Hardware Synthesis from Guarded Atomic Actions with Performance Specifications

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*, and relies on the fact that a rule derived by the composition of two rules behaves as if the two rules were scheduled simultaneously. Rule *composition* is a well understood transformation in the TRS theoretical framework; however, previous rule composition approaches resulted in an explosion of the number of rules during synthesis, making them impractical for realistic designs. We avoid this problem through *conditional composition* of rules which generates one rule instead of 2^n rules when we combine n rules. We then show how this conditional composition of rules can be compiled into an efficient hardware structure which introduces new but *derived interfaces* in modules. 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* or how branches are dealt with in the pipeline. In a somewhat surprising result, we show that simply by specifying a different schedule, one can automatically transform a single-issue processor pipeline into a superscalar pipeline. Scheduling specifications for performance opens up a new and rich avenue for architectural exploration.

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[4]. 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 and they should be a core component of any high-level synthesis flow.

The synthesis framework that our performance specifications apply to are rule-based languages, e.g., Bluespec. They provide the designer a simple model to reason with about the correctness of his/her design and have been quite successful in providing the designer a methodology and a synthesis tool that eliminates functional bugs related to complicated race conditions[2]. 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, 3].

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. This can be a challenging process and due to limitations in the original Bluespec scheduler cannot always be resolved without reverting to unsafe solutions, such as Bluespec Inc's RWire.

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[8] in rule-based synthesis. This paper improves on this work in three ways: (i) it presents an algorithm that derives an EHR like hardware structure for registers and modules based on the semantics of derived rules that use conditional actions, (ii) it provides a more general EHR-based synthesis algorithm for modular designs and designs that require multiple constraints to be satisfied simultaneously, and (iii) it demonstrates the practicality of the approach via a careful analysis of the circuits and schedules of a processor example. These contributions lead to a design environment which dramatically eases architectural exploration.

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 rule 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 π is true) do
    1) select any Ri , such that πi(s) is true
    2) s := δi(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[6]. 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[2, 9]. 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:

```
Rsub:   when ((x > y) & (y != 0))     => x := x – y;
Rswap:  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 xswap' = y;  yswap' = x; in
  Rswap,sub :  when ((x <= y) & (y != 0) &
                        (xswap' > yswap') & (yswap' != 0)) =>
                  x, y := xswap' – yswap', yswap';
```

After substitution this rule is equivalent to the following rule:

```
Rswap,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[5].) Without composition, CF and SC analysis would not have been able to derive this parallelism and only one of the two rules would have executed each cycle. (Later, 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[7]. 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". Lis' system 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[7]! These numbers reflect filtering out 24% to 41% of the possible composite rules. Although interesting from a theoretical viewpoint, this methodology is clearly not practical to generate hardware. We will show how to generate circuits for these thousands of derived rules without actually having to generate them.

2.3 Composition Using Conditional Actions

We now introduce conditional actions as an alternative method for rule composition. Conditional actions in rule generation subsume many natural behaviors of subsequences of rules firing, thereby dramatically reducing the number of rules that are generated during composition. Later, in Section 4 we show how to generate efficient circuits from these rule compositions based on conditional actions.

An example of the problem that conditional action address is the $R_{swap,sub}$ rule that we provided earlier. This rule only covered the case when both R_{swap} and R_{sub} rules were both applicable. As an alternative, consider the following rule based on conditional actions, where the meaning of "$" is that the actions following the "$"see the effect of actions before the "$".

