Architectural Retiming:
Pipelining Latency-Constrained Circuits *

Soha Hassoun and Carl Ebeling
Department of Computer Science and Engineering
University of Washington, Seattle, WA 98195-2350
{soha,ebeling}@cs.washington.edu
http://www.cs.washington.edu/research/projects/lis/www/archretiming

Abstract − This paper presents a new optimization technique called *architectural retiming* which is able to improve the performance of many latency-constrained circuits. Architectural retiming achieves this by increasing the number of registers on the latency-constrained path while preserving the functionality and latency of the circuit. This is done using the concept of a negative register, which can be implemented using precomputation and prediction. We use the name architectural retiming since it both reschedules operations in time and modifies the structure of the circuit to preserve its functionality. We illustrate the use of architectural retiming on two realistic examples and present performance improvement results for a number of sample circuits.

1 The Problem

The performance of synchronous digital systems is measured using throughput, the rate at which computations complete, and execution time, the total time between the start and completion of one computation[1]. Running the system at the smallest possible clock period increases throughput, which is inversely proportional to the clock period, T_c, and reduces execution time, which is directly proportional to T_c. For the system to function correctly, however, T_c must be greater than or equal to the longest combinational delay between each pair of registers. Retiming [8] can be used to spread the registers optimally along all paths, finding the minimum feasible T_c. But for any path p with n pipeline stages, $T_c \times n$ must be greater than or equal to the delay of the longest path through the n stages.

When a retimed circuit fails to run at the target T_c because of a path whose delay exceeds $T_c \times n$, further pipelining must be used to increase the number of registers along p. With more clock periods available to complete each computation, the clock period can be shorter. Increased pipelin-

ing, however, is often not an option because of latency constraints which fix the number of clock cycles allowed for a computation and thus the number of registers on a path.

Some latency constraints ensure that the circuit meets external performance requirements. If the execution time and the clock period are specified, then only a fixed number of clock periods is available to perform the computation. Many systems have such performance restrictions. Examples include circuits in memory systems and real-time interactive graphics, where the clock period is set for the overall system and the maximum execution time is specified for each operation. Other latency constraints are caused by cycles in the circuit. Changing the number of registers on a cycle changes the functionality of the circuit; therefore, the latency of each cycle is fixed. External constraints can also 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.

Regardless of how the latency constraint is derived, the computation along a latency-constrained path is required to complete in a fixed number of clock cycles. Assuming that all optimizations that decrease delay have been applied, the only remaining option is to increase the number of registers on the path without increasing the number of clock periods to perform the computation, 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 latency-constrained path without increasing the 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 limited by a latency-constrained path, increase the number of registers on that path, thereby decreasing the clock period, while preserving the circuit's functionality and latency.

We begin this paper by reviewing the notation that will be needed to describe architectural retiming. We then describe the basic concepts. Next, we demonstrate the application of architectural retiming to two real-world circuit examples. Finally, we report the results of applying the technique to a set of examples, and discuss previous related work.

---

[1] Execution time is commonly referred to as latency. For clarity, however, the term latency will be strictly used to refer to the number of clock cycles between the start and completion of one computation.

2 Preliminaries

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 is a single-output function of N input variables (or signals) $x_0, x_1, \ldots, x_{N-1}$ computing a variable y. 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, $y^t = f(x_0^t, x_1^t, \ldots, x_{N-1}^t)$.

The set of *fan-in* signals of a function f is the set of f's input variables. The set of *combinational transitive fan-in* signals of a function f, denoted by CT_f, is defined recursively as follows. For each fan-in variable x_i of f, if x_i is a primary input, then $x_i \in CT_f$. If x_i is an output of a register, then $x_i \in CT_f$. Otherwise, g_i computes x_i, i.e. $x_i = g_i()$ and CT_{g_i} is in CT_f. The set of *sequential transitive fan-in* signals across one register boundary, ST_f^1, is defined the same as CT_f except that the input path is allowed to cross one register. Unfolding [10] is used to refer correctly to ST_f^1 for a single-register cycle.

3 Overview of Architectural Retiming

Architectural retiming comprises two steps. First, a register is added to the latency-constrained path. Second, the circuit is changed to absorb the increased latency caused by the additional register.

