Architectural Retiming: An Overview

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

Abstract

Pipelining and retiming are two related techniques for improving the performance of synchronous circuits by reducing the clock period. Unfortunately these techniques are unable to improve many circuits encountered in practice because the clock cycle is limited by a critical cycle which neither technique can change. We present in this paper a new optimization technique that we call *architectural retiming* which is able to improve the performance of circuits for which both pipelining and retiming fail. Architectural retiming achieves this by increasing the number of registers on the critical cycle while preserving the functionality and perceived latency of the circuit. We have used the name architectural retiming because it both reschedules operations in time and modifies the structure of the circuit to preserve its functionality. This concept generalizes several ad hoc techniques such as precomputation and branch prediction that have been applied to synchronous circuits. We first describe the basic ideas behind architectural retiming and then give two examples of applying it to real circuits. Finally, we outline some of the issues that must be addressed to automate the architectural retiming process.

1 Introduction

In many applications the performance of a circuit dominates all other concerns. Even when performance is not of paramount importance, it can be leveraged to achieve other goals - it is a folk theorem that performance can be traded for almost anything else. For example, performance can be traded for cost via time-multiplexing and it has been shown that trading away excess performance by reducing the supply voltage is an easy way to reduce power consumption [5]. In this paper we present a new technique for increasing the performance of a synchronous circuit based on pipelining and retiming. This technique, called architectural retiming, can be used when neither pipelining nor retiming are applicable.

The performance of a synchronous circuit is measured using throughput, the rate at which the circuit completes computations, and execution time, the time it takes to perform a single computation from input to output[1]. If we assume that one computation completes each clock cycle, then the throughput of a circuit is $T = 1/T_c$ where T_c is the clock period. We will measure latency as N, the number of cycles required to complete one computation. We will refer to the time required to complete one computation as the execution time, $E = N \cdot T_c$. Improving the performance of a circuit usually means improving its throughput, even at the expense of increasing the latency. However, there are cases when latency is important and we would like to improve throughput without affecting the latency. Throughput is improved by simply reducing the clock period, which can be accomplished by many techniques including technology improvements, circuit techniques, logic optimizations, and algorithm improvements. We focus here on two general techniques for reducing the clock period, retiming and pipelining.

Retiming is a well-known technique [14] that minimizes the clock period by rescheduling the operations of the computation performed by a circuit, spreading them out optimally over the number of cycles available to the computation. This rescheduling is done by relocating the registers in the circuit while preserving the functional behavior of the circuit and its interface with the environment. Since retiming does not change the number of clock cycles taken by the computation it does not affect latency and actually decreases the execution time in proportion to the reduction in the clock period.

Retiming, while an important transformation, yields only modest improvements and only for a limited class of circuits [10]. In particular, retiming cannot improve the performance of circuits whose clock period is determined by a *critical cycle*, a cycle whose total delay divided by the number of registers equals the clock period. Critical cycles are

[1] Execution time is also referred to as latency. For clarity, however, we will reserve the term latency to refer to the number of clock cycles between the start and completion of a computation.

also known as max-ratio-cycles, which can be determined efficiently [12]. Note that in retiming, the critical cycle may be the cycle formed by connecting the outputs of the circuit to its inputs via a register in order to preserve the the relative timing and thus the latency of interface signals.

Pipelining is a technique that increases the number of clock cycles in which to perform the computation. Since more cycles are available, each cycle can be shorter, increasing the throughput. The latency is increased, but the execution time may increase or decrease depending on how much the clock period is reduced. The effect of pipelining can be optimized using retiming. An optimal procedure for pipelining a circuit is to add one register at a time to all inputs (or all outputs), using retiming to determine the optimal location for all registers.

Pipelining, too, cannot improve the performance of all circuits. Like retiming, it cannot improve the performance of circuits whose clock period is determined by a critical cycle. Unlike retiming, however, pipelining is allowed to add registers to the cycle formed by connecting the outputs to the inputs via a register. Indeed, this is the definition of pipelining. In some applications, however, pipelining by adding registers to this I/O cycle may not be permitted because of the increase in latency. Examples include memory systems and real-time interactive graphics.

Thus the application of both retiming and pipelining is prevented by the presence of a critical cycle. The only way to increase the performance of a circuit constrained by a critical cycle is to decrease the delay on the critical cycle or increase the number of registers. Assuming that all optimizations that decrease delay have been applied, the only remaining option is to increase the number of registers on the cycle without increasing the number clock periods around the cycle, a seeming contradiction.

In this paper we present a technique we call *architectural retiming* which attempts to do exactly this — increase the number of registers on a critical cycle or a path without increasing the perceived latency. We use the term architectural retiming because operations in the circuit are moved in time as in retiming but the structure of the circuit must be changed to preserve its functionality. More formally we can state the task of architectural retiming as follows:

Architectural Retiming

> Given a synchronous circuit whose clock period is constrained by some critical cycle, increase the number of registers on that cycle, thereby decreasing the clock period, while preserving the circuit's functionality and latency.

Note that the critical cycle may be the cycle formed by connecting the inputs to the outputs in order to preserve the interface timing.

We first examine the problems posed by architectural retiming and describe the basic ideas which are used to solve these problems. We then present two example real-life circuits that illustrate these concepts. Finally we discuss some of the issues that must be addressed in order to automate architectural retiming.

2 Overview of Architectural Retiming

Architectural retiming comprises two steps: First, a register is added to a critical cycle or path. Second, the circuit is changed to absorb the increased latency caused by the additional register. Before describing the architectural retiming process in detail, we examine the effect of adding a register to a critical cycle.

