Hardware Synthesis from Guarded Atomic Actions with Performance Specifications

Daniel L. Rosenband and Arvind
MIT CSAIL
{danlief, arvind}@csail.mit.edu

ABSTRACT

We present a new hardware synthesis methodology for guarded atomic actions (or rules), which satisfies performance-related scheduling specifications provided by the designer. The methodology is based on rule *composition*, a transformation which is well understood in the TRS theoretical framework; however, previous rule composition approaches, when applied to realistic hardware designs resulted in an explosion of the number of rules during synthesis. Using a new state element, the *Ephemeral History Register*, we show that we can recreate the effect of specified rule compositions without actually generating the composed rules. We demonstrate the approach via a small circuit example (GCD) and then show its impact on the methodology to implement pipelined processors in Bluespec. The results show simultaneous improvements in performance over previous rule-based synthesis approaches and the ease of expressing several performance-related concepts, such as *bypassing*. In a somewhat surprising result, we show that simply by specifying a different schedule, one can transform a single-issue processor pipeline into a superscalar pipeline automatically.

1. INTRODUCTION

Some performance guarantees in digital design are as important as correctness in the sense if they are not met we don't have an acceptable design. Suppose we have a pipelined processor which executes programs correctly but its various pipeline stages cannot fire concurrently because of some ultraconservative interlocking scheme. We are unlikely to accept such a design. In the reorder buffer (ROB) of a modern 2-way superscalar processor, the designer may not feel that the design task is over until the ROB has the capability of inserting two instructions, dispatching two instructions and writing-back the results from two functional units every cycle[5]. Even simple micro-architectures (and not just related to processors) can present designers with such performance-related challenges[1]. It is important to understand that such requirements emanate from the designer of the micro-architecture as opposed to some high-level specification of the design. To that extent, only the designer can provide such specifications. The main contributions of this paper are a new methodology in the context of rule-based synthesis that provides the designer the ability to specify such performance related goals, and a synthesis procedure using *Ephemeral History Registers* (EHR) to meet those goals.

Rule-based synthesis (e.g., Bluespec) has been quite successful in providing the designer a methodology and a synthesis tool that eliminates functional bugs related to complicated race conditions in designs[2]. It provides the designer a simple model to reason with about the correctness of his/her design. The model says that all legal behaviors can be explained in terms of some sequential and atomic firing of rules on the state elements. The Bluespec synthesis tool has also demonstrated that it generates RTL that is comparable in quality, i.e., area and time, to hand-coded Verilog[1, 4]. Bluespec relies on sophisticated scheduling of rules to achieve these goals. However, when the high-level performance goals of a designer are not met then an understanding of the schedule generated by the Bluespec compiler becomes imperative on the part of the designer before he or she can make improvements. There were also some fundamental limitations in Bluespec's orignial scheduler, which Hoe & Arvind[7] imposed to avoid long combinational paths. Bluespec Inc has incorporated RWire and other ingenious solutions to get around some of these limitations. Regardless of how well these solutions work, they require additional effort and may raise the barrier to using Bluespec.

Recently, Rosenband has shown how a new hardware element, the *Ephemeral History Register* (EHR), can be used in place of an ordinary register to implement scheduling constraints[9] in rule-based synthesis. EHR's provide the new capability of forwarding values created in a rule to the follow-on rules which may be scheduled in the same clock cycle. Without such a capability it is impossible to model bypassing or to make pipelines where various stages are connected by single-element FIFO's without stepping outside the TRS model. This paper improves on this work in two ways. It provides a systematic explanation of EHR in terms of *rule composition*, an idea that was explored by Lis[8]. It also shows via analytically and implementation results that a slightly more general version of the synthesis algorithm for EHR's works well in practice. We also derive a superscalar version of a single-issue processor description via scheduling specifications and show that EHR's offer a practical solution to implementing Lis' TRS transformation.

Paper Organization: In Section 2, we review the execution model of guarded atomic actions and introduce the idea of rule composition. In Section 3, we describe how performance guarantees may be specified via schedules. We also show how a new schedule specification may lead to transforming a single-issue pipeline into a multi-issue superscalar pipeline. Section 4 presents a new synthesis framework that through rule composition allows the designer to generate high-quality and well-performing circuits. In Section 5, we report experimental results, which show that the new synthesis procedure indeed does better than the current procedure and meets the performance guarantees. We end with brief conclusions in Section 6.

