

# Synthesis from Multi-cycle Atomic Actions as a Solution to the Timing Closure Problem

Michal Karczmarek and Arvind  
Computer Science and Artificial Intelligence Lab  
Massachusetts Institute of Technology  
Cambridge, Massachusetts 02139  
{karczma@alum arvind@csail}.mit.edu

**Abstract:** *One solution to the timing closure problem is to perform infrequent operations in more than one cycle. Despite simplicity of the solution statement, it is not easily considered because it requires changes in RTL, which, in turn, exacerbates the verification problem. We offer a timing closure solution guaranteed to preserve functional correctness of designs expressed using atomic actions or rules. We exploit the fact that the semantics of atomic actions are untimed, that is, the time to execute an action is not specified. The current hardware synthesis technique from atomic actions assumes that each rule takes one clock cycle to complete its computation. Consequently, the rule with the longest combinational path determines the clock cycle of the entire design, often leading to needlessly slow circuits.*

*We present a synthesis procedure for a system where the combinational circuits embodied in a rule can take multiple cycles without changing the semantics of the original design. We also present preliminary results based on an experimental compiler which uses the Bluespec (BSV) compiler front end and generates Verilog. The results show that the clock speed and the performance of circuits can be improved substantially by allowing slow paths to complete over multiple cycles. Our technique is orthogonal to solutions based on multiple clock domains.*

## 1 Introduction

Sometimes the timing closure problem is exacerbated by an infrequently taken slow combinational path in a design. For example, in a simple 5-stage pipelined CPU, the designer has to deal with the effect of cache misses, exception handling, and inherently slow instructions such as division in a way that it does not slow down the over all clock cycle. A common solution is to perform such operations in multiple cycles without worrying about the pipelining (throughput) of such operations. Often a designer deals with such issues at the micro-architecture level to avoid the timing closure problem altogether. However, there is a more insidious version of this problem where the architect or the designer is not aware of the timing problem until much later in the design process. In such cases an automatic transformation of the design, where the slow paths are turned into multi-cycle paths without affecting the semantics of the design, could be of great help to the designer. This paper offers such a transformation for designs expressed using atomic actions, specifically designs expressed in Bluespec. [1, 2]

The synthesis procedure underlying Bluespec[10] assumes that each atomic action or rule (we use these words interchangeably in this paper) can be performed in one cycle. This assumption dramatically simplifies both the analysis and implementation of concurrent atomic actions. In an

edge-triggered synchronous implementation, all the rules simply read the state elements at the beginning of the clock cycle and all the “non-conflicting” rules commit their updates at the end of the clock cycle. Thus, no shadow states or locking of various state elements is required to preserve atomicity. The drawback is that the rule with the slowest combinational path determines the overall clock period.

In this paper we modify the Bluespec synthesis procedure by breaking it down into several steps. We first estimate the combinational delay of each operation in every rule and replace the long latency operations *in situ* by an equivalent multi-cycle version of the operation by inserting flip-flops. (We don’t give any estimation procedure but any classical technique can be used for this purpose). This transformation results in an equivalent multi-cycle version of each rule which had long combinational delay. *The main technical contribution of this paper is a procedure to construct a scheduler, which concurrently schedules as many multi-cycle rules as possible. Our procedure is guaranteed to do at least as well as the procedure underlying the current BSV compiler in terms of cycle times and overall circuit performance*[10].

The main technique employed by our scheduling algorithm is based on optimistic concurrency ideas used in Transactional Memory systems but our implementation is much more fine-grained and efficient because of static analysis[3, 4, 7, 9]. We rely on the compiler to compute the read-write sets of each rule and the maximum amount of time a rule requires before it can commit results.

The remainder of the paper is organized as follows: Section 2 will introduce Bluespec; Section 3 will introduce multi-cycle rules and associated scheduling techniques; Section 4 presents our algorithm; Section 5 presents our results; Section 6 discusses related work; and finally, we present some ideas for future work.

## 2 Bluespec Explained via GCD

In this paper we will use the syntax and semantics of a simple Atomic Action language with guards described by Dave et al[5]. (This language represents the essence of Bluespec). Consider the example in Figure 1, which describes a module to compute the GCD of two numbers using the Euclidean method. The module has two registers  $x$  and  $y$ , one rule called *GCD*, and two interface methods called *Seed* and *Result*. The *Seed* method stores the initial numbers in registers  $x$  and  $y$ , the *GCD* rule computes successive remainders of the inputs until the remainder becomes 0, and

the *Result* method returns the computed answer. The rule has the guard *when*  $y \neq 0$  which must be *true* for the rule to execute. Similarly, the methods have guards which must be *true* before the method can be applied. The guards for the *Seed* and *Result* methods indicate that they can be activated only when the module is not already busy computing, (i.e. *when*  $y = 0$ ). “%” represents the remainder operator while “|” represents parallel composition of two actions and is a commutative operator. Thus, if we write  $(x := y \mid y := x)$  the values of  $x$  and  $y$  would be swapped at the end of the action. Also parallel assignments to a state variable in an action are illegal. The source language allows conditional assignments to a register but provides no ability to sequence actions in rule.

In Figure 2 we provide a circuit implementation of the *GCD* example. The  $x$  and  $y$  registers are updated when either the *GCD* rule is active or the *Seed* method is started. In this example, the conditions for these two actions are mutually exclusive because of the guards. The compiler has to make sure that it never asserts the *start* wire for a rule unless its *rdy* signal is high. If *Seed*’s *start* signal is asserted, the input values  $a$  and  $b$  are copied into  $x$  and  $y$ , respectively. When *Result*’s guard ( $rdy_{Result}$ ) is *true*, the user can read the output, which is simply the value of register  $x$ . Since *Result* does not modify any of registers in the module, the user can read the output without further communication. Rule *GCD* is enabled whenever its guard evaluates to *true*. When *GCD* is executed, it will copy  $y$  to  $x$ , and write the remainder,  $x \% y$ , to  $y$ .