```
Rswap&sub:  when (True)  =>
   if ((x <= y) & (y != 0))  then x, y := y, x; $
   if ((x > y) & (y != 0))   then x := x – y;
```

With appropriate renaming we can derive the following rule after eliminating the "*$*" ($x^0$ and y^0 refer to the initial value of *x* and *y*, respectively):

$R_{swap\&sub}$: when (True) =>
$x^1, y^1 = ((x^0 <= y^0) \& (y^0 != 0))$ **?** y^0, x^0 **:** x^0, y^0 ;
$x^2, y^2 = ((x^1 > y^1) \& (y^1 != 0))$ **?** $x^1 - y^1, y^1$ **:** x^1, y^1;
$x, y := x^2, y^2$

This new rule has the advantage that it behaves as rule R_{swap} if rule R_{sub} does not get enabled; it behaves as rule R_{sub} if rule R_{swap} does not get enabled and behaves as R_{swap} followed by R_{sub} if R_{swap} is enabled and that in turn does not disable R_{sub}. Hence, based on conditional actions, we have generated a single rule that behaves as three rules using traditional composition. In general, for *n* rules, this approach introduces at worst a linear number of additional rules, whereas traditional composition introduces an exponential number of new rules during composition.

For circuit generation, the key insight here is that x^0, x^1, x^2, y^0, y^1, y^2, represent different versions of the state variables *x* and *y* within the same clock period. These versions are related to each other by cascading conditions and combinational logic which is derived semantically from the application of the rules chosen for composition:

$x^1 = R_{swapEN}$ **?** y^0 **:** x^0;
$x^2 = R_{subEN}$ **?** $x^1 - y^1$ **:** x^1;
$y^1 = R_{swapEN}$ **?** x^0 **:** y^0;
$y^2 = y^1$;

Rosenband's Emphemeral History Registers (EHR)[8] provides a perfect hardware structure to capture this idea. We show the EHR circuit for the two rule composition case in Figure 1 below.

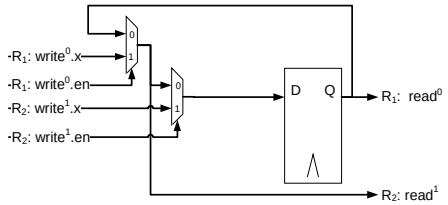

Figure 1: Two Rule Composition

The above rule examples only interact with registers. However, the notion of conditional actions, and hence the EHR style hardware structure naturally extends to modules and their interface methods. For example, two interface methods *m.g* and *m.h* are conditional methods that satisfy *m.g $ m.h* if their behavior can be explained as (i) *m.g* if only *m.g* is enabled, (ii) *m.h* if only *m.h* is enabled, and (iii) *m.g* followed by *m.h* if both methods are enabled.

Before showing how we can generate this style of conditionally composed modular circuits, we present a more realistic design in the next section. This will make the challenge of modular composition and synthesis in the presence of multiple performance constraints more clear.

3. SPECIFYING SCHEDULES FOR A PIPELINED PROCESSOR

Figure 2 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*, *clear*, and *first* methods, 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: due 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 3 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 $ E rules $ D rules $ 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. Using conditional rules we will be able to achieve the effect of all subsets of these rules without actually generating the subset rules.

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 $ WB $ E $ E $ D $ D $ F $ 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.

```
function stall(src) =
        if (bD.bypass(find_dest, src))  return true;
        else                            return false;
function bypassv(src) =
        if ((bE.bypass(find_match, (EVal src))) then
                return bE.bypass(find_val, (EVal src));
        else    return rf.read(src);
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)) =>
        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 2: 4-Stage Processor

One Element FIFO:

```
enq x       =  data :=x; full := 1;  when (full == 0)
deq         =  full := 0;            when (full == 1)
clear       =  full := 0;
first       =  return data;          when (full == 1)
bypass f v  =  return f(data, v);    when (full== 1)
```

Two Element FIFO:

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

Figure 3: FIFO Implementations

4. COMPOSITION USING THE EHR

The Ephemeral History Register was introduced by Rosenband to provide greater control over scheduling of rules [8]. 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. Each of these renamed rules corresponds to one of the conditional actions we have previosuly mentioned.

4.1 The Ephemeral History Register

The Ephemeral History Register (EHR) (see Figure 4) 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 or index. 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.