2. UNDERSTANDING SCHEDULING AS RULE COMPOSITION

This section reviews the execution model of atomic actions and explains scheduling in terms of rules composition.

2.1 Guarded Atomic Action Execution Model

Each guarded atomic action (or rule) consists of a body and a guard. The body describes the execution behavior of the rule if it is enabled. The guard (or predicate) specifies the condition that needs to be satisfied for the rule to be executable. We write rules in the form:

rule R_i: when $\pi_i(s)$ ==> $s := \delta_i(s)$

Here, π_i is the predicate and $s := \delta_i(s)$ is the body of rule R_i. Function δ_i computes the next state of the system from the current state s. The execution model for a set of such rules is to non-deterministically pick a rule whose predicate is true and then to atomically execute that rule's body. The execution continues as long as some predicate is true:

```
while (some $\pi$ is true) do
    1) select any $R_i$ , such that $\pi_i(s)$ is true
    2) $s := \delta_i(s)$
```

The base-line synthesis approach generates combinational logic for each rule's predicate (π) and each rule's state update function (δ). A scheduler then chooses one of the rules whose predicate is true and updates the state with the result of the corresponding update function (δ). This process repeats in every cycle. Hoe and Arvind's synthesis strategy uses more sophisticated scheduling than the one described above but does it in manner that does not introduce any new behaviors[7]. It is based on Conflict Free (CF) and Sequential Composition (SC) analysis of rules. Two rules R_1 and R_2 are CF if they do not read or write common state. In this case, whenever enabled, both rules can execute simultaneously and their execution could be explained as the execution of R_1 followed by R_2 or vice versa. Two rules R_1 and R_2 are SC if R_1 does not write any element that R_2 reads. The synthesis procedure ignores the updates of R_1 on those elements which are also updated by R_2 and generates a circuit that behaves as if R_1 executed before R_2. One thing to notice is that beyond a possible MUX at the input to registers concurrent scheduling of CF and SC rules does not increase the combinational path lengths and hence the clock cycle of a design.

In many designs aggressive CF and SC analysis is sufficient to uncover all, or at least the desirable amount of concurrency in rule scheduling. However, there are situations when one wants to schedule a rule that may be affected (even enabled) by a previous rule in the same cycle. Bypassing or value forwarding is a prime example of such situations; a rule, if it fires, produces a value which another follow on rule may want to use at the same time the value is to be stored in some register. Capturing this type of behavior is beyond CF and SC analysis. We first explain this idea via rule composition.

2.2 Rule Composition

A fundamental property of TRS's is that if we add a new rule to a set of rules it can only enable new behaviors; it can never disallow any of the old behaviors. Furthermore, if the new rule being added is a so called *derived rule* then it does not add any new behaviors[3, 11]. Given two rules R_a and R_b we can generate a *composite rule* that does R_b after R_a as follows:

$R_{a,b}$: when ($\pi_a(s)$ & $\pi_b(\delta_a(s))$) => $s := \delta_b(\delta_a(s))$

It is straightforward to construct the composed terms $\pi_b(\delta_a(s))$ and $\delta_b(\delta_a(s))$ when registers are the only state-elements and there are no modules. We illustrate this by the following two rules that describe Euclid's GCD algorithm, which computes the greatest common divisor of two numbers by repeated subtraction:

R_{sub}: when ((x > y) & (y != 0)) => x := x – y;
R_{swap}: when ((x <= y) & (y != 0)) => x, y := y, x;

Given these two rules, we can derive a new "high performance" $R_{swap,sub}$ rule that immediately performs a subtraction after a swap. We name the values written by R_{swap}, as x_{swap}' and y_{swap}':

let x_{swap}' = y; y_{swap}' = x; *in*
$R_{swap,sub}$: when ((x <= y) & (y != 0) &
(x_{swap}' > y_{swap}') & (y_{swap}' != 0)) =>
x, y := x_{swap}' – y_{swap}', y_{swap}';

