Fine Grain Incremental Rescheduling Via Architectural Retiming

Soha Hassoun

Tufts University, Medford, MA 02155

soha@eecs.tufts.edu

Abstract

With the decreasing feature sizes during VLSI fabrication and the dominance of interconnect delay over that of gates, control logic and wiring no longer have a negligible impact on delay and area. The need thus arises for developing techniques and tools to redesign incrementally to eliminate performance bottlenecks. Such a redesign effort corresponds to incrementally modifying an existing schedule obtained via high-level synthesis. In this paper we demonstrate that applying architectural retiming, a technique for pipelining latency-constrained circuits, results in incrementally modifying an existing schedule. Architectural retiming reschedules fine grain operations (ones that have a delay equal to or less than one clock cycle) to occur in earlier time steps, while modifying the design to preserve its correctness.

1 Introduction

High-Level Synthesis (HLS), which translates a behavioral description to a register transfer level (RTL) description, comprises scheduling and binding. Scheduling determines the start time of operations corresponding to the functions in the behavioral description. Latency and/or resource constraints shape the resulting schedule. Binding assigns operations to resources. Scheduling and binding, thus, synthesize an architecture with defined temporal and spatial properties. The resulting design is a register-transfer, abstract structural implementation.

With the decreasing feature sizes during VLSI fabrication and the dominance of interconnect delay over that of gates, control logic and wiring no longer have a negligible impact on delay and area. The need thus arises for (a) better delay and area models during HLS, and (b) the mechanism for relaying, perhaps dynamically, that information back to the scheduler. Alternatively, a schedule can be *incrementally* modified once detailed timing and area information is available – thus modifying the existing schedule only when and where necessary.

Incremental rescheduling is the process of modifying an existing schedule if the initial schedule does not meet its stated initial goals. Incremental rescheduling is appropriate once physical pipelining registers, steering logic, wiring delays, and control logic are appropriately estimated. Only minor changes are made because the goal is to minimally impact the existing schedule while respecting the precedence and latency constraints used during initial scheduling. Incremental rescheduling can be considered as a step in percolation scheduling [2], which refines an initial schedule in a stepwise fashion in as allowed by resource and timing constraints. However, incremental scheduling is applied much later in the design cycle to a structural description of the circuit.

The term *fine grain operation*, or operation, in this paper refers to one that executes during one or less clock cycle. In fine grain rescheduling, part of a one-cycle operation can be rescheduled, i.e. moved to earlier or later time steps.

Fine grain incremental rescheduling is equivalent to resynthesizing the circuit portion limiting the performance. Architectural retiming [4] is such an optimization technique. Given an RTL design, architectural retiming removes the performance bottleneck(s) by rescheduling the operations along the critical paths between primary inputs and outputs, or the critical cycles (paths posing the lowest iterative bound). We refer to both types of bottlenecks as latency-constrained paths because no pipelining is allowed to further reduce the lowest iteration bound.

We investigate in this paper the relationship between architectural retiming and fine grain incremental rescheduling. Architectural retiming reschedules operations either with unlimited resource constraints or with limited additional recourses. Without any resource constraints, architectural retiming reschedules an operation earlier in time (the previous pipeline stage), and synthesizes circuitry to perform the rescheduled operation quicker than in the original schedule. It thus changes the dependencies of an operation to an earlier time step than that specified in the original schedule. This results in performing a *precomputation* while allocating the needed additional resources. Without additional resources, architectural retiming pipelines a signal and speculatively schedules all its dependent operations. The only additional resources needed are the ones used to verify the correctness of the speculative operations and to recover after erroneous computations. This form of architectural retiming is referred to as *prediction*. Our goal in this paper is to show that architectural retiming is a form of fine grain rescheduling capable of incrementally modifying an existing schedule.

We begin this paper by reviewing our model and architectural retiming. Next, we discuss precomputation and prediction. For each, we describe the resulting schedule changes, we present an example, and we compare our technique with other high-level synthesis and scheduling approaches. We conclude by summarizing the contributions of this paper.

2 Model

To describe our algorithms, we use a simple form of control/data flow graph (CDFG) [2]. Our structural representation is described based on the following assumptions. 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 fine grain operation in the design is described using a single-output function, f, 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, $y^t = f(x_0^t, x_1^t, \ldots, x_{N-1}^t)$. Each function may be composed of Boolean and mathematical operations.

