Performance-Constrained Pipelining of Software Loops onto Reconfigurable Hardware

Greg Snider
Hewlett-Packard Laboratories
1501 Page Mill Road
Palo Alto, CA 94304 U.S.A.
greg_snider@hp.com

ABSTRACT
Retiming and slowdown are algorithms that can be used to pipeline synchronous circuits. Iterative modulo scheduling is an algorithm for software pipelining in the presence of resource constraints. Integrating the best features of both yields a pipelining algorithm, retimed modulo scheduling, that can more effectively exploit the idiosyncrasies of reconfigurable hardware. It also fits naturally into a design space exploration process to trade-off speed for power, energy or area.

1. PROBLEM
We wish to map software loops, such as C-language “for” and “while” loops, onto a hardware target, subject to various performance constraints such as throughput, power, energy or area. For many applications, a throughput constraint of “as fast as possible” is desired; if heat production is an issue, “as fast as possible, but no more than this much power” might be appropriate; mobile applications may require “at least this fast, with minimal energy expenditure” to conserve battery life. We therefore generally want to be able to accelerate loops, but be able to trade-off the degree of acceleration with power, energy and/or area constraints.

One simple strategy for accelerating loops is pipelining: overlapping successive iterations of the loop in time so that the overall execution time of the loop is reduced. Pipelining requires solving two subproblems:

- Allocation—assigning (and sharing when necessary) limited resources to implement the computation.
- Scheduling—orchestrating the resources in time and space to efficiently implement the overlapped iterations without interference between different parts of the computation.

If the two subproblems can be decoupled, pipelining can be implemented easily by first doing the allocation, then the scheduling. More often, though, the problems are coupled and must be solved at the same time. This coupling arises in sequential processors from the fixed number of registers and function units that must be timeshared in order to implement the computation. This is less of an issue in reconfigurable (or custom) hardware since the number of function units and registers is usually not restricted ahead of time. Still, allocation issues arise in FPGAs when using limited resources (such as read- and write-ports on internal RAMs) and will likely increase as FPGA manufacturers continue to add more specialized function units, such as pipelined multipliers, to their chips.

Leiserson’s retiming algorithm [1] can be adapted to pipeline synchronous circuits, and can be easily modified to trade-off speed and power, but suffers from three limitations:

- It assumes all functional components have already been allocated and so does not solve the allocation subproblem.
- It cannot handle components that contain internal pipeline stages, such as synchronous RAM or pipelined multipliers, which are common in reconfigurable hardware.
- It fails to preserve the initial state of the retimed circuit, requiring difficult transformations to compensate (and which are not guaranteed to work) [2, 3] and/or circuit modifications that can degrade the circuit’s speed [4].

Software pipelining algorithms, such as iterative modulo scheduling [5], can pipeline loops on sequential processors. Such algorithms address the allocation subproblem and can handle pipelined components, but use a simple integral propagation delay model that can lead to excessive registers and power dissipation when applied to hardware synthesis. Nor do they provide any mechanisms for trading off speed and power to the degree we desire.

We therefore seek a pipelining algorithm that handles both the allocation and scheduling subproblems, works with pipelined and unpipelined components with non-integer propagation delays, preserves the initial state of the computation, and allow tradeoffs between execution time, power, energy and area.

2. BACKGROUND

2.1 Retiming and Slowdown

Retiming is a transformation on synchronous circuits that moves registers within a circuit while preserving the circuit’s behavior with respect to its primary inputs and outputs. In particular, the latency of the circuit is preserved. Leiserson’s algorithm OPT2 [1] uses retiming to minimize the clock period of a synchronous circuit, but the basic algorithm can be adapted to address many other optimization problems such as area, performance or power.

An example of retiming is shown in Figure 1. If an operation has registers on all of its inputs, those registers can be moved to the output of the operation without changing its functionality. Operation “a” meets this requirement, so its input registers “r1” and “r2” can be moved to the position “r4.” Similarly, if the output of an operation drives a register (and nothing else), that register may be
moved backwards to the inputs. In this example, “r3” on the output of operation “d” may be moved back to create registers “r5” and “r6.” If all operations have a propagation delay of 1, the top circuit has a minimum clock period of 2 (which can be seen from the path (r1, a, d, r3)), while the bottom circuit has a minimum clock period of 1.

If the latency of the circuit is unimportant, which is usually the case for loops executing a large number of iterations, the basic OPT2 algorithm can be slightly modified to create a pipelining algorithm. The modification simply involves adding an “infinite” number of registers at the primary outputs, running the OPT2 algorithm, and then discarding unnecessary added registers (Figure 2).