2.1 Pipelining Cycles in a Circuit

Figure 1 shows a simple cycle which is assumed to be the critical cycle in a circuit. Neither retiming nor pipelining can add registers to this cycle to reduce the clock period, but this is precisely what must be done. Unfortunately, adding the register (b) changes the behavior of the circuit from that shown in Table 1(a) to that shown in Table 1(b). If we think of the cycle as a pipeline feeding itself, then adding a register on the cycle creates *bubbles* in the pipeline (shown as dashes in Table 1(b)). Adding the register thus increases the latency from one clock period to two clock periods but throughput cannot be increased. Thus there is no advantage to adding a register to the cycle unless there is some way to maintain the latency of the original cycle.

Those familiar with c-slowing [14] will recognize that we have c-slowed the circuit by a factor of two and we can gain a performance advantage in a straightforward way if we can process two unrelated data sets in parallel. C-slowing, however, is a technique that can be applied only in very restricted circumstances.

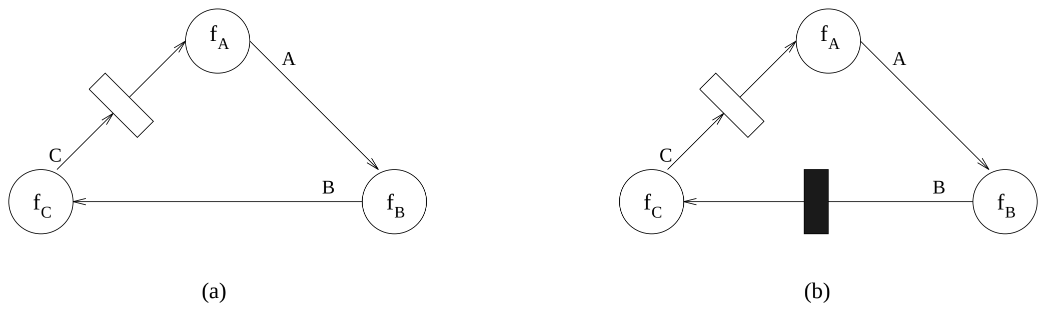


Figure 1: *(a) Original Circuit with simple critical cycle: the circles represent logic functions and the rectangles represent registers. (b) Modified Circuit: a register (shaded rectangle) is added.*

	$t-1$	t	$t+1$
A	a_0	a_1	a_2
B	b_0	b_1	b_2
C	c_0	c_1	c_2

(a)

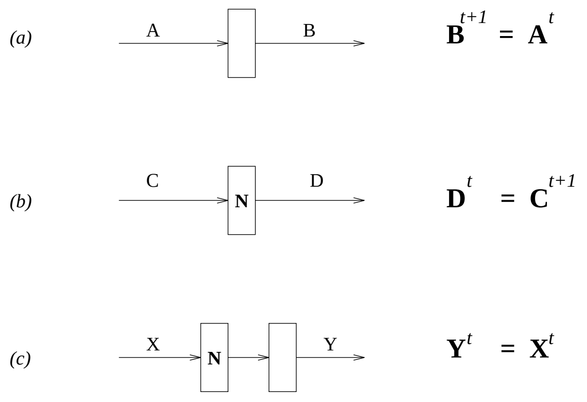
	$t-1$	t	$t+1$	$t+2$	$t+3$
A	-	a_1	-	a_2	-
B	-	b_1	-	b_2	-
C	c_0	-	c_1	-	c_2

(b)

Table 1: *(a) Schedule of the cyclic pipeline shown in Figure 1(a). The values a_i in each column labeled l is the value the variable A has in a specific cycle l. The arrows indicate the cyclic dependencies in the circuit. (b) Schedule of the circuit in Figure 1(b): Adding an extra register between f_B and f_C and preserving the functionality of the circuit introduces* bubbles *(marked as dashes).*

The challenge of architectural retiming then is to preserve the latency of the critical cycle and the functionality of the circuit while increasing the number of registers on a critical cycle. This is accomplished using the concept of a *negative register*. A normal register performs a shift forward in time as shown in Figure 2(a) while a negative register performs a shift backward in time (b). A negative register/normal register pair effectively cancel, reducing to a simple wire (c), which is our objective. The question then is how to implement a negative register.

Let us assume the input to the negative register is the result of the function $f(x_1, x_2, \ldots)$. The definition of the negative register requires that the output of the register at time t is the value of f at time $t+1$, that is $f(x_1^{t+1}, x_2^{t+1}, \ldots)$. The values $x_1^{t+1}, x_2^{t+1}, \ldots$ are, of course, not available at time t. There are two solutions to this problem.



(a)
A
B
$\mathbf{B}^{t+1} = \mathbf{A}^{t}$
(b)
C
N
D
$\mathbf{D}^{t} = \mathbf{C}^{t+1}$
(c)
X
N
Y
$\mathbf{Y}^{t} = \mathbf{X}^{t}$

Figure 2: *Normal and Negative registers: (a) Normal register: in cycle* $t+1$*, the variable B acquires the value of A in cycle t. (b) Negative Register: a shift backward in time. (c) A negative register/normal register pair: the combined effect of the register pair is a zero time shift between X and Y.*

	$t-1$	t	$t+1$	$t+2$	$t+3$
A		a_1	a_2	$\xcancel{a_3}$	a_3
B	b_0	b_1	b_2	$\xcancel{b_3}$	b_3
B^*	b_0^*	b_1^*	$\xcancel{b_2^*}$	b_2	b_3^*
C	c_0	c_1	$\xcancel{c_2}$	c_2	c_3
*Correct***?**		$\surd$	$\surd$	$\times$	$\surd$

Table 2: *Schedule for the cyclic pipeline in Figure 1(b) after applying prediction-based architectural retiming. $B*$ is the prediction. Verification of b_i^* against b_i occurs one cycle after the prediction.*

2.2 Precomputation

If the values of $x_1, x_2, \ldots$ at time $t+1$ can be computed directly from values available at time t then we can rewrite the function f as a function f' of values at time t. That is, $f(x_1^{t+1}, x_2^{t+1}, \ldots) = f'(g_1(y_1^t, y_2^t, \ldots), \ldots)$ where $x_1^{t+1} = g_1(y_1^t, y_2^t, \ldots), \ldots$. This can be done only if there is sufficient information in the circuit to precompute the value one cycle ahead of time, and the negative register is implemented as a function which is derived from the definition of the original circuit. It should be clear that by adding this implementation of a negative register in front of the normal register, the modified circuit of Figure 1 (b) is transformed to behave exactly like the original circuit (a).