To describe a computation over time, we use a table (or a schedule). For clarity, arrows showing dependencies are annotated with function names to denote the dependencies between signals. The schedule for the circuit path in Figure 1(a) is shown in Figure 2(a).

3   Architectural Retiming: A Review

Architectural retiming pipelines a latency-constrained path while preserving a circuit's latency and functionality. Latency constraints arise frequently in practice, and they are due to either cyclic dependencies or explicit performance constraints. Architectural retiming is comprised of 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. To preserve the circuit's latency and functionality, we use the *negative register* concept. A normal register performs a shift forward in time, while a negative register performs a shift backward in time. That is, the output of the negative register is computed one cycle before the actual arrival of its input signal. If z is the negative register's output, and y is the input, then $z^t = y^{t+1}$. A negative register cancels the effect of the added pipeline register; the register pair reduces to a wire. Resulting performance improvements are due to increasing the number of pipelining stages, and thus clock cycles available for the computation. This allows for a smaller clock and improved performance.

Two implementations of the negative register are possible: precomputation and prediction. In precomputation, the negative register is synthesized as a function that precomputes the input to the added pipeline register using signals from the previous pipeline stage. In prediction, the negative register's output is predicted one clock cycle before the arrival of its input. The predicting negative register is synthesized as a finite state machine capable of predicting new values, determining mispredictions, and correcting mispredictions. More details about architectural retiming and its use in both logic and structural architectural synthesis can be found in [3].

The next two sections investigate how precomputation and prediction relate to fine grain incremental rescheduling.

4   Incremental Rescheduling Without Resource Constraints

4.1   Precomputation

When implementing the negative register added by architectural retiming as a precomputation, we effectively syn-

thesizes a function f' that precomputes the input to the added pipeline register based on the inputs of the previous pipeline stage. Precomputation is illustrated along a path, p, in Figure 1. The output of the negative register is computed as follows:

$$
\begin{aligned}
z^t &= y^{t+1} \\
&= f(x^{t+1}) = f(g(m^t)) \\
&= f'(m)
\end{aligned}
$$

The function f' is the composition of functions f and g and evaluates based on the inputs of the previous pipeline stage. If f' can be computed faster than the original composition, then the total delay along the critical path is reduced, and the new path can be substituted for the critical one (note that some nodes along the critical path are retained if needed by other nodes in the circuit). Retiming can then optimally place the registers in the circuit. Precomputation thus exposes the concurrency in two adjacent pipeline stages. Note that precomputation is not possible if the inputs to the current pipeline stage are unavailable earlier in time (for example, they constitute one of the circuit's primary inputs).

We examine the impact of precomputation on the schedule (before any retiming). Although the pipeline register delays a signal by one clock cycle, its input is computed one clock cycle earlier. Thus, the precomputed signal (the input to the added pipeline register) is rescheduled one clock cycle earlier in time. Moreover, the new function f' computes this input based on signals in the same time step which are inputs to that pipeline stage. The original and modified schedules for two adjacent time steps t and $t+1$ for path p are illustrated in Figure 2.
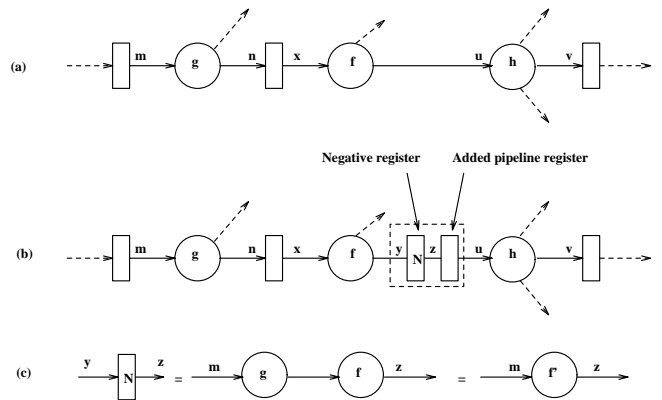

Figure 1: Path p. (a) Original path. (b) Architecturally retimed path: a negative register is added followed by a pipeline register. (c) Precomputation function f' implements.

Precomputation-based architectural retiming was shown to be effective in improving performance [5]. In addition, precomputation results in synthesizing two interesting architectural transformations: bypassing and lookahead.