Feedback loops, or cycles, within the circuit are usually the limiting factor in pipelining. Leiserson proved that the number of registers around a cycle cannot be changed without changing either the circuit’s basic structure or behavior. The “average weight” of a cycle is defined to be the sum of the propagation delays through all operations on that cycle, divided by the number of registers on the cycle. Papaefthymiou proved that the clock period of a retimed circuit could not be less than the maximum average-weight of any cycle within the original circuit [6].

One problem with retiming not addressed by Leiserson’s and Saxe’s original paper is that of preserving the initial state of the circuit [2, 3, 4]. For some applications, initial state is not an issue, either because the algorithm is not sensitive to it, or because users are willing to accept a transient difference in behavior at start-up in exchange for better performance. If the initial state is an integral part of the circuit’s behavior, though, the retimed circuit must be modified to present an equivalent initial state.

Not all registers in a circuit are likely to require initial values; such registers can be moved forward or backward without bother. Registers with initial values can be moved forward by simulating the operations through which the registers pass and adjusting the initial values of the retimed registers accordingly. Moving registers with initial values backwards, though, requires backward justification [7] which is NP-hard and may fail. If that occurs, the only recourse is to modify the circuit. The initial state computation problem is illustrated in Figure 3.

Suppose that you have a circuit where all operations have the same unit propagation delay, and that, for some reason, you require a clock period equal to that propagation delay. In other words, from a register output to a register input, there is only enough time to execute a single operation. Most conventional processors behave in just that manner: each cycle several registers are read, several operations use the outputs of those registers as inputs, and at the end of the cycle the results of the operations are written to registers. But if the original circuit contains a cycle with an average weight greater than one, it would appear that achieving this objective through retiming would be impossible since the minimum clock period cannot be less than the average weight cycle.

This problem can be circumvented (at a price) through a synchronous circuit transformation called slowdown [8]. In slowdown, each register in the original circuit is replaced by a sequence of c
registers, producing what is known as a \textit{c-slow} circuit. The resulting circuit is then retimed to distribute the registers and minimize the clock period (Figure 4). As long as the maximum average-weight cycle of the \textit{c-slow} circuit is less than or equal to 1, the retimed, \textit{c-slow} circuit will be able to execute with the desired clock period of 1. The penalty of slowdown, through, is suggested by its very name: if the original circuit were able to accept input values on every cycle, the retimed, \textit{c-slow} circuit would only be able to accept inputs and produce outputs every \textit{c} cycles. This means that, for a pipelined circuit, only \textit{1/c} stages of the pipeline contain valid data at any given time—the rest contain “garbage.” So even though the \textit{c-slow} circuit can run at a higher frequency, this could be largely or completely offset by the reduced rate of input processing.

When mapping a loop onto a custom circuit, it might seem that slowdown has limited benefits, since it requires a higher clock speed and power consumption, with limited improvement in performance for a single input data stream, depending on the maximum average-cycle weight. There are at least two scenarios where slowdown is useful, though: (1) when the input consists of \textit{c} unrelated input data streams, those data streams can be interleaved cycle-by-cycle on a \textit{c-slow} circuit [9]; (2) when a single iteration of the loop requires two instances of the same operation (such as a multiply) and there is only one available function unit that can perform it (a multiplier), that function unit can be timeshared between the two operations by scheduling the operations in different stages modulo \textit{c}. Leiserson’s retiming algorithm, though, does not automate such sharing.

When mapping a loop onto a conventional processor, slowdown not only provides the timesharing potential needed for the scarce function units, it also allows a computation with a cycle-weight greater than one to be mapped onto the processor at all. Slowdown is the foundation for compiling loops onto conventional processors.

For sparse circuits, Leiserson’s original algorithm is \(O(N^2 \log N)\), where \(N\) is the number of operations in the circuit. Papaefthymiou was able to reduce this to \(O(N^2)\) for circuits containing function units with arbitrary propagation delays, and to \(O(N^{1.5})\) for circuits containing function units with unit delays [6]. Shenoy and Rudell have proposed changes to speed up the basic algorithm [10], as have Pan and Chen for the case of latency preservation [11].

2.2 Iterative Modulo Scheduling

Modulo scheduling is a framework for defining algorithms to pipeline innermost loops onto sequential processors. Iterative modulo scheduling (IMS) is an instance of such an algorithm that schedules a loop such that resource conflicts between overlapping iterations are resolved [5]. Additionally, each iteration has an identical schedule, and successive iterations are staggered in time by a constant interval known as the initiation interval (II).