The function f' that implements the negative register includes two clock cycles worth of computation along the critical cycle, which is precisely where optimizations and transformations are most needed. The question remains whether this function, f', can be computed efficiently to reduce the max-cycle-ratio. If not, then we will have to resort to the next solution. In Section 3.1 we apply precomputation-based architectural retiming to the design of a memory interface.

2.3 Prediction

In most cases, there will not be sufficient information in the circuit to allow precomputation and we must rely on some oracle to *predict* the value produced by the negative register. If we can predict perfectly, as in precomputation, then no other change needs to be made to the circuit. If we cannot, then the prediction will at times be incorrect and unless the circuit can adjust to this error, it will produce the wrong result. Dealing with incorrect predictions requires first verifying whether predictions are correct and then nullifying the effect of incorrect predictions.

A prediction is verified one cycle after it is calculated using the actual value when it finally arrives. If the prediction was correct, the next prediction is allowed to proceed. If the prediction was incorrect, some measure must be taken to nullify the effects of the incorrect value and to restore the circuit to its previous state. Once restored to its previous state, the circuit continues operation using the delayed correct value. Table 2 illustrates this process. The value of B cannot be precomputed and thus a prediction, B^*, is used instead. In the cycle after B^* is generated, the real value of B is available and a comparison between the predicted value and the true value takes place. If the prediction was correct (as for predictions b_0^* and b_1^* in the table) the execution resumes with the next prediction value. If the prediction is incorrect (as for prediction b_2^*) then the real value is inserted in the pipeline and the circuit recomputes based on the correct value. Thus an incorrect prediction incurs a one cycle penalty.

The actual process of nullifying the effect of an incorrect prediction can be quite complex as we describe in Section 4.

Branch prediction in processors is a simple example of architectural retiming using prediction. The branch condition cannot be calculated when the branch instruction issues. The branch condition is therefore predicted and the next instruction(s) based on the branch prediction is executed. If the prediction is incorrect, the state of the processor must not be updated (i.e. the write operation to the register file is blocked), and the instruction at the correct branch address is executed. If the prediction is correct then no corrective action is necessary and execution continues. In Section 3.2 we apply prediction-based architectural retiming to the design of a multicomputer network routing chip.

3 Examples of the Application of Architectural Retiming

In this section we present two examples of applying architectural retiming to circuits taken from real systems to improve throughput while preserving the system's latency requirements. In the first example, the system cannot be pipelined

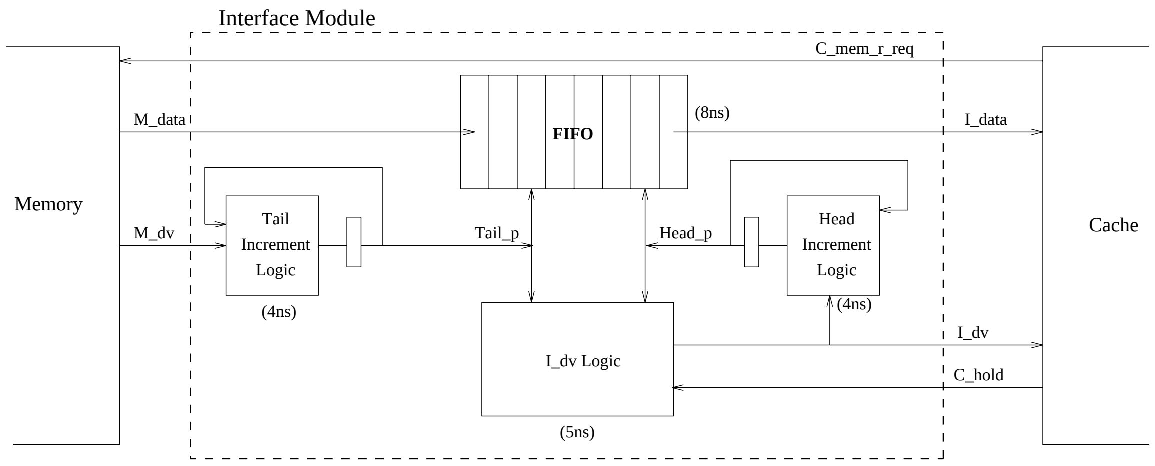


Figure 3: *Original implementation of the memory interface module. The numbers in parenthesis next to the logic blocks indicate the delay of the block's critical path. The system must run at $16ns$. The critical path ($20ns$) is due to reading the FIFO to generate the signal I_data ($8ns$) as I_data is required at the module's boundary $4ns$ after the edge of the clock.*

because of constraints on the latency from input to output. In the second example, the circuit cannot be improved by either retiming or pipelining because the clock period is constrained by a critical cycle. The first example is solved using precomputation-based architectural retiming and the second example is solved using prediction.