In Figure 3 we provide a model for building circuits from Bluespec descriptions without methods. All the guards are evaluated every cycle. The Rule Selector selects which rules to fire by asserting their *start* signals. Only rules with guards evaluating to *true* may fire, and the Rule Selector ensures that all selected rules can fire concurrently. The updates for all the rules are also computed every cycle. The MUX selects all the state updates which should happen as a result of firing the selected rules and sends them back to the global state. All this computation happens in a single cycle. The construction of a good Rule Selector is the central problem in high-quality synthesis from guarded atomic actions. It involves a lot static analysis on the part of the compiler to determine which rules are *Conflict Free* and which rules are *Sequentially Composable*[10]; we will describe these concepts later when we need them. The scheduler also relies on user annotation to prioritize the selection among conflicting rules that may be ready to fire at the same time. These annotations are used to create the *Urgency List*, which is a total ordering of all rules. Rule  $A$  is *more urgent* than rule  $B$  if  $A$  appears before  $B$  on the Urgency List, and is written  $A \rightarrow B$ . If  $A \rightarrow B$  and both  $rdy_A$  and  $rdy_B$  are *true* in the same cycle,  $A$  will fire and may block  $B$  from firing if  $A$  and  $B$  are conflicting.

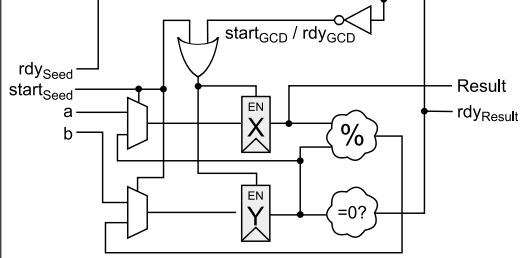
The circuit in Figure 3 will preserve atomicity, but potentially suffers from a serious flaw: if the designer creates a rule with a slow combinational path, that rule will slow down the clock speed for the entire design. The remain-

```

Module doGcd
  Register x = 0, y = 0
  Rule GCD :: (x := y | y := x % y) when y ≠ 0
  ActMeth Seed(a, b) :: (x := a | y := b) when y = 0
  ValMeth Result :: return x when y = 0
  Note: % operator represents remainder computation

```

Figure 1. GCD example



Note:  $start_{GCD}$  and  $rdy_{GCD}$  hold the same value.

Figure 2. Single-cycle GCD

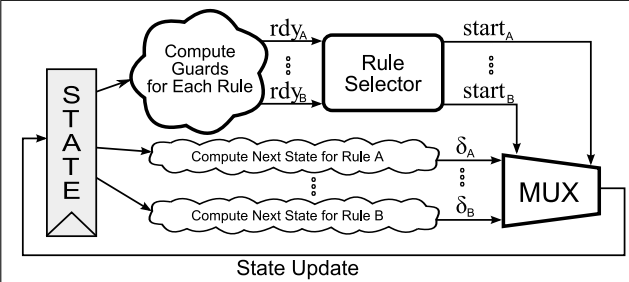


Figure 3. Standard Bluespec Model

der computation in our GCD example is one such combinational path. Our goal is to alleviate this problem by providing an automated technique to permit synthesis of efficient circuits with slow combinational paths in Bluespec while maintaining atomicity. The solution we provide relaxes the requirement that all Bluespec rules complete their firing within a single cycle. Thus a slow combinational path will be permitted to perform its computation over multiple cycles, allowing the circuit to function using a fast clock.

### 3 Multi-Cycle Rule Scheduling

#### 3.1 Two-cycle GCD

Consider the GCD design shown in Figure 4. The new design splits the computation of the remainder into two steps,  $\%_1$  and  $\%_2$ . The intermediate result is stored in a temporary register,  $t$ . We have used the sequential connective operator “;” to signify the two steps. The meaning of the sequential connective is what one would expect: the value assigned to  $t$  is visible to subsequent actions within the rule. A circuit implementation of this module can now perform the two steps in two cycles. The new *GCD* rule is a multi-cycle rule. Assuming that the remainder operation was the critical path in our circuit, the new implementation may improve the clock cycle time by up to a factor of 2. This cycle time reduction does not come for free. We have introduced extra state, and will have to address the problem of maintaining rule atomicity.

Figure 5 provides a circuit implementation of the code

from Figure 4. There are two notable differences between this circuit and one in Figure 2. The first difference is that we have split the computation of the remainder into two cycles (represented by drawing a register across the computation circuit – this register is the temporary  $t$  from Figure 4). The second is that we have changed the signal indicating the completion of the  $GCD$  rule. The new signal is called  $commit_{GCD}$  and is asserted one cycle after the rule becomes active because the computation of the remainder now takes 2 cycles.

Assuming that splitting of the remainder computation into two cycles has been done reasonably well, we may have reduced the clock cycle of this module by as much as 50%. Of course this clock reduction is unlikely to speedup the computation of GCD itself because GCD computation will require twice as many cycles as before. However, it should speedup the encompassing design if GCD was the circuit determining the clock of the whole design. Note that this example is exceedingly simple, because it only deals with scheduling a single rule. Real designs require scheduling multiple rules.

### 3.2 A Reference Scheduler for Multi-cycle Rules

Consider a reference schedule for multi-cycle execution of rules, shown in Figure 6. The reference scheduler allows only a single rule to be active in any cycle, which is selected by the Rule Selector. The fundamental difference between the models in Figures 3 and 6 comes from separating the  $start$  and  $commit$  signals and having a counter (CyclesLeft) which counts down as the data propagates through the currently active rule. When all rules take one cycle, as in Figure 3, the  $start$  and  $commit$  signals are one and the same. In our new model, asserting  $start_A$  indicates that rule  $A$  is firing and  $commit_A$  indicates that rule  $A$  is committing its updates. Once  $start_A$  is asserted it remains active until  $commit_A$  is asserted.