The challenge of architectural retiming then is to preserve the latency of the path and the functionality of the circuit while increasing the number of registers. This is accomplished using the concept of a *negative register*. A normal register performs a shift forward in time, while a negative register performs a shift backward in time. That is, the output of a negative register is defined by $z^t = y^{t+1}$. A negative register/normal register pair reduces to a simple wire, 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 variable y computed by the function $f(x_0, x_1, \ldots, x_{N-1})$. From the definition of the negative register, $z^t = y^{t+1} = f(x_0^{t+1}, x_1^{t+1}, \ldots, x_{N-1}^{t+1})$. The values x_i^{t+1} for $0 \leq i \leq N-1$ are, of course, not available at time t. There are two possible ways to compute each x_i^{t+1}: precomputation or prediction. The choice is made after examining CT_{x_i}, the combinational transitive fan-in set for x_i, for each x_i in the fan-in of f.

3.1 Precomputation

If the value of $x_i, 0 \leq i \leq N-1$, at time $t+1$ can be computed directly from values available at time t, then we can recompute the function f as a function f' of values at time t. That is, $f(\ldots, x_i^{t+1}, \ldots) = f'(\ldots, g_i(\ldots, y_{i_j}^t, \ldots), \ldots)$, where $x_i^{t+1} = g_i(\ldots, y_{i_j}^t, \ldots)$ for $0 \leq i \leq N-1$ and $0 \leq j \leq M-1$, where M is the cardinality of $ST_{x_i}^1$. The function f' which implements the negative register can then be derived from the definition of the original circuit by the recursive collapsing (or flattening) of each x_i into the variables in $ST_{x_i}^1$ or the primary inputs in CT_{x_i} for which f is a don't

care value. The recursive collapsing across a register boundary consists of substituting the input variable to the register for its output variable.

Precomputation can be done only if there is sufficient information in the circuit to allow *precomputing* the value one cycle ahead of time. Precomputation is possible under the following condition:

$$\forall_{p \in PI}, \ ((p \in CT_{x_i}) \ \Rightarrow \ (f(\ldots, g_i(\ldots, p, \ldots), \ldots) = X))$$

where PI is the set of primary inputs, and X is a don't care value for the function f.

The function f', which is synthesized by the precomputation, includes two clock cycles' worth of computation along the latency-constrained path, which is precisely where optimizations and transformations are most needed. The implementation of f' and its impact on reducing the clock period will vary depending on the specific circuit under consideration. The example presented in Section 4.1 illustrates how precomputation generates a bypass, an architectural transformation that can significantly improve performance.

3.2 Prediction

When the condition for performing precomputation is not met, then we must resort to an alternative solution to evaluate the output of the negative register, z^t. We can rely on an oracle to *predict* the value produced by the negative register (i.e. one cycle before the value is actually computed). 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 they are correct and then nullifying the effect of incorrect predictions.

A prediction is verified one cycle after it is calculated once the actual value is computed. 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. Thus, a one cycle penalty is associated with a misprediction The actual process of nullifying the effect of an incorrect prediction can be quite complex as we illustrate in the example in Section 4.2.

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. The effect of applying architectural retiming is to create an *elastic* interface, which provides (consumes) input (output) data at varying rates. Elastic interfaces are implemented by adding handshaking mechanisms to the circuit. 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.

The probability of a correct guess can be increased by utilizing partial information that can be precomputed. Each
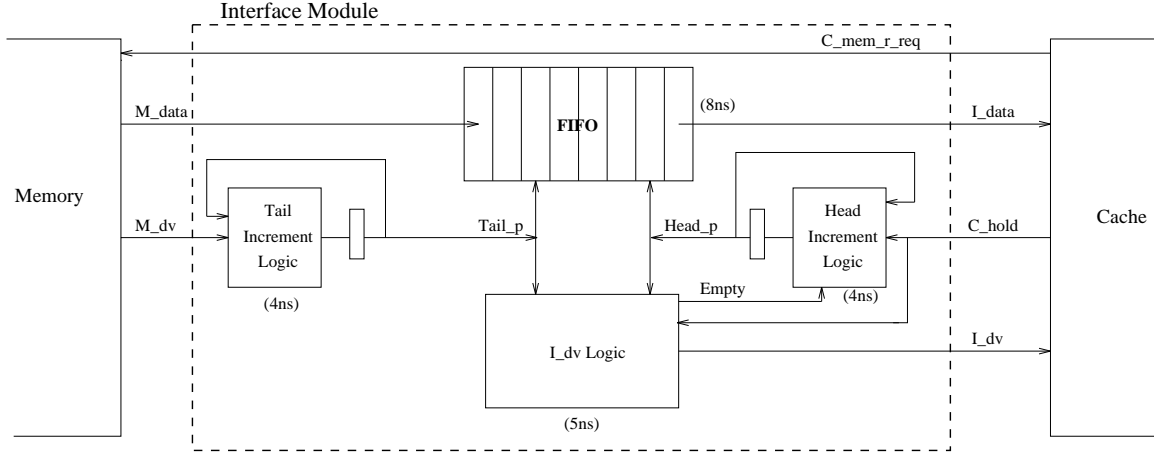
Figure 1: *Original implementation of the memory interface module. The numbers in parentheses next to the logic blocks indicate the delay of the block's longest combinational path. The system must run at 16ns. The latency-constrained path is from the Memory to the Cache. I_data arrives at the interface 4ns later than the specification, forcing the system's clock cycle to be stretched by 4ns in order to function correctly.*