IMS is functionally similar to retiming. Its assumptions about the target are more restrictive than retiming, reflecting the nature of the conventional processor that it targets:

- The clock period is assumed to be fixed.
- Every operation requires an integral number of clock periods in order to produce its result.
- The results of operations are written to (and read from) registers on timeslot boundaries.

IMS implicitly uses a slowdown transformation to solve both the cycle weight problem and the resource sharing problem. The maximum average cycle weight is referred to in IMS as the recurrence-constrained minimum initiation interval, or recMII. In the absence of resource interference, a loop may be scheduled with an \textit{initiation interval} (often called the II) equal to the recMII. This corresponds to a \textit{c-slowdown} transformation, with \textit{c} equal to II, with the same drawbacks as in retiming: inputs are accepted only every II cycles, with only 1/II stages of the pipeline performing useful computation at any given time (although the second drawback is masked to a large degree by aggressive sharing of resources).

Resource contention can make an initiation interval equal to recMII unachievable. Sharing issues lead to the concept of a resource-constrained minimum initiation interval, or resMII, which is determined by comparing the loop’s operations with the resources available for implementing them. For example, a single resource can be timeshared by II operations, as long as the timeslot modulo II for any one of the operations differs from each of the others (this modulo sharing constraint is the origin of modulo scheduling’s name).

IMS avoids the initial value problem of retiming. Conceptually, IMS pipelines by adding registers through slicing, rather than by moving registers. However, IMS’s lack of register motion (and the accompanying justification) can lead to inferior schedules when compared to retiming (see Figure 5). This can be compensated in some cases by using retiming as a preprocessing step to IMS.

The IMS algorithm begins with a dependence graph representing the loop computation. The graph is analyzed to determine recMII and resMII. The maximum of the two, called MII, is used as the initial value for II. An empty schedule consisting of discrete timeslots is created. Operations in the dependence graph are prioritized and inserted into a priority queue. Operations are then extracted from the priority queue one at a time and inserted into the schedule in the earliest possible timeslot such that: (1) input values needed by the operation have already been computed; and (2) the operation does not conflict with the modulo constraint of any other operation using the same resource. If no such timeslot exists, the operation is scheduled anyway, with conflicting operations unscheduled and returned to the priority queue. If all operations are successfully scheduled within an acceptable computation bound, the algorithm terminates successfully, otherwise the II is incremented and the scheduling repeated. Rau has published a detailed description of the algorithm [5].

\[\text{Figure 4. Slowdown and retiming. In the original circuit (a), each operation (white rectangle) has a delay of 1, and the minimum clock period determined by the registers (black rectangles) is 2. After a 2-slowdown transformation (b) followed by a forward retiming (c) the minimum clock period has been reduced to 1.}\]
3. OVERVIEW OF APPROACH

The basic pipelining algorithm described in this paper (called retimed modulo scheduling) is wrapped in an outer loop that explores the pipeline design space, looking for a pipeline implementation that best meets the given constraints. This is necessary since it is not easy to predict the performance of a pipeline without actually implementing it. However, for the “as fast as possible” constraint, this outer loop is short-circuited and the fastest possible pipeline is generated without exploration.

The loop to be accelerated is assumed to have been optimized and transformed to expose a suitable amount of parallelism and to minimize communication (using, for example, unimodular transformations [12], tiling [13], architectural retiming [14], etc.). The loop is then preprocessed with the following steps:

- Scalar output- and anti-dependences are removed by conversion to static single assignment (SSA) form. Array flow-dependence annotations are generated using dependence analysis [15].
- Control flow is converted to data flow with predicates using if-conversion [16] (Figure 6).
- An unpipelined circuit is then synthesized for the target platform. Generally, predicates in the original loop representation are replaced with register enables and write port enables in the circuit; each array is typically replaced with its own RAM. Array dependence annotations are attached to the corresponding read- and write-ports in the circuit.