The first cycle in which  $start_A$  is asserted, CyclesLeft is set to  $L_A$ , the computation delay of rule  $A$ . When CyclesLeft reaches 0, the  $commit_A$  signal for rule  $A$  is asserted and the computed data is stored in the target state registers. The CyclesLeft counter is loaded with a new value only when its present value is zero, and is decremented by 1 otherwise. Thus, the effect of the scheduler is felt only on those clock cycles when a scheduled rule has just completed or when the whole system is idling. This implementation guarantees rule atomicity because inputs to an active rule cannot change as no other rule is active.

The circuits to compute the next state can be built using multi-cycle combinational paths or using flip-flops for temporary variables. These flip-flops are not globally visible. All guards are still evaluated in a single cycle.

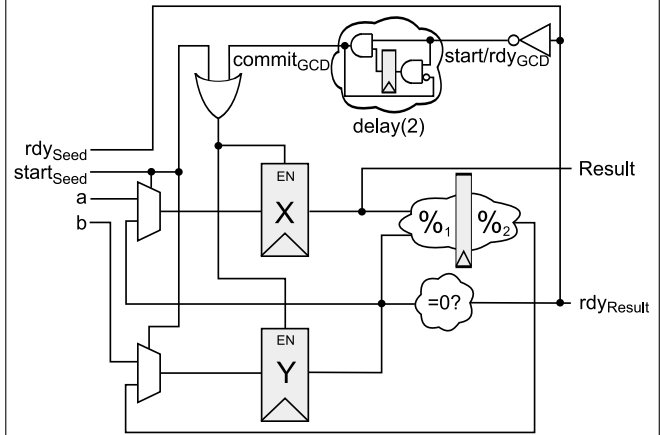
### 3.3 Read-Write Policy for Concurrent Scheduling of Rules

When multiple multi-cycle rules are active concurrently, we have to make sure that an active rule cannot destroy the input registers of another active rule. The procedure to guarantee this condition has to know exactly when a rule reads

#### Module *doGcd2*

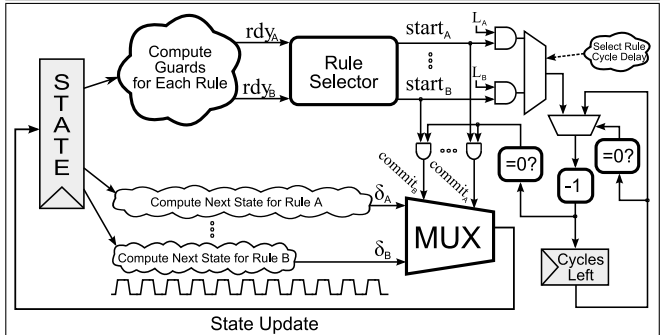
```
Register  $x = 0, y = 0, t$ 
Rule  $GCD :: (x := y \mid (t := \%_1(x, y);$ 
            $y := \%_2(x, y, t)))$  when  $y \neq 0$ 
ActMeth  $Seed(a, b) :: (x := a \mid y := b)$  when  $y = 0$ 
ValMeth  $Result :: return x$  when  $y = 0$ 
```

Figure 4. 2-cycle GCD example



The remainder calculation takes 2 cycles. Output of  $delay(2)$  circuit is  $true$  once its input is  $true$  for 2 cycles.

Figure 5. 2-cycle GCD



Rule Selector is a pure function and can assert only one  $start$  in any cycle.

Figure 6. One Rule-At-a-Time Multi-cycle model

and writes its various registers. For example, we could assume that a rule reads all its input registers in the first cycle it becomes active and after that those input registers can be changed without affecting the behavior of the rule. This will potentially introduce a lot of “shadow registers”. On the writing side, one can imagine writing the various output registers of a rule in any cycle whenever the associated data become available. These read-write policies can get complicated and their effect on the final results can be quite difficult to analyze. Therefore, we use a simple *read-throughout – write-last* policy.

*Read throughout* policy means that a rule can read its input registers in any cycle the rule is active, and consequently, the register must hold its value during that time. *Write-last* policy means that a rule does not commit any new values to its output registers until the very last cycle in which the rule

is active. This read-write policy simplifies the scheduling problem and helps implementing if-statements correctly.

In order to define the interaction between rules in reading and writing each others input and output registers we need the following definitions:

**Definition** (Range and Domain): Range of rule  $A$  is the set of all registers  $A$  can update and is written as  $R[A]$ . Domain of rule  $A$  is the set of all registers which  $A$  can read potentially and is written as  $D[A]$ .

**Definition** (Conflict Free Rules): Two rules  $A$  and  $B$  are Conflict Free ( $A \text{ CF } B$ ) iff  $(D[A] \cap R[B] = \emptyset) \wedge (D[B] \cap R[A] = \emptyset) \wedge (R[A] \cap R[B] = \emptyset)$ .

**Definition** (Sequentially Composable Rules): Two rules  $A$  and  $B$  are Sequentially Composable ( $A < B$ ) iff  $(R[A] \cap (D[B] \cup R[B]) = \emptyset) \wedge (D[A] \cap R[B] \neq \emptyset)$ .

Note: this definition is slightly different from [10] by precluding Conflict Free rules from being treated as Sequentially Composable Rules.

**Definition** (Conflicting Rules): Two rules  $A$  and  $B$  are Conflicting ( $A <> B$ ) iff  $\neg((A \text{ CF } B) \vee (A < B) \vee (B < A))$ .