Bypassing (forwarding) is an architectural optimization technique often used in processor design to reduce the latencies associated with writing and then reading the same location in a register file [12]. Bypassing is also applied in other
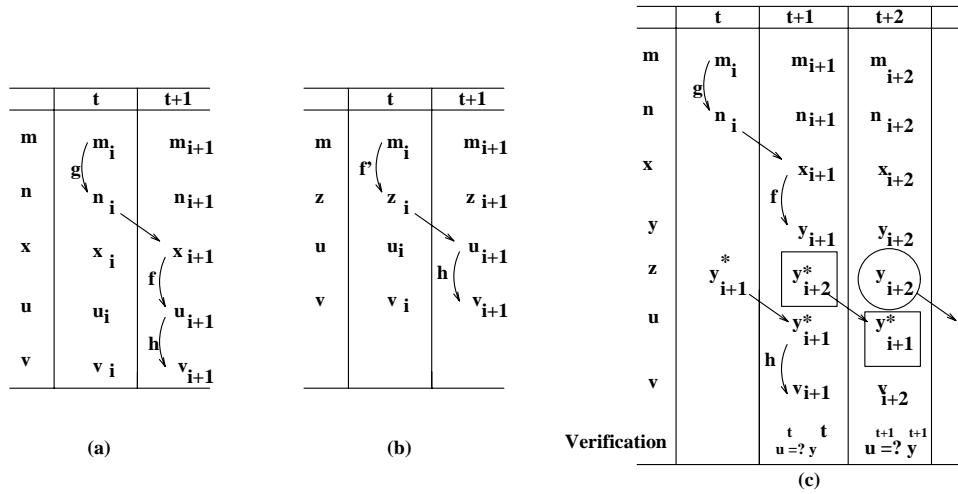
Figure 2: Schedules for path p in Figure 1. The labeled arrows indicate the function that computes the signal at the head of the arrow. Arrows crossing a time line indicate the presence of a pipeline register. (a) Original schedule. (b) Schedule after precomputation. (c) Schedule after prediction. The value y_{i+1}^* at the output of the negative register, z, in clock cycle t is a prediction. It is propagated to u in cycle $t+1$, and it is also verified against the true signal y_{i+1}. Assuming it is correct, the negative register predicts again, the value y_{i+2}^*. Assuming it is incorrect, in clock cycle $t+2$, signal z is assigned the correct value. Two clock cycles are then used to produce a correct value in case of a misprediction.

domains to hide register array latencies. By precomputing the output of an array of registers – representing a RAM, ROM, FIFO, Register file – results in synthesizing a bypass transformation and the necessary control logic.

When precomputing in single-register cycles, architectural retiming results in pipelining this cycle. This is similar to synthesizing a lookahead transformation on a recursive data flow graph used in high-level synthesis. The following example illustrates this transformation.

4.2 Precomputation Example: GCD

Consider the CDFG for GCD as shown in Figure 3. The corresponding RTL implementation is shown in Figure 4[1]. The reset circuitry and logic to compute *done* are not drawn for clarity. The critical cycle posing the lowest iteration bound involves computing $x > y$ and setting up the multiplexors.

We apply precomputation-based architectural retiming. We add a negative register followed by a pipeline register to pipeline the result of the comparison. The output of the negative register will be labeled m. The input will be labeled n. Using the definition of the negative register, we compute m first in terms of signals from time $t+1$, and then re-express these in terms of signals available at time t. Thus,

$$m^t = n^{t+1} = (x^{t+1} > y^{t+1})$$

We can evaluate x and y in terms of signals available in an earlier iteration, that is:

$$x^{t+1} > y^{t+1} = comp^t?(x^t - y^t > y^t) : (x^t > y^t - x^t)$$

---

[1] A faster implementation of GCD is to concurrently compute $x - y$ and $y - x$ and use the most significant bit of one of the results to select appropriately the new value. We choose the slower implementation to demonstrate how both precomputation and prediction can be applied to a simple example.

Signal $comp^t$ is the result of comparing x^t and y^t. A more efficient implementation of $x^{t+1} > y^{t+1}$ can be obtained by replacing $((x^t - y^t) > y^t)$ with the most significant bit of the operation $(2 \times y^t - x^t)$. Similarly, $(x^t > y^t - x^t)$ can be replaced by the most significant bit of $(y^t - 2 \times x^t)$.