3.1 Example 1: Memory Interface Module

The first example is the memory system[2] shown in Figure 3. The cache provides a read request $C\_mem\_r\_req$ and an address (not shown) in cycle t, and expects a number of packets of valid data in return during the following cycles. When the memory receives a read request from the cache, it attempts to send the requested data. If the memory has the valid data, it asserts the M_dv signal and sends a packet of data. The cache can take a data packet if the signal C_hold is deasserted. If C_hold is asserted, then the Interface Module (IM) is responsible for buffering the data coming from memory and passing them to the cache once it deasserts C_hold. The IM consists of a FIFO memory, and logic to conditionally advance the head and tail of the FIFO and to set the data valid signal, I_dv.

The original behavior of the IM can be specified as follows:[3]

$$\begin{aligned}
FIFO^{t+1}[tail\_p^t] &= M\_data^t & (1)\\
I\_data^t &= FIFO^t[head\_p^t] & (2)\\
I\_dv^t &= (C\_hold^t \vee empty^t)?\ 0 : 1 & (3)\\
new\_tail^t &= M\_dv^t?\ tail\_p^t + 1 : tail\_p^t & (4)\\
tail\_p^{t+1} &= new\_tail^t & (5)\\
new\_head^t &= (I\_dv^t)?\ head\_p^t + 1 : head\_p^t & (6)\\
head\_p^{t+1} &= new\_head^t & (7)\\
empty^t &= (head\_p^t == tail\_p^t) & (8)
\end{aligned}$$

The system is required to run at a clock period of $16ns$. The data from memory to the cache is specified to have a one cycle latency when C_hold is deasserted. The FIFO read operation has a delay of $8ns$ after the triggering edge of the clock. The I_data signal is required to arrive at the IM boundary $4ns$ after the clock edge.

The critical path in the circuit is from the FIFO (a fixed register) to the primary output, I_data. Signal I_dv arrives $4ns$ later than its timing specification and the system is forced to run at $20ns$. Pipelining the I_data signal by inserting a register between the FIFO output and the cache solves this problem, but increases the latency of the interface module, violating the system latency constraints. Retiming cannot move the fixed register to reduce the clock period.

[2]This example was suggested by Ed Frank of NetPower.
[3]The add operations are implicitly modulo the FIFO size.

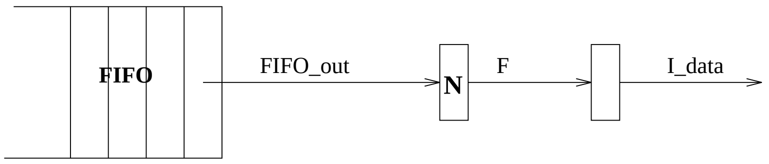


Figure 4: *A negative register/normal register pair added between the FIFO output and the cache.*

3.1.1 Applying Architectural Retiming

Since the clock period is constrained by the delay from the FIFO read to the cache, architectural retiming inserts a negative register/normal register pair between the FIFO output and the cache effectively pipelining this path without increasing the latency (Figure 4). The negative register is then implemented by synthesizing the logic necessary to precompute the output of the FIFO.

The logic that is needed to implement the negative register can be deduced from the original specification of the FIFO. By definition, the output of the negative register, F, at time t is $FIFO\_out$ at time $t+1$, but must be computed using only values available in time t.

From specifications (2), (3), (6), and (7),

$$\begin{aligned} FIFO\_out^{t+1} &= FIFO^{t+1}[head\_p^{t+1}] \\ &= (C\_hold^t \lor empty^t)? \\ &\quad FIFO^{t+1}[head\_p^t] : FIFO^{t+1}[head\_p^t + 1] \end{aligned}$$

We must refer to the state of the FIFO in cycle t. The FIFO is an array of elements indexed by $head\_p$ and $tail\_p$. According to specification (1), the FIFO state in cycle $t+1$ can differ from that in cycle t only in the location pointed to by the tail pointer $tail\_p^t$. Thus,

$$\begin{aligned} FIFO^{t+1}[head\_p^t] &= (head\_p^t == tail\_p^t)? \\ &\quad M\_data^t : FIFO^t[head\_p^t] \end{aligned}$$

and similarly,

$$\begin{aligned} FIFO^{t+1}[head\_p^t + 1] &= ((head\_p^t + 1) == tail\_p^t)? \\ &\quad M\_data^t : FIFO^t[head\_p^t + 1] \end{aligned}$$

Thus, using the three previous equations, we can compute the output of the negative register, F^t, as:

$$\begin{aligned} F^t &= FIFO\_out^{t+1} \\ &= empty^t ? \; M\_data^t : \\ &\quad (C\_hold^t ? \; FIFO^t[head\_p^t] : \\ &\quad (((head\_p^t + 1) == tail\_p^t)? \\ &\quad M\_data^t : FIFO^t[head\_p^t + 1])) \end{aligned}$$

The synthesis of the logic required to implement the negative register is possible since the value can be precomputed using values available at time t. The result is shown in Figure 5. The signal I_data is now available at the IM boundary $1ns$ after the clock (the delay in the register from the clock to the data is $1ns$), and it satisfies its timing requirement. None of the synthesized paths violate the specified clock period, and the architecturally-retimed circuit meets the latency specification. Note that architectural retiming has changed the architecture of the circuit. An extra register has been added along the critical path and a bypass path allows the data from the memory to the cache to maintain a one-cycle latency.