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.

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 5.

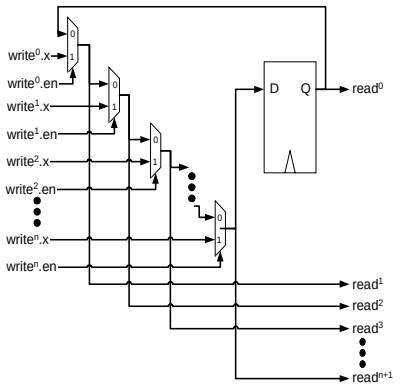

Figure 4: The Ephemeral History Register

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 5: Composition Requirements

We can use EHR's to satisfy the requirements in Figure 5:

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 $read^1$ / $write^1$.

The resulting EHR circuit is shown in Figure 1. 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. Effectively we have generated the conditional rule:

```
R1,2:  when (True)  =>
    t1 = R1(s);
    t2 = R2(t1);
    s := t2
```

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 index 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 \$ S_1 \$ S_3 \$ …, where each S_j is a set of rules. As previously described, 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. Below we present the PROPCONSTRAINTS algorithm, which transforms the design to satisfy the constraint C.

```
PROPCONSTRAINTS(C) =
  mToSched = ∅ ;
  foreach Si ∈ C do   /** schedule the rule (methods) in C */
    foreach Rj ∈ Si do
      mToSched = mToSched ∪ CREATECOMPOSABLE(Rj, i);
  PRUNE({R | R ∈ C});

   /*** construct interface requirements for modules ***/
  foreach module m ∈ mToSched do
    S0 = ∅ ;S1 = ∅ ;S2= ∅ ; …
    foreach m.g^i ∈ mToSched do  // compose methods
      Si = Si ∪ m.g^i;
    Cm = S0 x S1 …;
    PROPCONSTRAINTS(Cm); // recursively schedule each module


CREATECOMPOSABLE(R, i) =
   mToSched = ∅ ;              // 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
     // recurs if conditional interfaces are not provided
     if m does not provide conditional interfaces then
       mToSched = mToSched ∪ m.g^i;
   return mToSched;
```

4.5 Modular Composition Example

The PROPCONSTRAINTS procedure transforms rules (and methods) so that they satisfy a scheduling constraint. The procedure creates conditionally composed module interface methods with new version numbers and alters rules (and methods) to make calls to these new methods. This section examines what these version numbers mean in the context of modules.

As previously mentioned, the scheduling constraint for a 4-stage processor pipeline, excluding the jump taken rule is:

WB x {E_add x E_jz_nottaken} x {D_add, D_jz} x F

From these constraints, the PROPCONSTRAINTS procedure derives the following constraints for the FIFO interface methods:

bF: {first^0, deq^0} \$ enq^1
bD: {first^0, deq^0} \$ enq^1 \$ bypass^2
bE: {first^0, deq^0} \$ enq^1 \$ bypass^2

A reasonable jump taken rule constraint is:

WB \$ E_jz_taken

From this constraint the PROPCONSTRAINTS procedure derives the following FIFO interface method constraints:

bF: clear^0
bD: first^0 $ clear^1
bE: {first^0, deq^0}

Let us first examine what the new FIFO interface means if we only satisfy one constraint, e.g. *{first^0, deq^0} $ enq^1 $ bypass^2*. The behavior of the action methods (*enq* and *deq*) can be explained by conditional composition, where *t^i*'s represent the conditional values that result from the MUX structure:

t^{-1} = s;
if (m.deq^0.en) **then** t^0 = m.deq(s); **else** t^0 = t^{-1};
if (m.enq^1.en) **then** t^1 = m.enq(t^0, m.enq^1.x); **else** t^1 = t^0;
s = t^1

The two read methods (*first* and *bypass*) return values based on the temporary variables in this expression:

first^0 : **return** first(t^{-1});
bypass^2: **return** first(t^1);