Assuming a k bit wide datapath, we can therefore compute m^{t+1} as follows:

$$m^t = comp^t?(2 \times y^t - x^t)[k-1] : (y^t - 2 \times x^t)[k-1]$$

The optimized circuit is shown in Figure 6. The original and modified schedules for GCD is shown in Figure 5(a) and (b), respectively. Precomputation reschedules the fine grain operation used to compute *comp* to an earlier time step. Combined with the logic in the previous time step, a new function is synthesized to compute m. The value of signal *comp* is thus precomputed one clock cycle earlier than in the original schedule. Using SIS [14] to optimize and retime the circuit, we computed a speed up of 35% at an area increase of 63%, which is result of the increase in logic area by 49% and registers by 256%. Additional performance improvements are possible by reapplying precomputation to precompute the signal *new*.

4.3 Related Work

Precomputation overlaps with several existing techniques in high-level synthesis. The effect of using lookahead to pipeline cycles (loops with feedback) was previously discussed by Kogge [8]. He transforms a recurrence equation $x(n)$ that originally depends on the previous sequence, $x(n-1)$, and an external input $a(n)$, to a recurrence equation that depends on an earlier recurrence – $x(n-i)$, $a(n)$, and $b(i)$ – a collection of terms provided as inputs to the circuit in the previous i cycles. Kogge's technique then con-

	t	t+1
x	x_i	x_{i+1}
y	y_i	y_{i+1}
comp	$comp_i$	$comp_{i+1}$
new	new_i	new_{i+1}

(a)

	t	t+1
x	x_i	x_{i+1}
y	y_i	y_{i+1}
m	m_i	m_{i+1}
comp	$comp_i$	$comp_{i+1}$
new	new_i	new_{i+1}

(b)

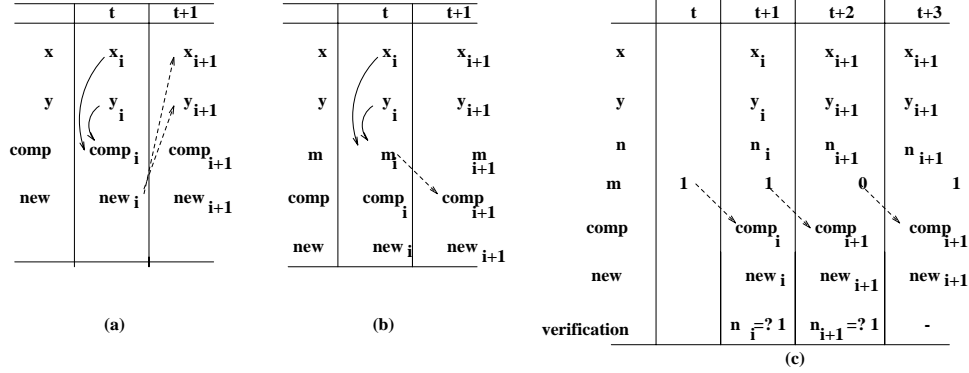|	t	t+1	t+2	t+3
x		x_i	x_{i+1}	x_{i+1}
y		y_i	y_{i+1}	y_{i+1}
n		n_i	n_{i+1}	n_{i+1}
m	1	1	0	1
comp		$comp_i$	$comp_{i+1}$	$comp_{i+1}$
new		new_i	new_{i+1}	new_{i+1}
verification		$n_i =? 1$	$n_{i+1} =? 1$	-

(c)

Figure 5: Execution of GCD. (a) Initial schedule: new is computed based on x and y available in the same clock cycle. (b) Precomputation schedule: *comp* is computed based on signals from the previous iteration. Note that the values of x and y from the previous pipeline stage are effectively x and y of the previous iteration. (c) Prediction schedule. The output of the negative register is predicted one clock cycle before the input arrives.
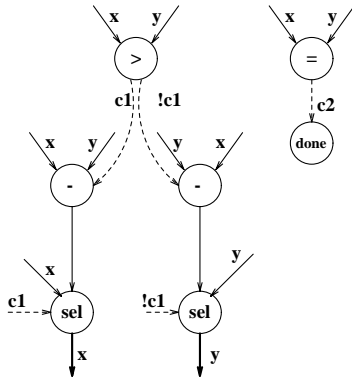

Figure 3: Control Data Flow Graph for GCD. The dashed arrows are conditions. The solid arrows are dependencies. The thick arrows at the bottom indicate crossing an iteration boundary.