After substitution this rule is equal to the following rule:

$R_{swap,sub}$: when ((x <= y) & (y != 0) & (y > x) & (x != 0)) =>
x, y := y – x, x;

Since the $R_{swap,sub}$ rule was formed by composition it can safely be added to the GCD rule system. We can then generate a circuit for the three rules: R_{sub}, R_{swap} and $R_{swap,sub}$ using CF and SC analysis, giving preference to the $R_{swap,sub}$ rule when it is applicable. This circuit performs better than the original rule system which only contained R_{sub} and R_{swap} since it allows both the swap and subtraction to occur within a single cycle. (Though the motivation is different this optimization has similarities with loop unrolling in behavioral compilers[6].) Without composition, CF and SC analysis would not have been able to derive this parallelism and one of the two rules would have executed each cycle. In the evaluation section we discuss the impact of this rule composition on area, cycle time and overall performance.

Mieszko Lis wrote a source-to-source TRS transformation system to compose rules and applied it to a number of designs including a pipelined processor[8]. His system produced new rules by taking a cross product of all the rules in a system and filtered out those composite rules that were "uninteresting" in the following sense. Composition of R_1 followed by R_2 was considered uninteresting if either (i) it showed that R_2 could not be enabled after R_1 executed or (ii) if R_1 and R_2 were already CF or SC. In the latter case the scheduler would have scheduled them concurrently any way and a new rule was not required. Lis was able to generate all the interesting composite rules and by applying it to a simple processor pipeline's rules was able to automatically generate all the rules for a 2-way superscalar version of the processor. He was further able to show the robustness of his transformation (and filtering) by applying the transformation again to the generated 2-way rules to produce the rules for a 4-way superscalar micro-architecture. What is fascinating about this work is that it is based purely on the semantics of TRS's and does not use any knowledge specific to processor design.

The biggest problem in exploiting Lis' transformation is that in spite of his filtering of "uninteresting" composite rules, the compiler can generate a large number of new rules. He reports that the number of rules increased from 13 for the single issue pipeline to 74 for 2-issue, 409 for 3-issue, 2,442 for 4-issue and 19,055 for 5-issue pipeline[8]! These numbers reflect filtering out 24% to 41% of the possible composite rules.

2.3 Rule Composition Using Conditional Actions

We now introduce an alternative method for rule composition that subsumes many natural behaviors of subsequences of rules firing.

This method makes use of conditional actions in rule generation and avoids the explosion in the number of new rules generated. Conditional actions permit us to combine many rules in one which can dramatically speed up the synthesis process even in the current scheduling scheme used in the Bluespec compiler.

Conditional actions such as "if q then a" execute action *a* only if condition *q* is true. The simplest form of action *a* is an assignment of a value to a register; more complex actions are represented by action method calls of a module[10]. Such action method calls may have implicit conditions, which can affect the firing of the rule calling the method. Now, consider the following rule where a_1 and a_2 represent actions:

```
R:  when p => a₁ ; if q then a₂
```

This rule can be understood as the composite of the following two mutually exclusive rules:

```
R₁ :  when p & q  => a₁ ; a₂
R₂ :  when p & !q => a₁
```

This transformation is always correct but may cause some subtle problems in a modular compilation flow when actions have implicit conditions. For example, one gets slightly different semantics depending on if the implicit conditions of a_2 are conjoined to just R_1 or both R_1 and R_2.

Using conditional actions we can generate a composite rule that executes R_b after R_a as follows:

```
$R_{a,b}$:  when (True)  =>
    $t_a$ = if $\pi_a(s)$ then $\delta_a(s)$ else s ;
    $t_b$ = if $\pi_b(t_a)$ then $\delta_b(t_a)$ else $t_a$ ;
    s := $t_b$
```

In the above code *s* contains state elements and t_a, t_b, should be read as temporary local variables (not registers) whose values are visible only within the rule and not maintained across cycles. This new rule has the advantage that it behaves as rule R_a if rule R_b does not get enabled; behaves as rule R_b if rule R_a does not get enabled and behaves as R_a followed by R_b if R_a is enabled and that in turn enables R_b.

