. We briefly presented two correction strategies, ASAP restoration and ALAP correction, to enable the circuit to function correctly after mispredictions. We have built a symbolic simulator, ARsim, that validates both correction strategies. We are currently investigating circuit implementation issues to enable us to architecturally retime and evaluate a set of examples.

References

[1] L. Benini, E. Macii, and M. Poncino. "Telescopic Units: Increasing the Average Throughput of Pipelined Designs by Adaptive Latency Control". In *Proc. 34th ACM-IEEE Design Automation Conf.*, pages 22–7, 1997.

[2] M. Hartmann and J. Orlin. "Finding Minimum Cost to Time Ratio Cycles With Small Integral Transit Times". Technical Report UNC/OR/TR/91-19, University of North Carolina, Chapel Hill, Oct. 1991.

[3] S. Hassoun. *"Architectural Retiming: A Technique for Optimizing Latency-Cosntrained Circuits"*. PhD thesis, University of Washington, Work in progress.

[4] S. Hassoun and C. Ebeling. "Architectural Retiming: Pipelining Latency-Constrained Circuits". In *Proc. of 33th ACM-IEEE Design Automation Conf.*, June 1996.

[5] U. Holtmann and R. Ernst. "Combining MBP-Speculative Computation and Loop Pipelining in High-Level Synthesis". In *Proc. European Design Automation Conf.*, pages 550–6, 1995.

[6] J. Hopcroft and J. Ullman. *"Introduction to Automata Theory, Languages, and Computation"*. Addison-Wesley, Reading, Massachusetts, 1979.

[7] Z. Kohavi. *"Switching and Finite Automata Theory"*. McGrwa Hill, 1978.

[8] C. E. Leiserson, F. Rose, and J. B. Saxe. "Optimizing Synchronous Circuitry by Retiming". In *Proc. of the 3rd Caltech Conference on VLSI*, Mar. 1983.

[9] I. Radivojević and F. Brewer. "Incorporating Speculative Execution in Exact Control-Dependent Scheduling". In *Proc. 31th ACM-IEEE Design Automation Conf.*, pages 479 –484, 1994.