Figure 4: Initial implementation for GCD. Every clock cycle, either x or y is updated with the value of *new*. When architectural retiming is applied, the two registers inside the dotted box are added.

ceptually unrolls the recurrence i times to allow the corresponding cyclic pipeline to complete one operation each cycle. Lookahead of iterative DSP data-flow graphs is also used in optimizing quantizer loops to achieve the smallest possible iteration bound [11, 10, 1].

Precomputation-based architectural retiming, however, is distinct in its approach for several reasons. First, it performs rescheduling and resynthesis at a fine grain level. Second, it applies to any path or "piece of code", and not only to recurrence equations. Third, precomputation manipulates functions (operations) that have both Boolean and mathematical identities. Thus, control and data can be concurrently synthesized and optimized, leading to more interesting optimizations when compared to using techniques that specifically target each domain separately (for example, tree height reduction and constant propagation are used specifically to optimize data flow graphs). Fourth, when applied to the output of a register array, precomputation discovers a bypass transformation – an important technique whose synthesis has not been previously addressed. Finally, by allowing several pairs of added pipeline and negative regis-
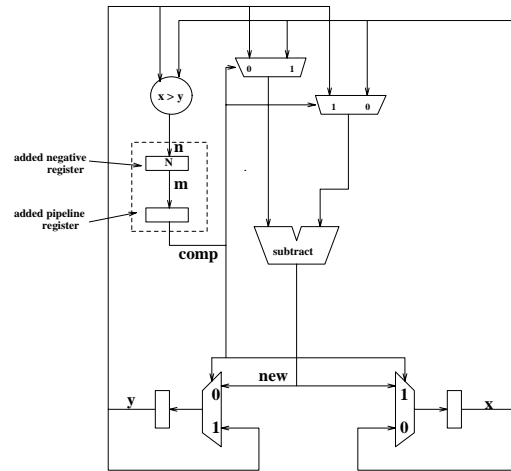
ters, the definition of precomputation can be easily modified to extend precomputation across several clock cycles. Architectural retiming's definition of precomputation is thus flexible and general.

5 Incremental Rescheduling With Resource Constraints

5.1 Prediction

Prediction-based architectural retiming delays a signal while speculatively executing all its dependent operations. The delayed signal is verified one clock cycle later. Execution continues if the prediction is correct; otherwise, correction is needed. See Figure 2(c) for a detailed schedule for the
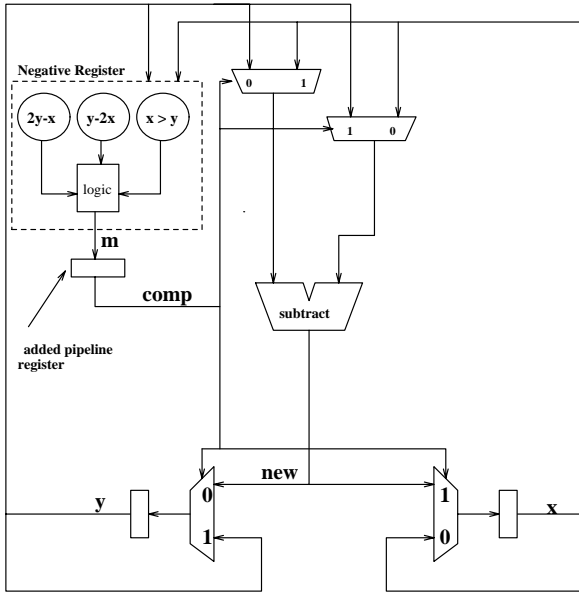
Figure 6: GCD implementation using precomputation. The critical cycle has roughly the same delay as the original circuit, but there are now two pipeline registers along that cycle, thus lowering the iteration bound.

path in Figure 1.

There are three issues in synthesizing prediction. First, the circuit must be modified to return to correct normal operation after a misprediction. A misprediction produces invalid (incorrect) data. This invalid data may propagate in the circuit affecting the next iterations of the computation. In our work, we developed two correction mechanisms, *As-Soon-As-Possible restoration* and *As-Late-As-Possible correction*. The first forces the circuit to return to normal operation by restoring the circuit's state to that of the previous clock cycle and re-executing the operations considering the correct, yet delayed, signal. The second mechanism, allows invalid data to propagate freely in the circuit, while making each node responsible for generating only one piece of invalid data for every misprediction in the circuit. The implementation of these two correction strategies generate varying and interesting architectures.