We will use these definitions to decide which rules can be scheduled to execute concurrently.

### 3.4 Opportunistic-Commit Scheduler

In Figure 7 we present an improvement over the scheduler in Figure 6. Notice that the circuits in the scheduler in Figure 6 compute updates for all the rules all the time even though they commit at most one rule in a cycle. The new scheduler opportunistically commits a non-conflicting subset of rules among those rules whose guards are true, and who have completed their next state computations. In order to keep track of which rules are ready to commit we have created a separate *CyclesLeft* counter for each rule. The counter for rule  $A$  is reset whenever a commit affects any of the registers that  $A$  reads. In order to determine when some register in  $R[A]$  has changed, we watch all *commit* commands and compute all the *inCh* (input change) signals as follows:

$$\begin{aligned} inCh_A(t) &= \bigvee_{B \in \mathcal{R}} ((A <> B \vee A < B) \wedge commit_B(t)) \\ cycLeft_A(t) &= inCh_A(t-1) ? (L_A - 1) : \\ &\quad \max(cycLeft_A(t-1) - 1, 0) \end{aligned}$$

$$rdyCommit_A(t) = (cycLeft_A(t) = 0)$$

where  $L_A$  is the propagation delay of rule  $A$ , which is the maximum number of cycles it takes for data from  $D[A]$  to propagate through  $A$ 's computational logic.

One interesting feature of the scheduler in Figure 7 is that it can use the same the Rule Selector circuit to generate *commit* signals that is used in the current Bluespec (BSV) compiler to generate *start* signals. The Rule Selector requires no new static analysis because we have removed the rules which are not ready to commit from the set of rules for consideration for scheduling; among these rules the Hoe-Arvind scheduler [10] is guaranteed to pick only those rules for committing that will not destroy the rule atomicity property.

### 3.5 Greedy-Reservation Opportunistic-Commit (GROC) Scheduler

In order to overcome the problem of unfairness we will use a system where the scheduler can use a *rsrv* command to

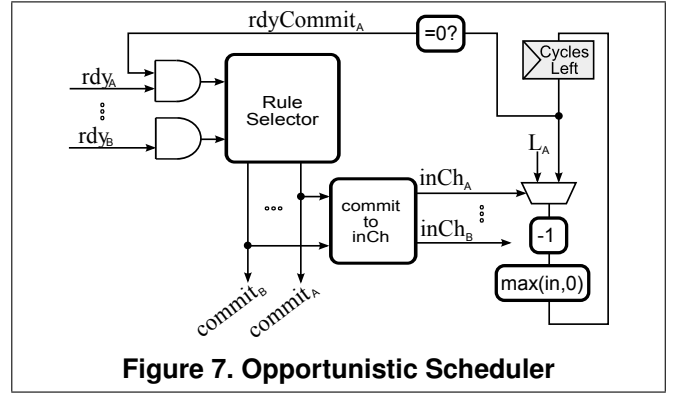


Figure 7. Opportunistic Scheduler

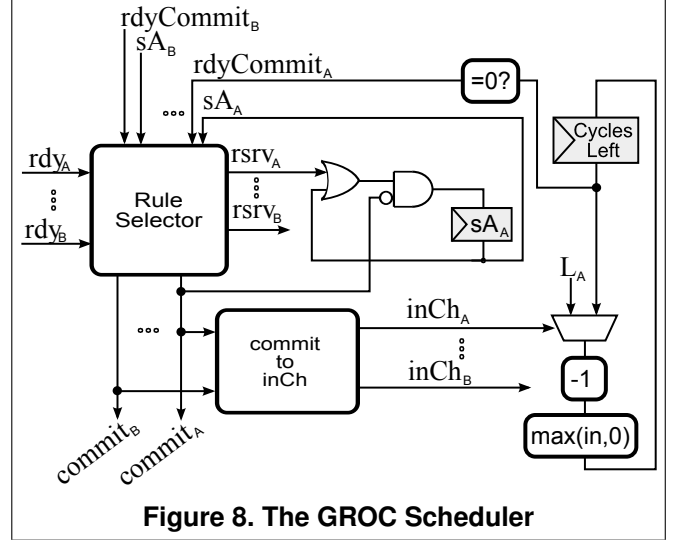


Figure 8. The GROC Scheduler

reserve a turn for a rule to commit its updates. Once a *rsrv* command has been issued, the rule is considered active and only an active rule is allowed to commit its update:

$$active_A(t) = (active_A(t-1) \wedge \neg commit_A(t-1)) \vee rsrv_A(t)$$

In order to keep track of which rules are active from cycle to cycle we need to introduce an extra bit of state per rule,  $sA_A$  (still active  $A$ ). It represents whether the particular rule was active in the previous cycle and did not commit, thus is still active in the current cycle.

$$\begin{aligned} sA_A(t) &= (sA_A(t-1) \vee rsrv_A(t-1)) \wedge \neg commit_A(t-1) \\ sA_A(0) &= false \end{aligned}$$

Once activated, the only way to deactivate a rule is by issuing its *commit* command. The ability to reserve a commit slot in the future ensures that no rule can be blocked permanently from committing by another conflicting rule. The new scheduler is shown in Figure 8. It uses the same definitions for  $inCh_A$ ,  $cycLeft_A$  and  $rdyCommit_A$  as in Section 3.4. We will design a new Rule Selector to generate *rsrv* and *commit* signals which will reserve and commit as many rules as possible while maintaining correctness.