At this point, the circuit is ready for retimed modulo scheduling, consisting of the following steps:

- The circuit is analyzed to determine lower bounds on the clock period and on the initiation interval. The initial clock period is set to the computed lower bound multiplied by a dilation factor (greater than or equal to 1.0) passed in as a parameter by the outer loop.
- Scheduling and allocation is attempted using the given initiation interval and clock period. If this fails, the clock period is increased and scheduling is reattempted. If repeated increases in the clock period still fail to result in a successful schedule, the clock period is reduced to its initial value, the II is incremented, and the process is repeated. This continues until scheduling succeeds.
- The scheduled circuit is compacted in an attempt to reduce the number of pipeline stages. The motivation here is that fewer stages will require fewer pipeline registers.
- Operations within the schedule are moved to further reduce the number of flip-flops in the circuit, thereby reducing both area and power.
- Registers are inserted into the circuit to reflect the pipelining of the schedule.
- If the initiation interval is greater than 1, the synthesized circuit is slightly modified to preserve correctness required by the implicit slowdown transformation.

4. RETIMED MODULO SCHEDULING

Iterative modulo scheduling provides the basic framework for the scheduling algorithm presented here, but generalized to reflect the additional degrees of freedom available in a reconfigurable target. In particular, the modifications include the following concepts and techniques from retiming:

- A variable clock period.
- Scheduling of operations with non-integral propagation delays.
- An explicit representation of registers to facilitate proper initialization and sequencing
- Chaining of operations between registers, effectively scheduling onto continuous time rather than into discrete timeslots.

In addition, we also support the scheduling of both pipelined and unpipelined operations.
4.1 Dependence Circuit

The input to the scheduler is a “dependence circuit” that is similar to the dependence graph used by many compilers, but differs in its timing model and in the way that recurrences are represented.

Nodes in the dependence circuit represent primitive operations in the target platform. When targeting an FPGA, for example, a node might represent a ripple adder, multiplexer, lookup table (LUT), block RAM, register, or other primitive. Each node is tagged with two attributes that determine how much time is required for the operation to execute and how the operation may be scheduled: internalPipelineStages and propagationDelay.

The internalPipelineStages attribute of a node is simply the number of internal pipeline stages in the operation. For LUTs, adders and other combinational primitives, it is 0; for registers it is 1; for more complex components, such as synchronous RAMs or pipelined multipliers, it can be 1 or greater. An operation with no internal pipeline stages may be scheduled to begin execution at any time within a clock period, or timeslot, as long it completes before the end of that period. A pipelined operation, though, may only be scheduled to begin at the start of a timeslot.

The propagationDelay attribute is a real number representing the amount of time needed for the operation to complete execution, measured from the triggering of the last internal pipeline stage (for pipelined operations) or from the time the operation was initiated (for non-pipelined operations). The propagationDelay for any operation must be not greater than the clock period or it cannot be scheduled.

The total execution time of an operation, called its delay, is thus determined by its propagationDelay, internalPipelineStages and the clock period as illustrated in Figure 7.

Edges in the circuit represent data flow from a source operation to a sink operation. Each is tagged with a distance attribute which represents the number of iterations that the data value crosses between its production and its consumption. To illustrate the concept of distance, consider the following loop:

```c
for (int i = 0; i < 100; i++)
  sum += array[i];
```

The first operation (multiplication) produces a value that is consumed by the second operation (addition) in the same iteration, so

```
while (someTest) {
  int j = i * 2;
  i = j + 1;
}
```

The edge representing this data flow is assigned a distance of 0. The value produced by the addition, though, is not consumed by the multiplication until the following iteration—the variable i, in fact, is used to store that result across the iteration boundary—so the corresponding edge is assigned a distance of 1. This flow of a data value between iterations, called a recurrence, is represented in the dependence circuit by a combination of distance-tagged edges and explicit registers to hold the data (in iterative modulo scheduling, registers are represented implicitly). Figure 8 illustrates a dependence circuit for a simple loop with a recurrence. Recurrences in array accesses are only slightly more complicated. Such an array is typically mapped to a RAM, with read and write accesses represented by read-port and write-port operations. The recurrence is represented by an edge from the write port to the read port, annotated with the distance as shown in Figure 9.

4.2 Recurrences vs. Delays

Registers serve two very different functions in pipelined circuits, and it is necessary to make the distinction early on since it impacts the pipelining algorithm (this distinction is not made in retiming).

```c
int array[100 + K];
int sum = 0;
for (int i = 0; i < 100; i++)
  array[i+K] = i;
sum += array[i];
```

The first operation (multiplication) produces a value that is consumed by the second operation (addition) in the same iteration, so
As mentioned earlier, a register in the initial dependence circuit is used to carry a data value from one iteration to a later one—this will be called a recurrence register. But a register can also be used to delay the propagation of data within the same iteration—this is, in fact, how registers are used when they are inserted to pipeline a signal—and we will refer to this as a delay register.