3.2 Chaos Router

The second example is taken from the chaos router, a two-dimensional, random, non-minimal adaptive packet router for implementing multicomputer interconnection networks [3]. Multi-flit packets from the network enter the router

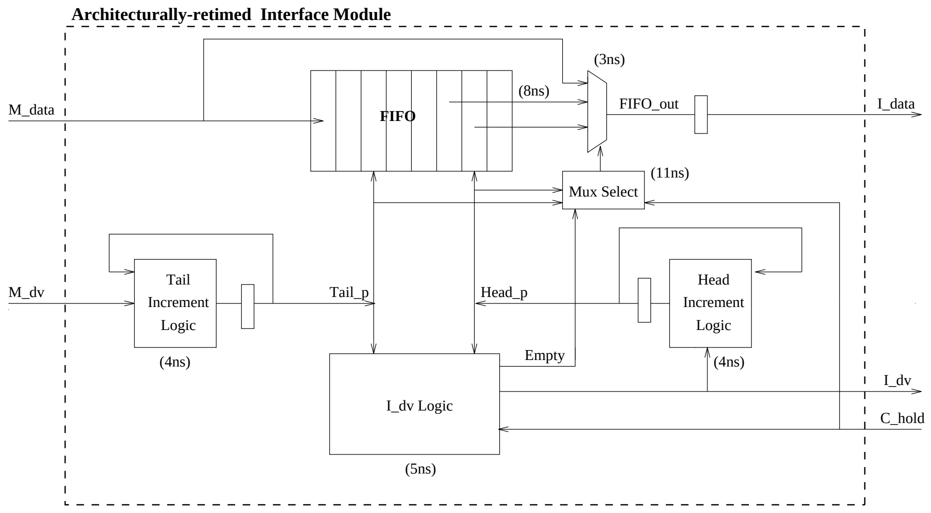


Figure 5: *Architecturally-retimed memory interface module. The implementation of the negative register results in generating a bypass path. None of the new synthesized paths violate the specified clock period ($16ns$). The latency and timing requirements for I_data are met.*

through input frames (buffers) and are routed to neighboring routers or the processor connected to the router through output frames. Incoming packets that cannot be routed immediately due to the unavailability of their requested output frames are temporarily buffered in the multiqueue, a modified FIFO buffer. When the multiqueue is full the chaos router routes only messages residing in the multiqueue.

One of the critical cycles in the chaos router's control logic is illustrated in Figure 6. The routing box attempts to make one new routing decision each clock cycle. A routing decision determines the next packet in an input frame or the multiqueue to be routed to an output frame. Packets in the multiqueue are given priority over packets in input frames. The multiqueue scoreboard manages the information for packets in the multiqueue, producing the signal Q_wants which indicates those output frames requested by multiqueue packets. The scoreboard uses the signal $Route\_from\_MQ$, which indicates which packet was chosen for routing, to eliminate requests by multiqueue packets that have been routed. The critical cycle in Figure 6 is through the routing box and the multiqueue scoreboard.

3.2.1 Applying Architectural Retiming

Architectural retiming adds a negative register/normal register pair on the critical cycle effectively pipelining the logic without adding latency. There are two possible locations where this register pair can be inserted: the $Route\_from\_MQ$ signal or the Q_wants signal. Since the information required to precompute either signal is not available, architectural retiming must use prediction. Choosing which signal to predict requires an analysis of the results of choosing each.

Predicting the signal $Route\_from\_MQ$, the identity of the packet chosen for routing, is difficult. It is not clear that we can do better than picking a value at random from those we know were going to be chosen, in which case we are seldom going to be able to predict correctly.

Predicting the value of Q_wants, the set of packets in the multiqueue requesting routing, is easier. The difference in this value from one cycle to the next comprises the addition of new requests, which are relatively infrequent, and the removal of the request just satisfied. Thus simply using the previous value of Q_wants for the prediction will be fairly accurate and the hardware cost associated with this prediction is minimal. Analyzing the effect of mispredicting this signal is somewhat complicated. The prediction could be incorrect in two cases.

The first case occurs when a packet requesting a specific output frame is chosen to be routed based on the old Q_wants and there are no other packets in the multiqueue that request the same output frame. This scenario can only occur immediately following a cycle in which a multiqueue-route operation had started. Since routing from the multiqueue is a multi-cycle operation, a new routing decision is not allowed until the multiqueue route is completed. Therefore, no actions are needed to nullify the effects of a misprediction.

The second case occurs when a new packet is routed into the multiqueue and the old Q_wants will not include the new packet's request for an output frame. The new packet then will miss the opportunity of being immediately

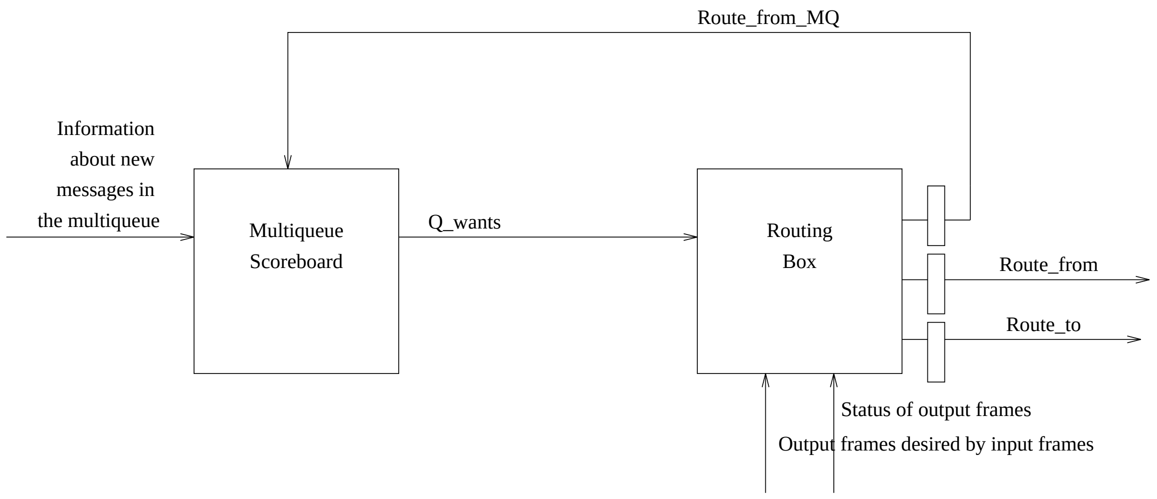


Figure 6: *Critical cycle in the control logic used to determine the routing decision in the chaos router.*

routed from the multiqueue. If the prediction is incorrect, then the routing decision must be canceled and the routing box must make a new decision based on correct information. Alternatively, the result of the misprediction can be allowed to proceed while the new packet resides in the multiqueue for a number of additional cycles until the packet gets the opportunity to be routed again. Although the latter solution is marginally unfair to the new packet, the router still functions correctly.