The second issue in synthesizing prediction is changing the circuit's I/O interface. Because two clock cycles are required to generate a new value in case of a misprediction, prediction-based architectural retiming produces a variable latency circuit. That is, the rate of consuming inputs and producing outputs is no longer a constant. To make the interface circuitry aware of this, handshaking signals that acknowledge consuming new inputs and signal invalid output data must be added to the circuit.

The final issue is deciding on a value to predict in order to minimize the frequency of mispredictions. In our work, we synthesize the negative register as a predicting finite state machine that uses transition probabilities provided by the designer to guide the predictions. We next examine how to apply prediction to the GCD example.

5.2   Prediction Example: GCD

We applied prediction-based architectural retiming to the GCD architecture shown in Figure 4 by adding the pipeline and negative register to signal *comp*. To compensate for the added pipeline register, operations dependent on *comp* are effectively rescheduled to execute speculatively. Assuming an equal likelihood of the signal being true or false, we randomly choose to predict it always true.

The final resynthesized circuit using the *As-Soon-As-Possible* correction is shown in Figure 7. A finite state machine either generates a new prediction (m is set to true) or corrects a previous guess (m is set to false) every clock cycle. If a misprediction is not detected in a specific clock cycle, then m is predicted to be true. The circuit will compute the operation $x - y$ in the following clock cycle. A misprediction is detected if n is false while m was predicted true in the previous cycle. In this case, the two added multiplexors allow the circuit to return to correct execution in the cycle following a misprediction, in which case the circuit computes $y - x$. The circuit's interface need not be modified because signal *done* is modified to never assert in the clock cycle during which misprediction is detected. Using SIS to evaluate the synthesized circuit, the speed up was 17% assuming a 50% prediction accuracy at an area increase of 92%.

The modified schedule is shown in Figure 5(c). In clock cycle $t + 2$, a misprediction is detected if the value of n_{i+1} is false. Signal m, the output of the FSM, is set false. The misprediction signal is asserted and the added multiplexors select the old values for x and y. In clock cycle $t + 3$, the circuit executes the operation $y - x$ based on previous values of x and y.
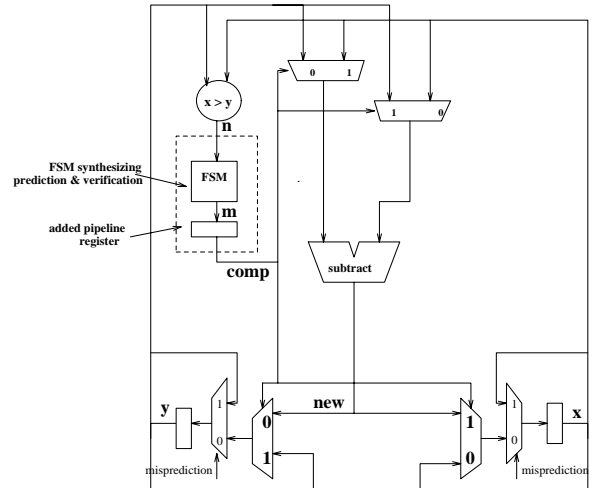

Figure 7: Using ASAP restoration correction strategy when applying prediction-based architectural retiming to restore the state of the circuit in case of a misprediction.

5.3   Related Work

Speculative execution can be classified according to its scope and nesting level. The scope (breadth) can span all possible execution paths, some of them, or it can be along a single

path. Speculating along one or two execution paths could be reasonable both in software and in hardware; however, speculating along multiple paths results in large hardware implementations. The speculative execution's nesting level (depth or extent) varies: allowing either a single outstanding speculation during active execution, or multiple ones. Speculation lasts only until a condition is resolved. Thus, in practice, the nesting level is limited.

Using speculative approaches in non-processor domains have been recently investigated by the high-level synthesis (HLS) community. Scheduling algorithms consider a variety of scoping and nesting levels.