This new FIFO interface has the effect of performing the composition of the interface methods if they are simultaneously enabled, e.g. if *first^0*, *deq^0*, *enq^1*, and *bypass^2* are all called, then the behavior is as though *first^0* and *deq^0* execute, followed by *enq^1* (which observes state changes that are made by *deq^0*), followed by *bypass^2* (which observes state changes made by both *deq* and *enq*). If only a subset of the methods execute, we still obtain the correct compositional behavior. For example, if *deq^0* and *bypass^2* execute (and not *enq^1*), then *bypass^2* directly observes the state that *deq^0* produces (*deq^0* produces *t^0*, bypass observes *t^1*, and since *enq* is not executing: *t^1 = t^0*). Thus, the behavior of a module interface with a single constraint is clear.

4.6 Multi-Constrained Modular Composition

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. Although this method always produces correct circuits, it can introduce critical paths that the designer might not have intended. We can illustrate this problem via the processor example from the previous sub-section in which we had one set of constraints that did not contain the branch taken rule, and one constraint that does contain the branch taken rule. If we call the PROPCONSTRAINTS procedure for each constraint and then merge the resulting interfaces, we obtain the following conditional FIFO methods:

bF: {first^0, deq^0, clear^0} $ enq^1
bD: {first^0, deq^0} $ {enq^1, clear^1} $ bypass^2
bE: {first^0, deq^0} $ enq^1 $ bypass^2

As an example of an unindent combinational path, values might be passed from *clear^0* to *enq^1* in the *bF* FIFO. Although functionally correct, this solution could produce a design with unsatisfactory cycle time. One option is to have a scheduler disallow both of these methods from being called concurrently, and marking the path as a *false-path*. However, an alternate solution that better fits a conventional synthesis flow exists.

The solution is to produce "separate" interfaces for the different constraint groups. In the above example, this would result in the following interfaces:

bF: {first^{a0}, deq^{a0}} $ enq^{a1} | clear^{b0}
bD: {first^{a0}, deq^{a0}} $ enq^{a1} $ bypass^{a2} | first^{b0} $ clear^{b1}
bE: {first^{a0}, deq^{a0}} $ enq^{a1} $ bypass^{a2} | first^{b0} $ deq^{b0}

An example circuit for a register with the following interface is shown below:

EHR_split: {read^{a0}, write^{a0}} **$** {read^{a1}, write^{a1}} **$** {read^{a2}, write^{a2}} **|**
{read^{b0}, write^{b0}} **$** {read^{b1}, write^{b1}} **$** {read^{b2}, write^{b2}}

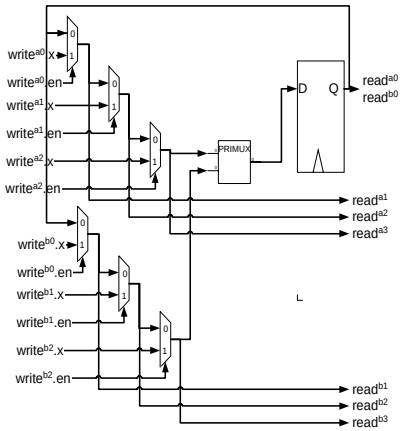

Figure 6: EHR with Split

Note: values are only forwarded from *write^{a*}* to *read^{a*}* and from *write^{b*}* to *read^{b*}*, and not from *write^{a*}* to *read^{b*}* or *write^{b*}* to *read^{a*}*. It should be clear that as with the EHR, this "split" structure can be generated for arbitrary conditional method interfaces.