## 4 A Greedy Rule Selector for the GROC Scheduler

Consider a scheduler that initially issues *rsrv* commands for a non-conflicting set of rules and simultaneously resets the *cycLeft* counter for these rules. Whenever a rule commits, the scheduler can issue additional reservations for some rules such that the new rules do not conflict with the

rules still holding reservations. This scheduler can be improved by resetting the *cycLeft* counter only when some commit affects the input of the rule and not at the time of reservation. As we will show, the GROC scheduler will allow opportunistic commits, that is, *rdyCommit<sub>A</sub>* may be asserted even before the rule ever becomes active. However, the Rule Selector of GROC scheduler will not issue a *commit<sub>A</sub>* unless rule *A* holds a reservation. The computation of the reservation and commit signals is a bit complicated but we will ensure that each execution of a rule *A* corresponds to the state of the input registers during a precisely defined cycle: the cycle in which the *rsrv<sub>A</sub>* signal is asserted. Earlier we have called this cycle the rule activation cycle. The results of an active rule are committed to the global state in the cycle when *commit<sub>A</sub>* signal is asserted. The subtlety is that the time duration between *commit<sub>A</sub>* and *rsrv<sub>A</sub>* may be less than *L<sub>A</sub>*.

#### 4.1 Scheduling Invariants

The Rule Selector preserves the following three invariants in order to guarantee atomic execution of the rules:

- Invariant 1.** if  $A <> B$ , a reservation for *A* cannot be issued if *B* already holds a reservation;
- Invariant 2.** if  $A < B$ , *B* cannot commit while *A* holds a reservation, i.e., is active;
- Invariant 3.** if  $A < B$ , a reservation for *A* cannot be issued if *B* holds a reservation, i.e., is already active;

We will now provide the motivation behind these invariants.

**Execution of Conflicting Rules:** Invariant 1 is obvious – concurrent scheduling of conflicting rules is guaranteed to lead to atomicity violation. Conflicting rules must be executed one after another.

**Rule Stretching:** Sequentially composable rules are characterized by having one rule read the output of the other, but not vice versa. Consider the example of two sequentially composable rules shown in Figure 9. Figure 9 demonstrates that it is possible for two SC rules to be active concurrently and still produce results consistent with atomic execution. In our example, the schedule of firing is *A, B, A, B*. Notice that even though *A* and *B* start in the same cycle, this is the only schedule which produces the final state in Figure 9.

When executing SC rules it is sometimes necessary to keep a rule active longer (delay its commit) than the expected number of cycles. We call this *rule stretching*. An example of rule stretching is demonstrated in Figure 10. The Figure demonstrates that committing rule *B* too early can lead to atomically inconsistent results of computation. This is the motivation underlying Invariant 2.

**Livelock Avoidance:** Consider the example shown in Figure 11. If all three rules are activated in the same cycle it will lead to a livelock, because Invariant 2 would not allow any of the three rules to commit. The issue here is that rules *A, B* and *C* form a cycle of dependencies. The Hoe-Arvind scheduler [10] avoided this problem by disallowing the concurrent scheduling of such cyclically dependent rules. Their scheduler permitted scheduling of only a subset of cycle-forming rules. Though such subsets are not unique, finding

Module *sequentially\_composable*

Register *x, y, t<sub>B</sub>*

Rule *A* ::  $x := f(x, y)$

Rule *B* ::  $(t_B := g_1(z); y := g_2(z, t_B))$

Urgency List:  $A \rightarrow B$

| cycle:         | 1                    | 2                         | 3                    | 4                         |
|----------------|----------------------|---------------------------|----------------------|---------------------------|
| A              |                      |                           |                      |                           |
| B              |                      |                           |                      |                           |
| x              | $x_1 = f(x_0, y_0)$  | $x_1$                     | $x_3 = f(x_1, y_2)$  | $x_3$                     |
| y              | $y_0$                | $y_2 = g_2(z_0, t_{B,1})$ | $y_2$                | $y_4 = g_2(z_0, t_{B,1})$ |
| z              | $z_0$                | $z_0$                     | $z_0$                | $z_0$                     |
| t <sub>B</sub> | $t_{B,1} = g_1(z_0)$ | $t_{B,1}$                 | $t_{B,3} = g_1(z_0)$ | $t_{B,3}$                 |

Figure 9. Sequentially Composable Rules

Module *rule\_stretching*

Register *x, y, z, t<sub>A</sub>*

Rule *A* ::  $(t_A := f_1(y); x := f_2(y, t_A))$

Rule *B* ::  $y := g(z)$

Urgency List:  $A \rightarrow B$

| cycle:         | 1                    | 2                         | cycle:         | 1                    | 2                         |
|----------------|----------------------|---------------------------|----------------|----------------------|---------------------------|
| A              |                      |                           | A              |                      |                           |
| B              |                      |                           | B              |                      |                           |
| x              | $x_0$                | $x_2 = f_2(y_1, t_{A,1})$ | x              | $x_0$                | $x_2 = f_2(y_0, t_{A,1})$ |
| y              | $y_1 = g(z_0)$       | $y_1$                     | y              | $y_0$                | $y_2 = g(z_0)$            |
| z              | $z_0$                | $z_0$                     | z              | $z_0$                | $z_0$                     |
| t <sub>A</sub> | $t_{A,1} = f_1(y_0)$ | $t_{A,1}$                 | t <sub>A</sub> | $t_{A,1} = f_1(y_0)$ | $t_{A,1}$                 |

Inconsistent Computation!      Delay committing B's Results until A finishes

Figure 10. Rule Stretching (Delayed Commit)

Module *deadlock*

Register *x, y, z*

Rule *A* ::  $x := f(y)$

Rule *B* ::  $y := g(z)$

Rule *C* ::  $z := h(x)$

Urgency List:  $A \rightarrow B \rightarrow C$

| cycle: | 1     | 2     | 3     | cycle: | 1     | 2     | 3     |
|--------|-------|-------|-------|--------|-------|-------|-------|
| A      |       |       |       | A      |       |       |       |
| B      |       |       |       | B      |       |       |       |
| C      |       |       |       | C      |       |       |       |
| x      | $x_0$ | $x_0$ | $x_0$ | x      | $x_0$ | $x_0$ | $x_0$ |
| y      | $y_0$ | $y_0$ | $y_0$ | y      | $y_0$ | $y_0$ | $y_0$ |
| z      | $z_0$ | $z_0$ | $z_0$ | z      | $z_0$ | $z_0$ | $z_0$ |

a)      b)

Since  $A < B$ ,  $B < C$  and  $C < A$ , Invariant 2 stops any rule from committing.