input x_i, for $0 \leq i \leq N - 1$, of the function f that computes the output of a negative register can be precomputed if $CT_{x_i} \cap PI = \emptyset$; otherwise, x_i^{t+1} or the signal(s) in the combinational transitive fan-in set for x_i that are also primary inputs must be predicted. When choosing a guess value for a signal, the frequency of mispredictions and the penalties associated with correcting the results of mispredictions must be evaluated to determine its impact on the overall performance of the cycle.

4 Examples

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 because of constraints on the latency from input to output. In the second example, the circuit is constrained by a cycle. The first example is solved using precomputation-based architectural retiming and the second example is solved using prediction.

4.1 Example 1: Memory Interface Module

The first example is the memory system[2] shown in Figure 1. 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 is specified as follows:[3]

$$FIFO^{t+1}[tail\_p^t] = M\_data^t \qquad (1)$$

---

[2]This example was suggested by Ed Frank of NetPower.

[3]The add operations are implicitly modulo the FIFO size.

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

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 to the primary output, I_data. Signal I_dv arrives $4ns$ later than its timing specification and the clock cycle is forced to be $4ns$ longer, or $20ns$. Pipelining the I_data signal by inserting a register between the FIFO output and the cache solves this problem, but violates the latency constraint. Retiming which only relocates registers in the circuit cannot reduce the clock period.

4.1.1 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 2). The negative register is
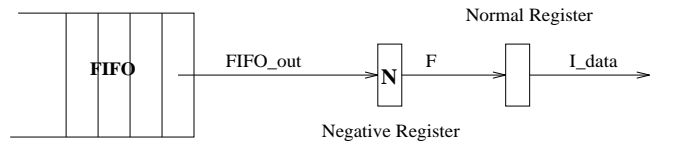

Figure 2: *A negative register/normal register pair added between the FIFO output and the cache.*
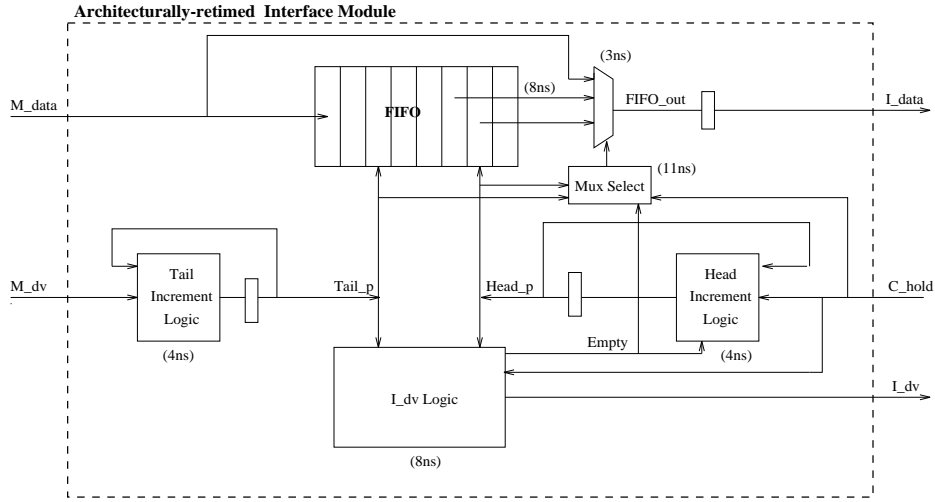
Architecturally-retimed Interface Module

Figure 3: *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.*

then implemented by synthesizing the logic necessary to pre-compute the output of the FIFO.