The registers within pipelined operations are generally delay registers, intended to reduce the clock period of the computation while delivering a result to the same iteration. One exception is that the first register in the pipelined operation may serve the function of a recurrence register if the input value was generated for consumption in a later iteration.

In the dependence circuit, the two different register usages are represented by the distances tagged on their input edges: recurrence registers have input edges with distance greater than 0, while delay registers have input edges with distance equal to 0. Registers within pipelined operations are delay registers, except that the first register may serve as a recurrence register if the input edge to the operation has a distance greater than 0.

### 4.3 Initiation Interval

The initiation interval, II, is the number of clock cycles separating successive iterations of the loop. In order to maximize the throughput of the loop, it is desirable to make II as small as possible. The minimum possible II, called the MII, is the larger of two values called ResMII and RecMII.

ResMII reflects the impact of contention for limited resources on the ability to create a tight schedule. This is less of an issue in reconfigurable or custom hardware than in sequential processors, since there is considerably less resource sharing in reconfigurable targets. The reduced sharing stems not only from the abundance of resources, but from the high sharing overhead. For example, in many FPGAs a multiplexer is as expensive as an adder, so that attempting to share an adder by multiplexing its inputs offers no advantage. Resource conflicts do show up, though, in attempting to allocate the typically small number of read- and write-ports on internal RAMs to multiple RAM accesses within an iteration—in that situation, sharing is often unavoidable. ResMII is computed as in iterative modulo scheduling [5].

RecMII reflects the impact of feedback (recurrences) on the ability to create a tight schedule. It may be computed by modifying the function ComputeMinDist [5], replacing all references to the “delay” of a node with the number of “internalPipelineStages”.

### 4.4 Clock Period

Different clock periods for pipelining a computation provide trade-offs between performance and area. Shorter clock periods will, up to a point, tend to increase the throughput of a circuit, but will also tend to increase the number of pipeline stages, thereby increasing the number of flip-flops in the final implementation. At some point, though, decreasing the clock period further offers no performance gain and can actually degrade performance as the circuit spreads out in space to accommodate the additional flip-flops.

Figure 4 illustrates the problem: an elementary feedback circuit within a larger circuit contains two operations, each with a propagation delay of 10 nsec, and a single register. For an II of 1, the minimum clock period at which this circuit can operate correctly is 20 nsec. A smaller clock period for this circuit can be obtained by performing a 2-slowdown transformation (Figure 4b) and redistributing the registers using retiming (Figure 4c). But the 2-slowdown transformation is equivalent to increasing the II to 2.

So even though the resulting circuit will execute at twice the clock rate (100 MHz vs. 50 MHz), the computation requires twice as many cycles as the original circuit and so the throughput is the same.

The driving force in clock period selection is the II—the smaller the better, since a small II minimizes the number of registers created by the slowdown transformation. Once the II has been determined, a lower bound on the clock period needed for maximum throughput can be determined.

The minimum clock period, MinCP, is calculated in a manner analogous to the computation of MII: it is the maximum of the resource-constrained minimum clock period, ResMinCP, and the recurrence-constrained minimum clock period, RecMinCP.

RecMinCP is simply the maximum of the following values:

- the largest propagationDelay of any operation in the circuit,
- the largest minimum clock period required by any pipelined component,
- the smallest clock period that can be provided by the system clock.

RecMinCP is a function of the initiation interval. Conceptually one can think of it as a measure of how evenly one can redistribute registers around each elementary circuit [17] within the design, such that the maximum propagation delay from register output to register input is minimized. In retiming a synchronous circuit containing only registers and non-pipelined operations, a lower bound on the clock period obtainable by spreading the registers around the circuit can be derived. Let $C$ represent the set of all elementary circuits within a design, with $c$ used to represent a member of that set. Let the function $\text{propagationDelay}(c)$ denote the sum of the propagation delays for all operations in elementary circuit $c$, and let the function $\text{registers}(c)$ represent the number of registers in circuit $c$. Papaefthymiou has shown [6] that, for a synchronous circuit without pipelined operations before slowdown, $\text{RecMinCP} = \max_{c \in C} \left( \frac{\text{propagationDelay}(c)}{\text{registers}(c)} \right)$

where II is the desired slowdown factor.