Using this method, the composition of the swap with sub rule in GCD will result in the following:

```
$R_{swap,sub}$':  when (True) =>
   if (x <= y) & (y != 0) then $t_x$ = y, $t_x$ = x;
   if (tx > $t_y$) & ($t_y$ != 0)) then  $t_x$' = $t_x$ - $t_y$, $t_y$' = $t_y$;
   x := $t_x$', y  =  $t_y$'
```

The difference between $R_{swap,sub}$ and $R_{swap,sub}$' given earlier is that this rule subsumes the functionality of the two rules which were used to compose it. The EHR based synthesis scheme, which we present in Section 4 behaves very similarly to such conditional rules.

3. SPECIFYING SCHEDULES FOR A PIPELINED PROCESSOR

Figure 1 shows a 4-stage pipeline for a toy processor which has only two instructions Add and Jz (Branch on zero). The stages are connected by FIFO buffers *bF*, *bD* and *bE*. In addition to the usual *enq, deq, first* methods, the FIFOs also support the *clear* method to kill all the entries in the FIFO. The *bD* and *bE* FIFOs also provide a *bypass* method to search the FIFO for a particular destination register and return the associated value (note: do to space limitations we do not provide the entire code for the *bypass* logic). The processor has a total of 7 rules: *F* fetches an instruction and puts it in *bF*; *D_add* and *D_jz* decode the first instruction in *bF* and fetch the operands either from the register file or the bypass path and enqueue the decoded instruction into *bD*; the *E* rules execute the first instruction in *bD* and either enqueue the results in *bE* or, in case of a branch taken, clear the fetched instructions from *bF* and *bD*; *WB* writes back the value in the register file. In Figure 2 we give two implementations of the FIFO, one with a single element and another one which can contain up to two elements.

For this processor pipeline to work properly, it is important that the single element FIFO be able to *enq* and *deq* in a single cycle, otherwise at best alternate pipeline stages will operate concurrently. We also expect that rules that *deq* a FIFO should appear to execute before the rules that *enq* into the same FIFO (otherwise values could fly through the FIFO without ever getting latched – clearly not the intent of a pipeline stage). Similarly, for values to be bypassed from the execute stage to the decode stage the execute stage rule should appear to take effect before the decode stage rules fetch their operands. Based on these observations a designer may want to specify the following schedule:

Schedule1:   WB rule x E rules x D rules x F rule

This schedule says: take a rule each from every group of rules (e.g. WB, Eadd, D_jz, F) and execute them in one cycle giving the effect of WB, followed by E_add, followed by D_jz, followed by F. It is as if we want to combine all the rules in a particular order and produce a gigantic rule that makes all the stages move like a synchronous pipeline. Additionally, if any of the stages cannot execute, for example due to a stall condition, then if possible, the remaining subset of rules should continue to execute. In the next section we present an algorithm to achieve this effect in synthesis without actually generating the big rule.

For the sake of modularity we also want our design to work if we replace the one-element FIFO's with the two-element FIFO's. Assuming we have a two-element FIFO, consider the following schedule:

Schedule2:   WB x WB x E x E x D x D x F x F

This schedule says write back two instructions one after another, execute two instructions one after another, decode two instructions one after another and fetch two instructions one after another – all in one clock cycle. This is precisely the way a two-way superscalar processor is supposed to function. It should not come as a surprise that if the machine has to actually behave like a two-way issue machine then it would need more resources. Indeed we would see that implementing this schedule would require more interfaces on the FIFO's and register files and, if sufficient storage in the form of registers is not provided, the design will result in modules whose methods may not be enabled properly.