The logic 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),

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

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 3. 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 the 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.

4.2 Example 2: The 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 [2]. Multi-flit packets from the network enter the router 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 cycles in the chaos router's control logic is illustrated in Figure 4. 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 signal $Route\_from\_MQ$, which indicates which packet was chosen for routing, is used by the scoreboard to eliminate requests by multiqueue packets that have been routed. The chaos router is forced to run at a slower clock period than desired because of the long delay through the multiqueue scoreboard and the routing box.
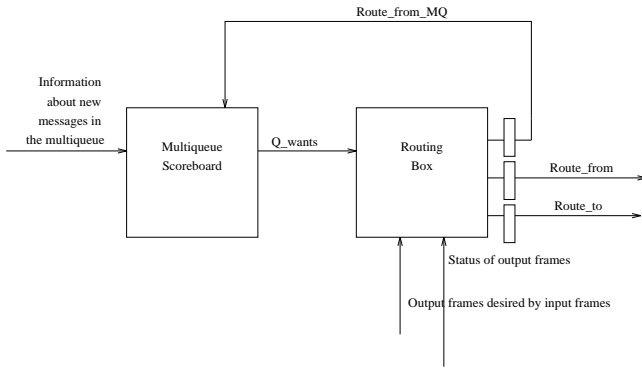
Figure 4: *The cycle in the chaos router that limits performance is through the routing box and the scoreboard.*

4.2.1 Applying Architectural Retiming

Architectural retiming adds a negative register/normal register pair on the 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 $Route\_from\_MQ$, which is 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, which is 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 will then miss the opportunity of being immediately 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.
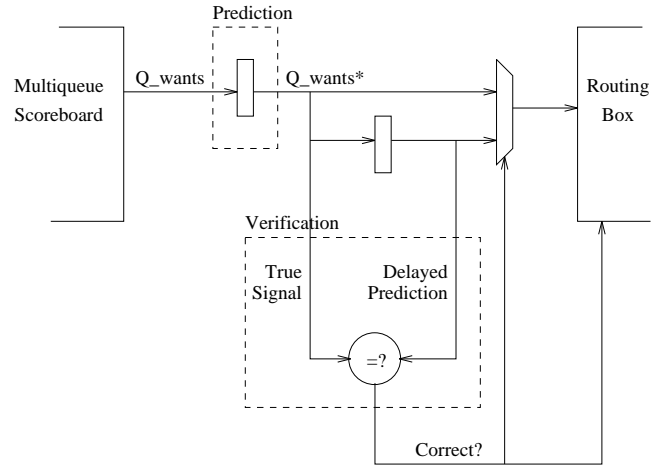

Figure 5: *Architecturally-retimed circuit: The prediction value for Q_wants^* is the old value of Q_wants. The verification occurs in the cycle after the prediction by comparing the delayed Q_wants^* with the pipelined Q_wants.*

In Figure 5, we show how the 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.

5 Experiments

Architectural retiming was applied to a number of circuits and the results are shown in Figure 6. The examples chosen are real circuits (with the exception of 3N+1) that have a latency-constrained path. Examples MIM and MDV are based on the interface module presented in Section 4.1. Example QC is a queue controller circuit. Example SEQ is a small two-stage sequential circuit that cannot be further optimized by current sequential optimization techniques [9,4,3].

The original cycle in examples FA2 and FA3 is a single-cycle circuit that performs a fetch from a RAM and add operation each cycle. FA2 and FA3 refer to a different implementation of the architectural retiming solution. Circuit 3N+1 is a single-register cycle that architectural retiming unfolds. The circuit calculates the function: if N^t is even, $N^{t+1} = N^t/2$, else $N^{t+1} = N^t \times 2 + 1$.