Pipelined operations complicate this a bit, since the registers within such operations are fixed in place and unavailable for distribution around the loop. Furthermore, as mentioned earlier, internal registers generally act as delays, not as mechanisms for implementing recurrences. The only registers available for distribution within the loop are recurrence registers (there must always be at least one) which are recognized by the non-zero distances on their input edges. The correct method for determining RecMinCP is thus to focus on the sum of the distances around the loop, not the registers. Let $\text{dist}(c)$ represent the sum of distances for all edges within an elementary circuit $c$. Then RecMinCP can be reformulated to handle both pipelined and unpipelined operations thus:

$\text{RecMinCP} = \max_{c \in C} \left( \frac{\text{propagationDelay}(c)}{II \times \text{dist}(c)} \right)$

Similarly it can be shown that a recurrence-constrained upper bound on the minimum clock period, RecMaxCP, is:

$\text{RecMaxCP} = \text{RecMinCP} + \max_{op \in O} \left( \text{propagationDelay}(op) \right)$

where $op$ is an operation and $O$ is the set of operations in all elementary circuits.
4.5 Scheduling
The core scheduling algorithm may be viewed as a modification of iterative modulo scheduling. Since that algorithm has been described in detail by Rau [5], this section will focus on the differences. The main changes include:

- Treating the clock period as a variable rather than as a constant.
- Scheduling of operations onto continuous time rather than discrete timeslots.
- Explicit representation of recurrence registers (but not delay registers) in the dependence circuit.
- Real, rather than integer, scheduling priorities for operations.

The top level procedure, `RetimedModuloSchedule()`, is shown in Figure 10. The procedure takes a single parameter, `clockDilation`, which is used by the outer design-space-exploration loop to force the scheduler to consider longer clock periods, and hence shorter pipelines, than the minimum possible.

The algorithm begins by calculating the minimum clock period (which may be increased by the `clockDilation` parameter) and minimum II and attempts to schedule the loop using those parameters. If the scheduling fails, it is because either the clock period or II is too small. Since it generally preferred to keep the II as small as possible, the clock period is increased slightly and scheduling reattempted. If this still fails, the process of slightly increasing the clock period and rescheduling is repeated until the clock period reaches the upper bound, RecMaxCP. If scheduling still fails at the upper bound of the clock period, it is assumed that the problem lies with the II, so the clock period is reset to its initial minimum value, the II is incremented, and the entire process repeated until scheduling succeeds. Note that the process of exploring the clock period could be accelerated with a binary search.

Since operations are scheduled onto a continuous timeline rather than discrete timeslots, some modifications to IMS’s notions of time, delay and scheduling priorities are required. First, the timescale is normalized such that the current clock period has the value of 1.0. This is accomplished by the procedure `setClockPeriod()` in Figure 10 which is used to inform each operation in the loop of the current clock period being investigated. Thus the formula for the delay of an operation becomes:

\[
\text{delay}(op) = \max(0, \text{pipelineStage}(op) - 1) + \frac{\text{propagationDelay}(op)}{\text{clockPeriod}}
\]

When computing the earliest and latest possible scheduling time for an operation, the following constraints apply:

- pipelined operations may only start at integer valued times (1.0, 2.0, ...)  
- unpipelined operations may start at any time, as long as they complete before the end of a timeslot (e.g. an unpipelined operation with a normalized propagation delay of .6 may not be scheduled later than .4 within any timeslot).

Since the scheduling priority of an operation is based on the schedule times of its predecessors, scheduling priority becomes a real value, and the scheduling priority queue must sort operations based on their real-valued priorities.

4.6 Schedule Compaction
In some cases it is possible to reduce the number of pipeline stages by reducing the slack between each strongly connected component (SCC) and all the operations that feed it.

```java
procedure RetimedModuloSchedule(float clockDilation) {
    // clockDilation must be >= 1.0. It is used by the caller to 
    // explore parts of the design space with fewer pipeline stages.
    II = max(resMII()) + minClockPeriod;  
    maxClockPeriod = max(resMaxCP(), recMaxCP(II)) / clockDilation;  
    clockPeriod = minClockPeriod;  
    do {
        if scheduling was successful
            return;
        clockPeriod = clockPeriod * 1.1;
    } while (clockPeriod <= maxClockPeriod * 1.1);
    II = II + 1;
    clockPeriod = minClockPeriod;
}
```

Figure 10. Retimed modulo scheduling main algorithm.

The slack of a dependence edge is defined to be the difference between the time that the edge’s value is consumed and the time that it is produced. If the slack of all edges feeding an SCC equals or exceeds 1.0, then potentially that SCC can be moved to an earlier point in the schedule while preserving the pipeline’s functionality. Similarly, if the slack of all outputs leaving an SCC equals or exceeds 1.0, the component could, perhaps, be moved to a later point in the schedule.