Wakabayashi and Tanaka propose a scheduling algorithm to parallelize multiple nests of conditional branches; however, they only don't consider loop dependencies [15]. Holtmann and Ernst describe four examples for which they explore applying a technique modeled on multiple branch prediction in a processor [6]. Their methodology is to ignore control dependencies during scheduling and then add register sets to restore the state in case of prediction error. ASAP restoration associated with prediction is similar to their correction mechanisms, however, since architectural retiming is based on optimizing the RTL description, knowledge of exact circuit topology and connectivity allows more flexibility. For example, in ASAP restoration, architectural retiming does not restore the state of register files, but rather synthesizes bypass circuitry to delay updating register arrays instead of restoring them in case of misprediction.

Holtmann and Ernst's technique is applied to a program description and, thus, it predicts only explicit if-then control points in the description. In their follow-up work, they present a detailed scheduling algorithm [7] to be used in high-level synthesis. The scheduling algorithm performs speculation along the most probable path. Radivojević et al. describe a scheduling technique that employs pre-execution [13]. 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. Lakshminarayana et al. recently described how to incorporate speculative execution in a generic schedule [9]. They allow speculative execution along multiple paths, and arbitrarily deep into nested branches.

Prediction-based architectural retiming is different than these high-level scheduling approaches because it could be applied along any signal along the critical path if the misprediction frequencies are minimal (i.e. when clever predictions are possible). This is in contrast to related work that just speculates based on conditionals. By examining speculation at the structural level, we consider changes necessary to deal with the variable latency. Our two correction strategies explore two different methodologies for using additional multiplexors and registers to return a circuit to correct operation after mispredictions. Finally, prediction is not limited to one nesting level. By inserting multiple pairs of negative/pipeline registers we can effectively increase the nesting level − that is, we could allow multiple outstanding mispredictions.

6 Conclusion

This paper has two main contributions. First, it demonstrates that architectural retiming is capable of performing fine grain incremental rescheduling of operations along crit-

ical paths. Incremental rescheduling will become more important as deep submicron issues become more prevalent. Incremental rescheduling, and thus RTL optimizations, will become a middle ground between HLS and logic synthesis. Second, it explores in detail the relationship between architectural retiming and related work in high-level scheduling and synthesis. Synthesis algorithms that implement architectural retiming can be found in [3].

The techniques synthesized by architectural retiming are not new; however, architectural retiming uses the conceptual pipelining with negative and pipeline registers to unify and generalize a few techniques such as some forms of speculative execution, lookahead and bypassing. The approach proposed is refreshing and offers a novel way of performing incremental scheduling of fine grain operations.

REFERENCES

[1] F. Catthoor, W. Geurts, and H. D. Man. "Loop Transformation Methodology for Fixed-rate Video Image and Telecom Processing Applications". In *Proceedings of the International Conference of Application-Specific Processors*, pages 427–38, 1994.

[2] G. De Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, 1994.

[3] S. Hassoun. *"Architectural Retiming: A Technique for Optimizing Latency-Cosntrained Circuits"*. PhD thesis, University of Washington, December 1997.

[4] S. Hassoun and C. Ebeling. "Architectural Retiming: Pipelining Latency-Constrained Circuits". In *Proc. of ACM-IEEE Design Automation Conf.*, pages 708–13, June 1996.

[5] S. Hassoun and C. Ebeling. "Using Precomputation in Architecture and Logic Resynthesis". In *Proc. of the 1998 International Conference on Computer-Aided Design*, 1998.

[6] 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.

[7] 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.

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

[9] G. Lakshminarayana, A. Raghunathan, and N. Jha. "Incorporating Speculative Exeuction into Scheduling of Control-Flow Intensive Behavioral Descriptions". *Proc. ACM-IEEE Design Automation Conf.*, pages 108–13, June 1998.

[10] K. Parhi. "Algorithm Transformation Techniques for Concurrent Processors". *Proceedings of the IEEE*, 77(12):1879 − 95, December 1989.

[11] K. Parhi. "Look-ahead in Dynamic Programming and Quantizer Loops". In *IEEE International Symposium on Circuits and Systems*, pages 1382–7, 1989.

[12] D. Patterson and J. Hennessy. *"Computer Architecture : A Quantitative Approach"*. Morgan Kaufmann Publishers, 1990.

[13] 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.

[14] 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.

[15] K. Wakabayashi and H. Tanaka. "Global Scheduling Independent of Control Dependencies Based on Condition Vectors". *Proc. ACM-IEEE Design Automation Conf.*, pages 112–5, June 1992.