5. RESULTS AND EVALUATION

We evaluated the new synthesis methodology to confirm that it produces functionally correct results, that the performance meets the designer's expectations, and that 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-constrained 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} $ R_{sub}, and the third design was scheduled to satisfy the constraint: R_{swap} $ R_{sub} $ R_{swap} $ 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 needs 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 7: GCD Results

Figure 8 shows the compilation results for a 4-stage processor pipeline. In addition to an unconstrained design (the traditional Bluespec flow), a composed design which behaves like a standard pipeline (see Schedule 1 in Section 3), and a superscalar design (see Schedule 2 in Section 3) were studied. For the composed design we show that it is easy to study the behavior (both cycle count and cycle time) of different pipelines by simply changing the design's schedule. In the case of the superscalar design we examine why the cycle time at first does not match expectations and what simple changes can be made to dramatically improve the performance of the superscalar design.

We synthesized the designs using one and two element FIFO's as pipeline stages since a two element FIFO is required for a superscalar implementation to perform well. A simple benchmark loop with arithmetic operations and conditional branches was run on all designs. As a reference we show timing numbers for some of the key processor components in Figure 9. These numbers are approximate since each synthesis run selects slightly different implementations. However, it is clear that unless we further pipeline the design, no design can have a cycle time of much less than 1.6ns since we must sequentially get the decode FIFO output (Clk to Q – about 0.3ns), pass through an adder in the execute stage (about 0.9ns), pass through at least one level mux (0.3ns) and then satisfy setup time (0.1ns).

Design	Bench. (cycles)	Area 10ns (μm^2)	Timing 10ns (ns)	Area 1ns (μm^2)	Timing 1ns (ns)
1 element fifo:					
No Constr.	18525	24762	5.8	33073	1.6
Comp J $ F	9881	25362	7.5	34161	2.2
Comp DFJ	9881	25180	8.0	34896	2.6
Comp J | F	11115	25001	6.6	34511	1.9
2 x Super	11115	25264	6.8	36037	1.9
2 element fifo:					
No Constr.	18525	32240	7.4	39033	1.9
Comp J | F	11115	32535	8.4	47084	2.63
2 x Super	7410	45296	10.0	62649	4.7
2 x Super+Fixes	7410	40180	9.9	62053	3.0

Figure 8: 4-Stage Processor Results

As expected, the unconstrained implementation performs poorly since the standard Bluespec compiler can only schedule alternating stages to execute in each cycle. However, the cycle time is close to optimal. The composed pipeline is more interesting since it allows us to experiment with several interesting schedules. Depending on the schedule we choose, we observe varying cycle counts and cycle times. We should note that the only change we made to the composed designs during these experiments is that we changed the scheduling constraints. We did not change the underlying code. The algorithms we presented in earlier sections ensure that correctness of the designs is maintained in this process.

Component	Propagation Delay
32 bit addition	0.9ns
32 bit increment	0.6ns
32 bit compare to 0	0.6ns
2-1 MUX (32 bits wide)	0.3ns
Clk to Output + Setup Time	0.4ns

Figure 9: Component Delays

The three composed designs that we examines were: (i) *J $ F*: the jump taken appears to execute before the fetch rule. This means that the fetch rule uses the branch target in the same cycle that the branch is resolved. The branch resolution followed by fetch becomes the critical path. (Note: *J* refers to the branch taken rule) (ii) *D $ F $ J*: We move the branch taken rule to the "end of the cycle". This eliminates the critical path from jump taken through fetch. However, this means that the branch taken observes the results of the decode stage – effectively we have moved the branch resolution into the decode stage. Hence the critical path becomes: execute an add instruction, bypass it into the decode stage and compare it with 0 to see if the branch is taken. This is a long critical path, but is a design used in many processors. Finally, another common pipeline design (iii) is *J | F*: this is effectively splitting the access to the PC (see section 4.6) and ensures that fetch and jump do not execute simultaneously (fetch cannot observe the branch target in the cycle that the branch is resolved). This eliminates the critical path from the first case but in turn has a slightly higher cycle count since branch taken and fetch cannot execute in the same cycle. Clearly there are trade-offs with all three of these designs. This high-level scheduling mechanism provides a very simple tool to experiment with the different pipelines and measure the impacts on cycle time and cycle count.