There are situations where one may want the system to behave either as R1 x R2 or as R2 xR1 depending upon some other predicate. An important aspect of the algorithm that we present in the next section is that it can compose multiple rules according to different schedules and then combine these composite rules into one design by using the normal rule based synthesis.

```
function stall(src1) =
        if (bD.bypass(find_dest, src1))  return true;
        else                             return false;
function bypassv(src1) =
        if ((bE.bypass(find_match, (EVal src1)))
                return bE.bypass(find_val, (EVal src1));
        else    return rf.read(src1);
F:  when (true) =>
        bF.enq(imem[pc]);
        pc <= pc + 4;
D_add: when (bF.first() == (Add rd ra rb)) &
                (!stall(ra)) & (!stall(rb)) =>
        bD.enq(EAdd rd bypassv(ra) bypassv(rb));
        bF.deq();
D_jz: when (bF.first() == (Jz cd addr)) &
                (!stall(cd)) & (!stall(addr)) =>
        bD.enq(EJz bypassv(cd) bypassv(addr));
        bF.deq();
E_add: when (bD.first() == (EAdd rd va vb)) =>
        bE.enq(EVal rd (va + vb));
        bD.deq();
E_jz_taken: when ((bD.first() == (EJz cd av)) & (cd == 0))          =>
        pc <= av;
        bF.clear();
        bD.clear();
E_jz_nottaken: when ((bD.first() == (EJz cd av))&(cd != 0)) =>
        bD.deq();
WB: when (bE.first() == (EVal rd vr)) =>
        rf.write(rd, vr);
        bE.deq()
```

Figure 1: 4-Stage Processor

One Element FIFO:

```
enq x =     data.write x; full.write 1        when (full.read == 0)
deq =       full.write 0;                     when (full.read == 1)
first =     return data.read;                 when (full.read == 1)
bypass f v = return f(data.read, v);          when (full.read == 1)
```

Two Element FIFO:

```
enq x =  data_1.write x
            if (full_0.read == 0) then data_0.write x;
            full_1.write (full_0.read); full_0.write 1;
when (full_1.read == 0)
deq =       full_0.write (full_1.read)
            full_1.write 0;
            when (full_0.read == 1)
first =     return data_0.read; when (full_0.read == 1)
```

Figure 2: FIFO Implementations

4. COMPOSITION USING THE EHR

The Ephemeral History Register was introduced by Rosenband to provide greater control over scheduling of rules [9]. It provides new scheduling capabilities that cannot be achieved using just SC and CF analysis. We will first review the EHR's functionality and then show how the EHR can be used directly to exploit the new style of composed rules. The innovative part of the EHR synthesis scheme is that it actually never generates the composite rules --- given the specification of a schedule it generates annotations on each method call and these annotations are further propagated inside modules until we reach registers, which are then replaced by EHR's.

4.1 The Ephemeral History Register

The Ephemeral History Register (EHR) (see Figure 3) is a new primitive state element that supports the forwarding of values from one rule to another. It is called Ephemeral History Register because it maintains a history of all writes that occur to the register within a clock cycle. Each of the values that were written (the history) can be read through one of the read interfaces. However, the history is lost at the beginning of the next cycle. We refer to the superscript index of a method as its version. For example, *write*^{2} is version *2* of the *write* method. Each write method has two signals associated with it: *x*, the data input and *en*, the control input that indicates that the *write* method is being called and must execute to preserve rule atomicity. A value is not written unless the associated *en* signal is asserted.

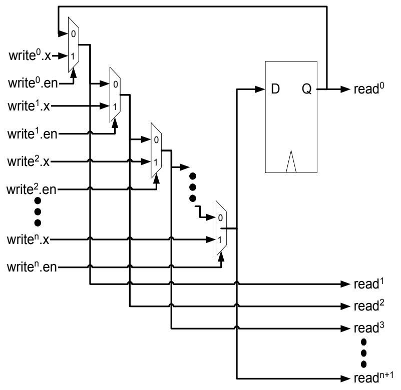

Figure 3: The Ephemeral History Register

It is clear that we can use the EHR in place of a standard primitive register element by replacing calls to the register *read* and *write* methods with calls to the EHR *read*^{0} and *write*^{0} methods. These interfaces behave exactly as those of a normal register if none of the other interfaces are being used. We can also use the EHR to create composed rules.