The numbers reported are based on applying architectural retiming and performing behavioral optimizations by hand. SIS [11] was then used to optimize the resulting circuits. Two optimizing scripts were applied to each circuit: script.delay and the sequence of the command full_simplify followed by the two scripts script.rugged and script.delay. We chose the optimized circuit that had the smallest delay even at the expense of added area.
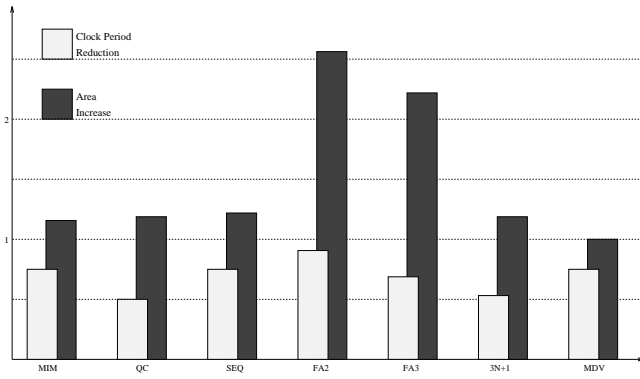
Figure 6: *Applying architectural retiming: Clock Period reduction vs. area increase.*

The average clock period reduction obtained using architectural retiming is 31%. The average area increase is 14% not including the unfolded circuits (FA2, FA3, 3N+1), and 99% for the unfolded circuits. At first glance the area penalty might seem excessive; however, this area increase is calculated with respect to the area of the latency-constrained path or cycle being architecturally retimed, which is typically only a small fraction of the entire circuit.

6   Related Work

Architectural retiming is an optimization technique that encompasses and overlaps other techniques and concepts. The term negative register was used in peripheral retiming [9], an optimization technique for sequential circuits. Unlike architectural retiming, which actually implements negative registers, peripheral retiming uses negative registers as a bookkeeping technique to keep track of the number of registers borrowed from the environment. Negative registers were also used in Ruby [6] to skew data while composing a circuit design.

Precomputation-based architectural retiming performs sequential logic optimization of latency-constrained paths by exposing adjacent pipeline stages for combinational optimization, which is not performed by current sequential optimization techniques. De Micheli applies local algebraic transformations across latch boundaries [4]. Peripheral retiming [9] moved registers from the interior of a circuit to its environment to allow the whole circuit to be optimized using combinational optimization. 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 [3]. This technique identifies the least stringent set of arrival and required timing constraints which are passed to a combinational delay optimizer along with the circuit.

Kogge discusses the problem of pipelining circuits with feedback [7]. He solves the problem of transforming a recurrence equation $x(n)$ that originally depends on the previous sequence, $x(n-1)$ to a recurrence equation that depends on an earlier recurrence. Kogge's technique conceptually unfolds the recurrence to allow the corresponding cyclic pipeline to complete one operation each cycle. Unfolding of iterative DSP data-flow graphs is also used in multiprocessor scheduling to expose the graph's hidden concurrency to allow the scheduler to achieve the smallest possible iteration bound [10].

Holtmann and Ernst present a scheduling algorithm that applies a speculative technique that is modeled after multiple branch prediction in a processor [5]. Precomputation has been used by Alidina et al. to restructure circuits to consume less power [1].

7   Conclusion and Future Work

We have found architectural retiming to be an elegant and powerful formulation that promises to generalize and automatically generate a number of ad hoc sequential optimization techniques that address the problem of improving the performance of latency-constrained circuits. In this paper we have presented the basic ideas of architectural retiming and shown how it can be used to speed up two realistic circuits. Our initial results for a set of example circuits show that architectural retiming can significantly improve performance. We are currently developing an interactive architectural retiming tool, ART, which will provide a framework for developing and refining 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. 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.

[3] 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.

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

[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] G. Jones and M. Sheeran. "Circuit Design in Ruby". In *IFIP WG 10.5 Lecture Notes*, pages 13–70, 1990.

[7] P. Kogge. *"The Architecture of Pipelined Computers"*. McGraw-Hill, 1981.

[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] 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.

[10] K. Parhi and D. Messerschmitt. "Static Rate-Optimal Scheduling Of Iterative Data-Flow Programs Via Optimum Unfolding.". *IEEE Transactions on Computers*, 40(2):178–95, February 1991.

[11] E. Sentovich, K. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. Stephan, R. Brayton, and A. Sangiovanni-Vincentelli. "SIS: A System for Sequential Circuit Synthesis". Technical Report UCB/ERL M92/41, University of California, Dept. of Electrical Engineering and Computer Science, May 1992.