In Figure 7, we show how the critical cycle can be architecturally retimed when predicting Q_wants. The delay around the cycle is increased slightly by the propagation delay of the added register and the delay of the 2:1 multiplexer that selects between the predicted Q_wants^* and the delayed true Q_wants, but the cycle has been effectively pipelined, allowing a reduced clock period. Retiming can relocate the registers in the cycle to achieve the minimum clock period. Architectural retiming allows the router chip to run at a smaller clock period at the expense of a one clock cycle penalty when Q_wants is predicted incorrectly. Since the architecture of the router has been changed, the overall performance improvement of reducing the clock period at the expense of an occasional cycle penalty must be determined by simulating the entire system.

4 Issues in Architectural Retiming

Architectural retiming attempts to reduce the clock period for a given circuit by adding a negative register/normal register pair along the critical cycle. As described in Section 2, the negative register can be implemented as either a precomputation or prediction. The two examples in Section 3 demonstrated the effectiveness of architectural retiming in increasing the number of registers along the critical path without increasing its perceived latency.

The next step is to automate the architectural retiming process. Automation offers two advantages. First, it will allow designers to write simple behavioral models and delegate complex timing issues to an automatic tool, which speeds up the overall design cycle. Second, it will offer a fast way of exploring high-performance design alternatives. Because there is usually more than one position along critical paths where a normal register/negative register pair can be inserted, architectural retiming can generate several design alternatives that vary in cost and performance.

In this section we briefly outline some of the more important issues that must be addressed to successfully automate architectural retiming. Specifically, we discuss the choice of where to insert the negative/normal register pair, the techniques needed to effectively optimize the function created by architectural retiming, prediction strategies, and nullifying the effects of incorrect predictions.

4.1 Location of Negative Register/Normal Register Pair

Any point along the critical cycle is a potential insertion point for the register pair. At a specific insertion point the negative register may be implemented as either a precomputation or a prediction, while at others the implementation must

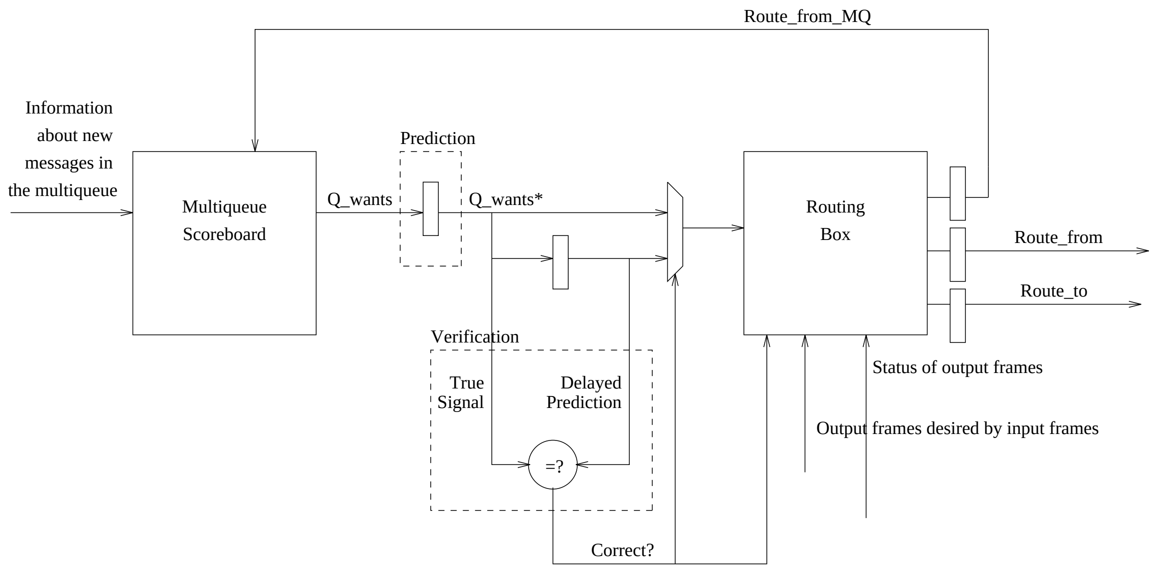


Figure 7: *Architecturally retimed : The prediction value for Q_wants^* is the old value of Q_wants. The verification occurs in the cycle after the prediction by comparing the delayed Q_wants^* with the pipelined Q_wants.*

be a prediction. Both precomputation and prediction result in synthesizing additional logic, however, unlike prediction, precomputation does not incur any cycle penalties. Each possible precomputation or prediction must be evaluated to determine the best point(s) to insert the negative register/normal register pair.

In the case of a precomputation, performance improvement can be easily evaluated. The transformed circuit's clock period can be determined once the architecturally-retimed circuit is synthesized and optimized. Performance evaluation is more difficult in the case of prediction due to the one-cycle penalty associated with misprediction. As was illustrated in Section 3.2, performance evaluation requires simulating the entire system to analyze the impact of the one-cycle penalty. The area penalty in both cases can be easily obtained after synthesis. For most circuits there will be no clear *best* architectural retiming solution. Instead, better performance will be achieved at the expense of added area and complexity. An automated architectural retiming tool must generate all possible solutions that meet user-defined performance and area constraints.