4.2 Composition Using EHR

Before explaining how to use the EHR to generate circuits that behave like composed rules we examine the requirements imposed by our approach. Suppose we are given rules R_1 and R_2 and want to achieve the effect of the composed rule $R_{1,2}$. We replace rules R_1 and R_2 by rules R_1' and R_2' such that rule R_1' behaves the same as R_1 in isolation, i.e. when rule R_2' does not execute (and similarly for R_2'). However, when R_1' and R_2' both execute, then the behavior of the two rules executing should be the same as that of $R_{1,2}$. Clearly, if R_1 and R_2 do not access common state, then R_1' and R_2' are equivalent to the original rules. However, if they do access common state, then reads and writes must satisfy the constraints in Figure 4. For a given state element with initial value v_0, the table specifies which values the rules must observe when reading the state element, and what element the state element should take after the rules have executed. We assume that R_1 writes value v_1 and R_2 writes value v_2 (which may be dependent on v_1). The table makes clear that R_2' must observe any values that R_1' writes, and the final value must reflect the "last" rule that writes it. The last two table entries correspond to the execution of $R_{1,2}$.

R_1' executes	R_2' executes	R_1' writes	R_2' writes	R_1' reads	R_2' reads	Final value
Yes	No	No		v_0		v_0
Yes	No	Yes		v_0		v_1
No	Yes		No		v_0	v_0
No	Yes		Yes		v_0	v_2
Yes	Yes	No	No	v_0	v_0	v_0
Yes	Yes	No	Yes	v_0	v_0	v_2
Yes	Yes	Yes	No	v_0	v_1	v_1
Yes	Yes	Yes	Yes	v_0	v_1	v_2

Figure 4: Composition Requirements

The procedure uses EHR's to satisfy the requirements in Figure 4:

1) Replace all registers accessed by R_1 and R_2 with EHR's.
2) Replace all *read* / *write* in R_1 by calls to $read^0$ / $write^0$.
3) Replace all *read* / *write* in R_2 by calls to $read^1$ / $write^1$.

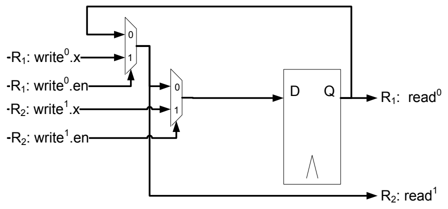

Figure 5: Two Rule Composition

The resulting EHR circuit is shown in Figure 5. Each of the rules R_1' and R_2' execute individually as before. However, when executing together they exhibit the behavior of the composed rule $R_{1,2}$. What makes this possible is that the EHR circuit allows rule R_2' to observe the values that are written by R_1'. When R_1' does not execute ($write^0.en$ is 0), and the EHR returns the current state of the register to R_2' ($read^1$). However, when R_1' does execute and writes a value to the register ($write^0.en$ is 1), then the value that the R_2' read interface ($read^1$) returns is the value that was written by R_1' ($write^0.x$). Such forwarding of values from one rule to another was not possible before the EHR was introduced.

This procedure can be generalized in a straightforward way to generate the composition of rules R_0, R_1, R_2, R_3, … R_n so that it appears as if the rules execute in the listed order. In almost all cases, the designer will also want all subsets of these rules to be composed in the same order. We can achieve this effect by replacing all read and write method calls in R_i by calls to $read^i$ and $write^i$ and by using a EHR with enough ports. This procedure works for the same reasons that it works in the case of two rules -- a "later" rule in the composition order observes, via forwarding, any values that the next earliest rule writes.

4.3 Pruning and Other Optimizations

The above algorithm does not always generate the optimal circuit (in terms of area and timing). For example, suppose R_3, as part of a sequence R_0, R_1, R_2, R_3, is the only rule to access a register reg_{only3}. The algorithm turns reg_{only3}. into an EHR and provides R_3 access to it via interfaces $read^3$ and $write^3$. However, since none of the other rules access the ports 0, 1, or 2 of the register reg_{only3} it is wasteful to have R_3 tap the register at such a high version number. It could simply have accessed the register through the $read^0$ and $write^0$ interfaces. Thus, after each call to label the methods we should also call the PRUNE procedure which eliminates "gaps" in EHR references:

```
PRUNE($R_0$, $R_1$, …, $R_n$) =
  1) access = { $reg_i$ | $reg_i$ is read or written in one of $R_0$, …, $R_n$}
  2) for $i$ = n downto 0 do
       foreach $r \in access$ do
         if (r.$read^i$ and r.$write^i$ are unused) then
           decrement all access $r.read^j$ to $r.read^{j-1}$ for $j > i$
           decrement all access $r.write^j$ to $r.write^{j-1}$ for $j > i$
```