Figure 11. Livelock avoidance

one is easy, especially given the Urgency List. One can simply delete the least urgent rule that breaks the cycle and schedule the rest.

In the multi-cycle execution model, the problem is slightly harder because such cyclically dependent rules may not all begin at the same time. For example, as shown in 11b) the rules  $B$  and  $C$  may already be executing when one has to decide whether to activate rule  $A$ . If  $A$  is activated it will lead to a livelock. This is the motivation underlying Invariant 3 which prohibits such an activation of rule  $A$ . It is worth noting that Invariants 1 and 3 guarantee that active rules at any time form a partial order with respect to the sequential composability ( $<$ ) relation. Consequently, it is always possible to pick an active rule which can be committed before all others without violating atomicity.

Our Invariant 3 is not the most permissive invariant because it may prevent some legal schedules. For example, in Figure 9 it is possible to fire rule  $A$  in cycles 2 and 4, resulting in the execution  $A, A, B, A, A, B$ . We will not explore such aggressive schedules in this paper.

## 4.2 Scheduling Algorithm

We now present the procedures to generate the  $rsrv_A$  and  $commit_A$  signals based on the invariants we have discussed in Section 4.1. Both procedures rely on the compiler computed conflict properties for each pair of rules. The RuleSelectReserve procedure generates the  $rsrv$  signals and requires the  $sA$  and  $rdy$  input signals while the RuleSelectCommit procedure generates the  $commit$  signals and requires the  $rdyCommit$  signals as well as the set of active rules generated by RuleSelectReserve.

In order to set  $rsrv_A$  at time  $t$  we need to make sure that 1.  $rdy_A(t)$  is true; 2.  $A$  is not already active; and 3.  $A$  does not conflict with the rules that have already been reserved. The rules are examined in the decreasing order of urgency. The following algorithm builds the set of active rules  $AR$  and a set of non-active rules  $NA$ :

### Procedure RuleSelectReserve:

1. Initially  $AR = \{A \mid sA_A(t)\}$  and  $NA = \mathcal{R} - AR$
2.  $\forall A \in AR : rsrv_A(t) = false$
3. repeat while  $NA \neq \emptyset$
- 3.1 Let  $A$  be the most urgent rule in  $NA$ :  $\nexists B \in NA : B \rightarrow A$
- 3.2  $rsrv_A(t) = rdy_A(t) \wedge (\bigwedge_{B \in AR} \neg((A <> B) \vee (A < B)))$
- 3.3  $NA = NA - \{A\}$
- 3.4 if  $rsrv_A(t)$  then  $AR = AR \cup \{A\}$

Step 3.2 of the algorithm above preserves Invariants 1 and 3. As a consequence the rules in  $AR$  will form a partial order over the sequentially composable ( $<$ ) relation.

In order to compute  $commit_A(t)$  we need to make sure that 1.  $active_A(t)$  is true; 2.  $rdyCommit_A(t)$  is true and 3. committing  $A$  would not violate Invariant 2. The algorithm uses the set  $AR$  computed above and the set  $CR$  which is the set of rules which are active and ready to commit. It examines the rules in  $CR$  in an order defined by sequential composability ( $<$ ) and decides for each rule whether it should be allowed to commit:

### Procedure RuleSelectCommit:

1. Initially  $CR = \{A \mid active_A(t) \wedge rdyCommit_A(t)\}$ ;  
 $AR = \{A \mid active_A(t)\}$
2.  $\forall A \in \mathcal{R} - CR : commit_A(t) = false$
3. Repeat while  $CR \neq \emptyset$
- 3.1 Select  $A \in CR$  such that  $\nexists B \in CR : B < A$
- 3.2  $commit_A(t) = \nexists B \in AR : (B < A)$
- 3.3  $CR = CR - \{A\}$
- 3.4 if  $commit_A(t)$  then  $AR = AR - \{A\}$

It is important to note that this algorithm is well defined (deadlock free) only if the ( $<$ ) relation is a partial order on  $CR$ . This is not true in general but is true for active rules because of Invariants 1 and 2 and because  $CR$  is a subset of  $AR$ .

These algorithms can be turned into pure combinational logic for a given set of rules because ( $<>$ ) and ( $<$ ) relation among rules are known at compile time. We will illustrate this in the next section using the GCD example.

We will like to note one point about the efficiency of RuleSelectCommit procedure before we leave this section. Even though the RuleSelectCommit procedure is efficient – it generates its output in a single pass over the set  $CR$  – it can potentially generate a lot of combinational logic because of step 3.1. This step essentially requires a topological sort of any subset of rules based on the ( $<$ ) relation. This inefficiency can be overcome by a version of the algorithm that sometimes may not commit as many rules every cycle as theoretically possible.

The simplification of the algorithm is based on the fact that the rules in step 3.1 can be examined in any order, that is, we can ignore the ( $B < A$ ) check without the loss of correctness. So instead of checking rules according to the ( $<$ ) relation, we can check the rules in the urgency order, which is very efficient to implement in hardware.