4.2 Concurrent Optimization of Arithmetic and Boolean Functions

Architectural retiming creates a new combinational function that implements the negative register. As was illustrated in Section 3.1, the new function is represented as a combination of Boolean and arithmetic functions. The function must be optimized to enable synthesizing a circuit with the shortest possible delay. Current optimization techniques that restructure critical paths are based either on optimizing Boolean functions [7, 2, 19, 11] or on exclusively manipulating arithmetic expressions [20, 15, 17, 6]. Optimizing arithmetic and Boolean functions concurrently yields better results than can be achieved by arithmetic optimization followed by Boolean optimization.

We present a small example to illustrate the benefits of concurrently optimizing Boolean and arithmetic functions. We assume that the function Y is created by architectural retiming, where X, Y, A, B, C, and D are *n*-bit variables, and $cond1$ and $cond2$ are single-bit variables. We wish to restructure Y to have the shortest delay from inputs A and B.

$$Y = (cond2?\ X :\ A)\ +\ D$$
$$X = (cond1?\ A :\ B)\ +\ C$$

Eliminating X from both equations and utilizing the identity $A + 0 + D = A + D$, Y can be rewritten as:

$$Y = ((cond1\ OR\ !cond2?)\ A :\ B)\ +\ (cond2?\ C :\ 0)\ +\ D$$

This optimization reduces the delay of the path from the inputs A and B to the output Y significantly. In the original circuit the delay is through a multiplexer, followed by a 2-input adder, followed by another multiplexer and then another

2-input adder. In the transformed description the delay of the path is through a multiplexer followed by two 2-input adder, or better yet, a 3-input adder.

4.3 Prediction Strategies

Several prediction strategies are possible when using prediction to implement the negative register. Probabilistic analysis of input signals and execution traces from a representative set of programs can be used in setting the prediction value. The above strategies will most likely yield better predictions than random ones, however, they may not yield the best prediction values. An alternative promising strategy is to generate prediction values based on understanding the circuit's behavior. One aspect of the circuit's behavior that can yield reasonable prediction values is how the behavior of signals varies from one cycle to the next. For example, in the chaos router in Section 3.2 understanding that the value of Q_wants does not change often from one cycle to the next yields a reasonable prediction value, the old value of Q_wants. The result is a prediction value that is *dynamic*, varying from one cycle to the next based on the old value of Q_wants. Understanding the implications of mispredictions can also drive the prediction strategy. Predicting the less likely value may significantly improve performance at a much smaller cost (area, power consumption) than that required by any other prediction value. Ultimately, the prediction value should be chosen to balance the performance penalty and the implementation costs of the negative register.

4.4 Nullifying the Effects of Mispredictions

A straightforward method of fixing the results of an erroneous prediction is to restore the state of the circuit and re-execute the computation using the correct delayed signal. Restoring the state of every register in the circuit is costly. Instead, architectural retiming must selectively restore the state of the circuit. One method of limiting the restoration extent is to allow the results of mispredictions to propagate in the circuit, *blocking* only specific registers from updating their state. This restoration method requires identifying *commit* points in the circuit. Typical commit points are registers that store values that are needed to calculate the next computation. For example, in a processor the register file is a commit point. In case of a branch misprediction, only the register file is blocked from updating its state. Another method of limiting the restoration extent is to ignore fixing the results of incorrect predictions when they cause *don't care* operations. Don't care operations are NOPs that propagate through the circuit without affecting any commit points.

Applying architectural retiming sometimes requires changing the interface of the circuit with its environment. Because an incorrect prediction might occur during some cycle, the circuit's primary outputs must be paired with validation signals to inform the interface when a primary output is incorrect. The primary inputs to the circuit must be provided again by the interface during the cycle when the circuit is executing the substitute (correct) computation. Therefore, a data-not-taken signal is required to notify the interface when primary inputs must be repeated. Alternatively, a stall signal may be added to block the interface from providing more inputs to the circuit while it is computing the substitute computation. The effect of applying architectural retiming to a circuit is to create an *elastic* interface which can provide (consume) input (output) data at varying rates and is implemented using handshaking mechanisms. A change in the circuit's interface may be undesirable or difficult to implement but is inevitable if performance is to be improved. Moreover, many circuits already have interface protocols which architectural retiming can take advantage of.

Allowing mispredictions to propagate through the circuit is an alternative to re-executing the correct operation and restoring the circuit to its previous state. In some cases allowing mispredictions to propagate causes a slight modification of the circuit's behavior. Sometimes this modification is acceptable and it does not cause a great performance penalty. For instance, in the chaos router (Section 3.2) using the old value of Q_wants as a prediction value results in ignoring incoming packets for one cycle. An incorrect prediction then may result in routing messages slightly out of order compared to the order performed by the original circuit. Such modification of the circuit behavior might be acceptable especially if the clock period of the overall system is reduced.

5 Related Work

Architectural retiming is an optimization technique that encompasses and overlaps with others. The term negative register was used in peripheral retiming [16], an optimization technique for sequential circuits. Negative registers were only allowed on peripheral edges which connect the circuit with it environment. Unlike architectural retiming's definition of negative registers which allows them to have different implementations, peripheral retiming specifically uses the term to signify borrowing registers from the environment.

Architectural retiming performs sequential logic optimization by exposing adjacent pipeline stages *along* critical paths for combinational optimization, an approach which is not used by current sequential optimization techniques. De Micheli applies local algebraic transformations across latch boundaries [8]. Peripheral retiming [16] allowed negative weights on peripheral edges to allow the whole circuit to be optimized using combinational optimization. The optimized circuit is then retimed using conventional retiming [14]. Dey et al. resorted to circuit partitioning to identify subcircuits with equal-weight reconvergent paths (petals) to which peripheral retiming can be applied [9]. Chakardhar et al. took a more timing-driven approach to sequential optimization that results in applying combinational optimization techniques to each stage in the circuit [4]. The technique identifies the least stringent set of arrival and required timing constraints which are passed to the combinational delay optimizer along with the circuit.