4.4 Modular Composition

This section presents a new modular compilation algorithm for rule based designs. It takes as input a modular design with scheduling constraints and produces a new design that is functionally equivalent and is guaranteed to satisfy the scheduling requirements. Each scheduling constraint C takes the form S_0 x S_1 x S_3 x …, where each S_j is a set of rules. We apply composition to module interface methods the same way as to rules. This gives us interface methods which can be composed to satisfy a constraint. Note: EHR is the key replacement that allows arbitrary composition of rules that access registers. Other state elements such as memories can be extended straightforwardly to provide EHR like read and write interfaces.

```
PROPCONSTRAINTS(C) =
  $mToSched = \varnothing$ ;
  foreach $S_i \in C$ do    /** schedule the rule (methods) in C */
    foreach $R_j \in S_i$ do
      $mToSched = mToSched \cup$ CREATECOMPOSABLE($R_j$, $i$);
  PRUNE({R | R $\in$ C});

  /*** construct interface requirements for modules ***/
  foreach module $m \in mToSched$ do
    $S_0 = \varnothing$ ;$S_1 = \varnothing$ ;$S_2 = \varnothing$ ; …
    foreach $\underline{m.g^i} \in mToSched$ do
      $S_i = S_i \cup m.g^i$;
    $C_m = S_0$ x $S_1$ …;
    PROPCONSTRAINTS($C_m$);  // recursively schedule each module


CREATECOMPOSABLE($R$, $i$) =
  $mToSched = \varnothing$ ;              // set of methods to schedule
  R' = R;                              // create a copy of R
  foreach method call $m.g$ in $R'$ do
    $R' = R'$ [$m.g^i$ / $m.g$];         // rename method calls
    if $m$ is not an EHR then          // EHR's do not recurse
      $mToSched = mToSched \cup m.g^i$;
  return $mToSched$;
```

A final step in giving the designer complete flexibility is to allow many sequences of rules to be composed. For example, the designer may want three composition sequences to be generated: (i) R_0, R_1, R_2; (ii) R_2, R_3; and (iii) R_3, R_0. The most straightforward way to accomplish this is to create copies of rules that occur in multiple sequences and to then call the PROPCONSTRAINT procedure on each sequence. After PROPCONSTRAINT completes we construct a circuit and scheduler for the design using the normal Bluespec synthesis. This combines composition sequences as well as rules that were unconstrained.

5. RESULTS

We evaluated the new synthesis methodology to confirm that it produces functionally correct results, the performance meets the designer's expectations, and the final circuit quality remains high. To implement the new flow we created the EHR state-element in Verilog and imported it, along with its interface scheduling properties into Bluespec. We then created the designs using registers as the only primitive state elements, i.e. FIFO's, RF's, etc. were created in Bluespec from registers only. We then transformed the design into a new design according to the procedure outlined in Section 4 for each scheduling requirement. The resulting design was then fed through the Bluespec compiler to produce RTL Verilog, which was then synthesized using Synopsys Design Compiler to generate area and timing numbers for the TSMC 0.13µm G process. We generated area and timing numbers for two different timing constraints to illustrate numbers for an area and a timing-constraint synthesis run. We also simulated each design to measure functional performance.

Figure 6 shows the results for GCD designs to meet 3 different scheduling constraints. The first design is the original design and does not incorporate any transformations. The second design composed R_{swap} x R_{sub}, and the third design was scheduled to satisfy the constraint: R_{swap} x R_{sub} x R_{swap} x R_{sub}. As is expected, as more rules are composed, fewer cycles are required to compute results. Similarly, the critical path increases as more rules are composed. Allowing the same rule to execute multiple times within a cycle also increases the area, as is seen with the last constraint since all associated logic need to be duplicated.