When moving an SCC either forwards or backwards, one must be careful not to destroy the correct operation of the pipeline, since such motion can potentially upset the resource time-sharing that had been so carefully crafted by the modulo scheduling. To avoid this, it is necessary to examine the resource usage of the SCC: if it shares no resources with the rest of the circuit, it may be safely moved an integral number of timeslots within the bounds dictated by the slack, otherwise it may only be shifted by a multiple of the initiation interval.

Schedule compaction simply involves minimizing the slack on all input edges to all SCCs, moving each SCC backwards to its earliest legal point in the schedule.

4.7 Flip-flop Reduction
Although schedule compaction can reduce the number of flip-flops in the final implementation, there remains additional potential for reduction by moving SCCs within the compacted schedule. If slack exists on all inputs and outputs, the SCC may moved either earlier or later, with one direction usually more advantageous than the other, as illustrated in figure 11.
There are many different ways in which the operations can be shifted around in order to reduce flip-flops; our compiler examines two points in that space and selects the one that yields the lowest register count. The first design point involves moving all SCC’s to their earliest possible times in the schedule (which might reduce the number of stages in the pipeline), and then “relaxing” the SCCs (in the manner suggested by figure 11) to find a local minimum. The second design point starts from the compacted and relaxed first point, moves all SCC’s to their latest possible times in the schedule, without increasing the number of stages, and then relaxes them to find another local mimima. The design point with the smaller number of flip-flops is then selected as the result of the reduction.

Figure 11. Differences in data widths bias the direction of motion during flip-flop reduction. In the original circuit (center), operations a and b are scheduled 2 timeslots apart, as are b and c, but the data value from a to b is only 4 bits wide, while the data value from b to c is 8 bits wide. Moving operation b forwards thus reduces the number of flip-flops needed to pipeline the operations; moving it backwards increases them.

4.8 Register Insertion

Scheduling causes the results of some operations to be consumed in timeslots other than the one in which they were produced. When this happens, delay registers are inserted to delay the values to their proper times (Figure 12).

4.9 Predicates and Initial Values

Predicates are used to enable the writing of state variables (recurrence registers and RAMs). Figure 13 shows how predication is implemented on recurrence registers: a predicate controlling the writing of a state variable drives the enable input of the register holding that variable. Since predicates appear as ordinary signals within the dependence circuit, pipelining automatically inserts delay registers as needed on predicate signals so that the pipelined circuit behaves correctly. This highlights one difference in the implementation of the two different types of registers: recurrence registers require enable inputs, while delay registers do not. This same mechanism also partly solves the problem of ensuring that variables with initial values are implemented correctly. The remainder of the solution requires that recurrence registers not only have enable inputs, but also be capable of being initialized to a desired state (e.g., on power-up or by a global initialization signal). Furthermore, delay registers must be initialized to the zero state. The result is that recurrence registers are automatically sequenced and initialized correctly by the pipelining.

How this works is illustrated in Figure 14, which shows a simple loop with two recurrences (represented by the variables x and y) mapped onto a two-stage pipeline with an II of 1. At the beginning of execution, the recurrence registers are initialized with their ini-

Figure 12. Register insertion. After scheduling, the result of operation a is produced in one timeslot, but consumed by operation b in the following timeslot. Inserting a delay register (gray rectangle) delivers the value to b at the proper time.

Figure 13. Predicates and recurrence registers. The recurrence register driving the signal x is enabled by the predicate, p, so that x is incremented only when p is true. Recurrence registers require enable inputs for their implementation. (Predicates for writing RAMs are implemented using enable inputs on write ports.)

Figure 14. Initial values and pipelining. Recurrence registers (black rectangles) are required to have both enable inputs (left side of rectangle) and the ability to be initialized to a given value (right side). Delay registers (gray rectangles) inserted for pipelining do not require enable inputs, but must be initialized to 0. Pipelining delays the enable signals to recurrence registers, so that they hold their initial values until needed.
4.10 Sequencer Fixup for II > 1

If the initialization interval is greater than one, only 1/II stages of the pipeline are computing useful data at any given time. The pipeline generated to this point, though, potentially enables all stages on every cycle. This presents a problem with state variables, particularly those stored in RAMs, since it allows garbage to be written into state variables during the “off” cycles, potentially corrupting the computation. This is easily corrected by ANDing an additional signal with the enables of recurrence registers and RAM write ports (Figure 15).

Table 1 Compiling the Viterbi decoder onto an XC2V250 FPGA.