## 4.3 The GROC Scheduler for GCD

The GCD example in Section 3 uses Bluespec methods, which our scheduler does not accommodate. In order to use this example here, we modified the module to produce test data and print results. We present the complete example in Figure 12. The complete module consists of three rules:  $GCD$ ,  $Seed$ , and  $Print$  (abbreviated to  $G$ ,  $S$  and  $P$ ). Those three rules have the following interactions:  $G <> S$ ,  $P < G$  and  $P < S$ . Of course, the two rules which update their data are also self-conflicting:  $G <> G$  and  $S <> S$ .

During creation of the  $rsrv$  and  $commit$  signals for the module, we have to take care to respect the Invariants, as described in Section 4.2.  $rsrv$  signals for  $G$  and  $S$  have to respect Invariant 1 ( $G <> S$ ), while  $P$  has to respect Invariant 3 ( $P < S$  and  $P < G$ ).  $commit$  signals for  $G$  and  $S$  rules have to respect Invariant 2 ( $P < S$  and  $P < G$ ).

## 5 Implementation and Results

We have implemented our algorithm in a compiler (bsc2) which compiles BSV into Verilog. We rely on the commercial Bluespec compiler to provide as input an internal representation (IR) which corresponds roughly to BTRS. As our technique does not yet support separate module compilation, we require that the modules be flattened, something

```

Module doGcd2
  Register x = 1, y = 0, a = 1, b = 1, t
  Rule GCD :: (x := y | (t := %01(x, y) ;
                y := %02(x, y, t))) when y ≠ 0
  Rule Seed :: (x := a | y := b | a := a + b | b := a)
                when y = 0
  Rule Print :: $display(a + "%" + b + " = " + x)
                when y = 0

Urgency List: Print → GCD → Seed

```

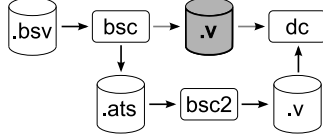
**Figure 12. Complete GCD Example**

```

rsrvG(t) := rdyG(t) ∧ ¬sAG(t) ∧ ¬sAS(t)
rsrvS(t) := rdyS(t) ∧ ¬sAS(t) ∧ ¬activeG(t)
rsrvP(t) := rdyP(t) ∧ ¬sAP(t) ∧ ¬(sAG(t) ∨ sAS(t))
commitG(t) := activeG(t) ∧ rdyCommitG(t) ∧
              ¬(activeP(t) ∧ ¬commitP(t))
commitS(t) := activeS(t) ∧ rdyCommitS(t) ∧
              ¬(activeP(t) ∧ ¬commitP(t))
commitP(t) := activeP(t) ∧ rdyCommitP(t)

```

**Figure 13. Complete GCD Control and Scheduling Equations**



Our compiler, bsc2, replaces the grayed out file from bsc.

**Figure 14. Tool flow**

| num cycles | GCD     | CPU w/ DIV |
|------------|---------|------------|
| 1          | 12.44ns | 25.11ns    |
| 2          | 7.32ns  | 13.61ns    |
| 4          | 4.28ns  | 6.54ns     |
| 8          | 2.42ns  | 3.62ns     |
| 16         | 1.45ns  | 3.55ns     |

**Figure 15. Synthesis Results**

the Bluespec compiler will do. Our tool flow is depicted in Figure 14.

Our compiler first analyzes the IR and inserts latches on long paths in rule bodies. It also generates an Urgency List, and builds a scheduling circuit for the module being compiled. The generated Verilog corresponds directly to the input IR. When no multi-cycle rules are created, the performance of the resulting circuits is a few percent worse than those produced by the commercial Bluespec compiler. Our compiler allows inserting Verilog code as macros, thus allowing us to produce output from `$display` statements and include basic libraries such as FIFOs. Including these libraries is dangerous because the compiler cannot guarantee safety of multi-cycle calls in black box macros. We have compiled two designs with our compiler, simulated its execution and verified correctness of the results. We used standard CAD tools to synthesize and place-and-route the RTL. We list our results in Figure 15. To test our compiler we compiled two designs using our compiler and synthesized

them using Synopsys Design Compiler version V-2004.06-SP2 with TSMC 130nm cell libraries.

The first design is the GCD example used throughout this paper. We defined the registers to be 16 bits wide and produced compiled versions with varying number of stages in the *GCD* rule. The version with a single-cycle *GCD* rule resulted in a cycle time of 12.44ns. A version with a 16-cycle *GCD* rule resulted in a cycle time of 1.45ns for a speedup of 8.6 times. While the overall delay of computing the GCD has not dropped due to our compiler, we have achieved the goal of increasing the clock speed. As long as the design using our GCD uses it sparingly, the overall circuit should benefit from an improved cycle time.

The other design we compiled was a pipelined 4-stage CPU with full bypassing. We have built two versions of this CPU: one with a DIV instruction and one without. We compiled the design without the DIV instruction without inserting any multi-cycle rules, and obtained a cycle time of 3.53ns. We then added a rule and instruction decode logic to handle the DIV instruction. We compiled the same design with the division rule taking 1, 2, 4, 8 and 16 cycles to complete. Single-cycle division caused the clock cycle to jump to 25.11ns. 16-cycle division reduced the clock cycle to 3.55ns, which is essentially the same as the original design without a division instruction. For the 16-cycle division the area increased by 51%. This was rather expected since we started with a rather small design and more than doubled the number of registers in it. No modification of the source Bluespec files was necessary. Our CPU maintains correct behavior even though different instructions take different numbers of cycles.

In addition to the positive synthesis results, we have also observed that Design Compiler takes significantly less time to compile designs with fast multi-cycle rules. We believe that this is because short paths between registers are easier to optimize. This may be of some importance, as CAD tools typically are a bottleneck in the design cycle.