GCD Input	Measure	No Constr.	R_{swap} X R_{sub}	R_{swap} X R_{sub} X R_{swap} X R_{sub}
Input 1	cycles	91	78	39
Input 2	cycles	117	101	51
10ns constr. Area (µm^2)		5221	6479	13705
10ns constr. Timing (ns)		10	10	10
5ns constr. Area (µm^2)		5909	9003	26638
5ns constr. Timing (ns)		4.54	5.00	5.3

Figure 6: GCD Results

Figure 7 shows the results for a 4-stage processor pipeline. In Addition to an unconstrained design (the traditional Bluespec flow), two more designs, one each corresponding to the Schedules 1 and 2 (see Section 3), were studied. Additionally, all designs were synthesized using one and two element FIFO's as pipeline stages. A simple benchmark loop with additions, and jumps was run on all designs. The unconstrained implementation, as expected, performs poorly as only alternating stages execute in each cycle. The 2 x superscalar transformed design does not execute twice as fast as the design according to Schedule 1 because of branches in the instruction stream. The superscalar design is larger since logic needs to be replicated. We were somewhat surprised by the timing result for the superscalar 2 element FIFO implementation. It seems the critical path is Branch compare, branch compare, fetch pc + 1, fetch pc + 1, i.e., – 2 32-bit compares in sequence with two increments. One would expect such long combinational paths once in a while because of the specified schedule. It would be easy to fix this problem, once identified. We did not try that yet. However, all these designs were very easy to generate – only difference is the constraints – so this is a quick way to implement and get feedback.

Design	Benchmark (cycles)	Area 10ns (µm^2)	Timing 10ns (ns)	Area 2ns (µm^2)	Timing 2ns (ns)
1 element fifo:					
No Constr.	18525	24762	5.85	26632	2.00
Compose	11115	25094	6.83	33360	2.00
2 x Super	11115	25264	6.78	34099	2.04
2 element fifo:					
No Constr.	18525	32240	7.38	39033	2.00
Compose	11115	32535	8.38	47084	2.63
2 x Super	7410	45296	9.99	62649	4.72

Figure 7: 4-Stage Processor Results

6. CONCLUSION

We have presented a new synthesis algorithm for guarded atomic actions that provides important advantages over previous approaches. The key advantages are that it allows a designer to specify scheduling constraints and the compiler automatically generates circuits that satisfy these constraints. Results show that the desired concurrency is achieved and that the synthesis algorithm works. Once this level of concurrency is achieved, the designer can focus on pathologically long combinational paths if any and try different architectural fixes. Additionally, we have shown that using a super-scalar transformation we can achieve desired concurrency without the rule explosion which previously was required for such transformations.

7. REFERENCES

[1] Arvind, Nikhil, R.S., Rosenband, D.L. and Dave, N., High-level Synthesis: An Essential Ingredient for Designing Complex ASICs. in *ICCAD*, (San Jose, 2004).

[2] Arvind and Shen, X. Using term rewriting systems to design and verify processors. *Micro, IEEE*, *19* (3). 36-46.

[3] Baader, F. and Nipkow, T. *Term Rewriting and All That*. Cambridge University Press, Cambridge, UK, 1998.

[4] Bluespec, I. and Systems, I. Benchmarking of Bluespec Compiler Uncovers No Compromises in Quality of Results (QoR) *www.bluespec.com/images/pdfs/InterraReport042604.pdf*, 2004.

[5] Dave, N., Designing a Reorder Buffer in Bluespec. in *MEMOCODE*, (San Diego, 2004).

[6] Gupta, S., Dutt, N., Gupta, R. and Nicolau, A., SPARK: a high-level synthesis framework for applying parallelizing compiler transformations. in *VLSI Design, 2003. Proceedings. 16th International Conference on*, (2003), 461-466.

[7] Hoe, J.C. and Arvind Operation-centric hardware description and synthesis. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, *23* (9). 1277-1288.

[8] Lis, M.N. Superscalar Processors via Automatic Mircoarchitecture Transformations, Massachusetts Institute of Technology, Cambridge, MA, 2000.

[9] Rosenband, D.L., The Ephemeral History Register: Flexible Scheduling for Rule-Based Designs. in *MEMOCODE*, (2004).

[10] Rosenband, D.L. and Arvind, Modular Scheduling of Guarded Atomic Actions. in *Proceedings of the 41st Design Automation Conference (DAC)*, (2004).

[11] Terese *Term Rewriting Systems*. Cambridge University Press, Cambridge, UK, 2003.