<table>
<thead>
<tr>
<th>implementation</th>
<th>freq(MHz)</th>
<th>stages</th>
<th>LUTs</th>
<th>flip-flops (ex. RAM)</th>
<th>gates</th>
<th>relative power</th>
<th>relative energy</th>
<th>including RAM</th>
<th>excluding RAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>191</td>
<td>13</td>
<td>377</td>
<td>1295</td>
<td>13,969</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>b</td>
<td>161</td>
<td>12</td>
<td>377</td>
<td>1146</td>
<td>12,973</td>
<td>.90</td>
<td>.95</td>
<td>.77</td>
<td>.92</td>
</tr>
<tr>
<td>c</td>
<td>151</td>
<td>10</td>
<td>377</td>
<td>1023</td>
<td>12,229</td>
<td>.91</td>
<td>.91</td>
<td>.77</td>
<td>.82</td>
</tr>
<tr>
<td>d</td>
<td>132</td>
<td>7</td>
<td>372</td>
<td>796</td>
<td>9,877</td>
<td>.99</td>
<td>.96</td>
<td>.73</td>
<td>.77</td>
</tr>
<tr>
<td>e</td>
<td>76</td>
<td>4</td>
<td>371</td>
<td>355</td>
<td>6,033</td>
<td>.82</td>
<td>.82</td>
<td>.76</td>
<td>.76</td>
</tr>
<tr>
<td>f</td>
<td>49</td>
<td>3</td>
<td>370</td>
<td>239</td>
<td>5,169</td>
<td>.91</td>
<td>.91</td>
<td>.71</td>
<td>.71</td>
</tr>
</tbody>
</table>

5. RESULTS

To show the effectiveness of retimed modulo scheduling, we show the results of exploring the pipeline design space of a single application, a Viterbi decoder. The application, written in C, is a nested loop with several arrays, forcing the use of synchronous, internal RAM in addition to LUTs. Our compiler, called Xax, was used to compile the C to configuration bits for a Xilinx XC2V250 FPGA. The resulting data regarding circuit size and speed come from the Xilinx toolchain. Power was estimated using a simple model based on the Xilinx power estimator. Energy usage for an implementation was estimated by multiplying its power by the length of time needed for the computation to execute.

Table 1 shows the raw results. Six different implementations were produced, ranging from a high-speed (191 MHz) version with 13 pipeline stages, to a slow (49 MHz), low-power version with only 3 stages. Since the application is RAM intensive, and RAM requirements are not changed by pipelining, we estimated power and energy with and without the RAMs to illustrate the effects of pipelining on both RAM-intensive and non-RAM-intensive circuitry. The II for each implementation was 1.

Although the number of LUTs required for each implementation was nearly the same, the number of flip-flops varied by more than a factor of 5 from the smallest to the largest configuration, while power varied by about a factor of 4 (Figure 16). Despite the reduction in area, total energy usage was not greatly reduced in the smaller implementations, suggesting only modest benefit for battery-powered devices at slower speeds (Figure 17).

Perhaps the most interesting trade-off for this particular application regards area (Figure 18). The smallest Virtex II FPGA that the 191 MHz implementation could fit in is the XC2V250, while the 132 MHz implementation (and possibly the 151 MHz one as well)
could use the smaller XC2V80. The 76 MHz version could fit in the bottom-end XC2V40 for an even cheaper implementation.

6. RELATED WORK

The Garp compiler [18] also uses iterative modulo scheduling as the foundation of its pipelining algorithm, but Garp assumes a fixed clock frequency, so it’s not able to trade-off clock speed and area. Initialization of state variables in Garp is more complicated: a host controller steps the circuit up to points where recurrence values are first used, initializes the recurrence registers with the proper initial values, and then continues execution. Other systems which build pipelining on top of iterative modulo scheduling include NAPA-C [19] and PICO [20].

Hybridizations of classic retiming and scheduling include work by Srinivasan [21], Calland [22], Weinhardt [23] and O’Neil [24], Monteiro [25], Yun [26] and Musoll [27] have investigated incorporating power constraints into scheduling.

7. CONCLUSIONS

Reconfigurable logic offers degrees of freedom that cannot be exploited by iterative modulo scheduling: nonintegral operation propagation delay, variable clock frequency, and the ability to chain operations between registers. Retiming is unable to deal with resource constraints and pipelined components. Retimed modulo scheduling handles these opportunities and constraints, and allows automated exploration of the pipeline implementation space, investigating trade-offs between throughput, power, energy and area.

8. REFERENCES