Recently, the research community has shown interest in various forms of pre-execution, speculation, and precomputation to achieve improvements in performance. Holtmann and Ernst describe four examples for which they explore applying a speculative technique that is modeled after multiple branch prediction in a processor [13]. Their methodology is to ignore control dependencies during scheduling and then add register sets to restore the state in case of prediction error. Their predictions are based on data gathered from a program profile. Their technique is applied to a program description and thus, it only predicts explicit `if-then` control points in the description.

Alidina et al. use precomputation to restructure circuits to consume less power [1]. They propose precomputing a stage's output values based on a subset of input conditions. The original stage is turned off in the following cycle to avoid any switching activity. The choice of the input conditions determines whether the precomputation logic will have less power dissipation than the original stage. Alidina et al. present a method for synthesizing the precomputation logic and selecting the input conditions. Radivojević et al. describe a scheduling technique that employs pre-execution [18]. 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.

6 Conclusion and Future Work

In this paper, we have presented the concept of architectural retiming as a way to organize and formalize some of the ad hoc ideas in architecture transformation by focusing on the bottleneck to performance improvements in sequential circuits. Pipelining critical cycles using negative/normal register pairs is an elegant formulation of the transformation required to speed up cycle-bound circuits. We have described two alternatives for implementing negative registers and presented two realistic examples where these transformations provide increased performance. We are currently developing an interactive architectural retiming tool, ART, which will provide a framework for investigating the algorithms required to apply architectural retiming.

References

[1] M. Alidina, J. Monteiro, S. Devadas, A. Ghosh, and M. Papaefthymiou. "Precomputation-Based Sequential Logic Optimization for Low Power". In *Proc. of the 1994 IEEE International Conf on CAD*, pages 74 –81, 1994.

[2] K. Barlett, W. Cohen, A. de Geus, and G. Hachtel. "Synthesis and Optimization of Multilevel Logic Under Timing Constraints". *IEEE Transactions on Computers*, CAD-5(4):582–595, October 1986.

[3] K. Bolding, S.-C. Cheung, S.-E. Choi, C. Ebeling, S. Hassoun, T. A. Ngo, and R. Wille. "The Chaos Router Chip: Design and Implementation of an Adaptive Router". In *Proceedings of VLSI '93*, Sept. 1993.

[4] S. T. Chakradhar, S. Dey, M. Potkonjak, and S. Rothweiler. "Sequential Circuit Delay Optimization Using Global Path Delays". In *Proc. 30th ACM-IEEE Design Automation Conf.*, pages 483 –489, 1993.

[5] A. Chandrakasan, S. Sheng, and R. Brodersen. "Low-Power CMOS Digital Design.". *IEEE Journal of Solid-State Circuits*, 27(4):473–84, April 1992.

[6] L. F. Chao. "Optimizing Cyclic Data-Flow Graphs via Associativity". In *Proceedings of the Fourth Great Lakes Symposium on VLSI*, pages 6 –10, 1994.

[7] J. Darringer, D. Brand, J. Gerbi, W. Joyner, and L. Trevillyan. "LSS: A System for Production Logic Synthesis". *IBM Journal of Research Development*, 28(5):326–8, September 1984.

[8] G. De Micheli. “Synchronous Logic Synthesis: Algorithms for Cycle-Time Minimization”. *IEEE Transactions on Computer-Aided Design*, 10(1):63–73, Jan. 1991.

[9] S. Dey, F. Brglez, and G. Kedem. “Partitioning Sequential Circuits for Logic Optimization.”. In *IEEE International Conference on Computer Design*, pages 70–6, 1991.

[10] C. Ebeling and B. Lockyear. “On the Performance of Level-Clocked Circuits”. In *Advanced Research in VLSI*, pages 242–356, 1995.

[11] J. Fishburn. “A Depth-Decreasing Heuristic for Combinational Logic; or How to Convert a Ripple-Carry Adder into a Carry-Lookahead Adder or Anything in-between.”. In *Proc. 31th ACM-IEEE Design Automation Conf.*, pages 361 –4, June 1990.

[12] 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.

[13] U. Holtmann and R. Ernst. “Experiments with Low-Level Speculative Computation Based on Multiple Branch Prediction”. *IEEE Transactions on VLSI Systems*, 1(3):262–267, September 1993.

[14] 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.

[15] F. Maciel, Y. Miyanaga, and K. Tochinai. “Optimizing and Scheduling DSP Programs for High Performance VLSI Designs.”. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Science*, E57-A(1):1191–201, 1992.

[16] S. Malik, E. M. Sentovich, R. K. Brayton, and A. Sangiovanni-Vincentelli. “Retiming and Resynthesis: Optimizing Sequential Networks with Combinational Techniques”. *IEEE Transactions on Computer-Aided Design*, 10(1):74–84, Jan. 1991.

[17] M. Potkonjak and J. Rabaey. “Maximally Fast and Arbitrarily Fast Implementation of Linear Computations”. In *Proc. of the 1992 IEEE International Conf on CAD*, pages 304 – 08, 1992.

[18] 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.

[19] K. Singh, A. Wang, R. Brayton, and A. Sangiovanni-Vincentelli. “Timing Optimization of Combinational Logic”. In *Proc. of the 1988 IEEE International Conf on CAD*, pages 282–5, Nov. 1988.

[20] H. Trickey. “Flamel: A High-Level Hardware Compiler”. *IEEE Transactions on Computer-Aided Design*, CAD-6(2):259 –269, March 1987.