## 6 Related Work

Our algorithm has some similarities with the scoreboard scheduling technique used in CDC 6600[14]. The traditional scoreboard deals with a dynamic stream of instructions which share physical resources and which have some sequentiality constraints based on data dependencies. The rules with their atomicity and resource constraints can be thought of as a fixed set of instruction that need to be scheduled (dispatched) each cycle because the execution of a rule potentially affects the availability of rules for scheduling in the next cycle. Conservative scheduling in both cases means scheduling fewer entities.

Transactional Memories [3, 4, 8, 9, 13] deal with atomic updates and are conceptually closest to the problems discussed in this paper. However, the current TM systems assume that the read set and write set (the domain and range) of a transaction cannot be statically estimated by a compiler. Neither do they assume how long it will take to compute a transaction. Thus, the TM systems deal with a much more difficult problem which requires shadow state and difficult

computation involving intersection and union of read sets and write sets. However, *our work can be viewed as an extremely efficient implementation for small transactions.*

## 7 Summary and Future Work

In many designs, we have faced the situation where an infrequently executed rule or action dominates the cycle time. The multi-cycle rule synthesis technique presented in this paper provides a convenient way for the designer to try out the alternative of turning a long combinational circuit into a multi-cycle circuit with minimal or no source modification. Using our experimental compiler we have shown that by letting a potentially slow action compute over several cycles, we can improve the clock speed of the design to the point where other, much faster, paths dominate the design. This can result in significantly better circuits if the slow rules represent uncommon cases in actual execution.

There are several extensions that are possible for this work. Indeed some of them are essential for the technique to deal with realistic designs. The most obvious need is for modular synthesis. Bluespec designs of any consequence incorporate separately synthesized modules. Rosenband [12] had introduced a technique for separate synthesis of composable modules. We intend to develop a similar system for multi-cycle modules to enable linking of libraries and to broaden the family of circuits we can synthesize. This extension is not as straightforward as one might assume because it requires a significantly more complicated call protocol and because guards may be evaluated over multiple cycles as a consequence.

Rosenband [11] introduced a GAA scheduling technique which enables firing multiple instances of the same rule in a single cycle. Dave [5] introduced the idea of creating schedules by rule composition. Both these approaches are orthogonal to ours, as they attempt to lock down the schedule of execution within a clock cycle. It would be interesting to study the specification of schedules that can guarantee some performance in the presence of multi-cycle rules.

We would also like to study the use of pipelining to permit multiple instances of the same multi-cycle rule to be active concurrently. Similarly, we would like to study the automatic folding of a computation to use fewer gates (e.g., fold many subtract/compare units in a divider into a single repeated stage and a counter).

Finally, we would like to extend our technique to allow rules with dynamically varying computation delays. This will allow us to make use of the fact that often the length of computation is data dependent. An efficient scheduling algorithm to do so would also allow us to produce efficient asynchronous circuits. [6]

On the theoretical side, we would like to develop a formal proof of correctness for our algorithm.

## Acknowledgments

We are very thankful to the National Science Foundation for supporting this work (#CCF-0541164). We also want to thank Nirav Dave and Ravi Nanavati for helpful comments in preparing the final draft of this paper.

## References

- [1] Lennart Augustsson, Jacob Schwarz, and Rishiyur S. Nikhil. Bluespec Language definition, 2001. Sandburst Corp.
- [2] Bluespec Inc. <http://www.bluespec.com>.
- [3] Chi Cao Minh, Martin Trautmann, JaeWoong Chung, Austen McDonald, Nathan Bronson, Jared Casper, Christos Kozyrakis, and Kunle Olukotun. An effective hybrid transactional memory system with strong isolation guarantees. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*. Jun 2007.
- [4] Luis Ceze, James Tuck, Josep Torrellas, and Calin Cascaval. Bulk disambiguation of speculative threads in multiprocessors. In *ISCA '06: Proceedings of the 33rd annual international symposium on Computer Architecture*, pages 227–238, Washington, DC, USA, 2006. IEEE Computer Society.
- [5] Nirav Dave, Arvind, and Michael Pellauer. Scheduling as rule composition. In *MEMOCODE '07: Proceedings of the 5th IEEE/ACM International Conference on Formal Methods and Models for Code-sign*, pages 51–60, Washington, DC, USA, 2007. IEEE Computer Society.
- [6] A. Davis and S.M. Nowick. An Introduction to Asynchronous Circuit Design. Technical Report UUCS-97-013, Computer Science Department, University of Utah, September 1997.
- [7] M. Dubois, C. Scheurich, and F. Briggs. Memory access buffering in multiprocessors. In *ISCA '86: Proceedings of the 13th annual international symposium on Computer architecture*, pages 434–442, Los Alamitos, CA, USA, 1986. IEEE Computer Society Press.
- [8] M. Herlihy, V. Luchangco, M. Moir, and W. Scherer. Software transactional memory for dynamic-sized data structures, 2003.
- [9] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the Twentieth Annual International Symposium on Computer Architecture*, 1993.
- [10] James C. Hoe and Arvind. Synthesis of Operation-Centric Hardware Descriptions. In *Proceedings of ICCAD'00*, pages 511–518, San Jose, CA, 2000.
- [11] Daniel L. Rosenband. Hardware synthesis from guarded atomic actions with performance specifications. In *ICCAD*, pages 784–791, 2005.
- [12] Daniel L. Rosenband and Arvind. Modular scheduling of guarded atomic actions. In *DAC '04: Proceedings of the 41st annual conference on Design automation*, pages 55–60, New York, NY, USA, 2004. ACM.
- [13] Nir Shavit and Dan Touitou. Software transactional memory. In *Symposium on Principles of Distributed Computing*, pages 204–213, 1995.
- [14] J. E. Thornton. Parallel Operation in the Control Data 6600. *AFIPS Proc. FJCC*, Pt. II, Vol. 26:33–40, 1964.