As was mentioned earlier, we can apply the same scheduling algorithms to generate a superscalar design. The results in Figure 8 show that we obtain benchmark cycle count improvements by switching to the superscalar design. This is expected since in many cases each stage can execute two instructions per cycle. However, the performance is only about 33% below that of the standard composed pipeline (case iii) because the pipeline is cleared after each branch taken – this has a larger relative penalty in the superscalar design than in the composed design. Somewhat disturbingly, the cycle time for the superscalar design is more than twice that of the single element FIFO composed design (4.7ns vs. 1.9ns). In an optimal implementation we would expect the superscalar design to have a cycle time of only slightly more than the composed pipeline (about two MUX stages, or about 0.6ns worse). Below we discuss several simple changes we can make to the circuit generation and the FIFO implementation to reduce the superscalar cycle time from 4.6ns to 3.0ns (about 0.5ns within optimal). Note: This is the only design for which we altered code to improve cycle times – all other designs were directly derived from the original processor code and transformed using the conditional composition algorithms.

The first change is a simple circuit transformation shown in Figure 10. This is a simple logic transformation that Synopsys design compiler currently does not perform, but which is easy to

add to the Bluespec compilation. In this case, the *Bz1_taken* signal is on the critical path. In the original design (on the left side of the figure) the next PC computation for the second fetch in the superscalar fetch stage cannot be computed until the earlier branch is resolved. By simply moving logic across the MUX we can improve this path.

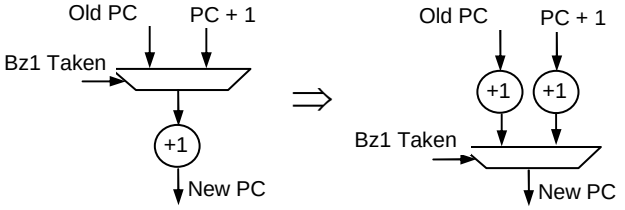

Figure 10: Moving Logic Across a Mux

A more interesting change that had a dramatic impact on the cycle time of the superscalar design is that we slightly changed the two element FIFO specification. These changes do not alter the behavior of the FIFO, but imbed high-level logic that we have about the FIFO into its circuits. For example, we know that after dequeueing from the FIFO twice, it will be empty. Since the write back stage in the superscalar design will always execute twice if the FIFO contains two valid elements (and once if it contains only one element), the execute stage does not need to check that the FIFO between the write back and execute stages is empty. Such a check can add one or two MUX's to the critical path (0.6ns). We can achieve this effect by rewriting the *enq* method as follows (the changes to this method are highlighted in italics):

```
enq x =  data_1.write x
            if (full_0 == 0) then data_0 := x;
            full_0(1);
            if (full_0 == 1) then   full_1 := 1;
            else  full_1 := 0;
      when ((full_1 == 0) || (full_0 == 0))
```

Clearly, these changes do not alter the behavior of the design: We know that if *full_0* is 0, then *full_1* is also always 0, so it is safe to add the check of *(full_0 == 0)* to the method's implicit condition. Similarly, we can write the value 0 to *full_1* if the FIFO is empty and we are enqueueing a value since the value will be placed in the "0" slot. Although these changes do not change the functionality, they have the impact of allowing constants to be effectively propagated through the pipeline – for example after this change, the execute stage logic is optimized via constant propagation to no longer need to check if the FIFO it is enqueueing in is full.

Another example of this type of change is to the FIFO *clear* method. Again we highlight the change in italics. Obviously, the data values can have any value after the FIFO is cleared. However, by setting the *data_0* value during a *clear* method call to the value it would have after a *deq*, the logic that reads from the FIFO can be optimize: regardless of what the "first" rule in a stage does (*deq* or *clear*), it always moves *data_1* into *data_0*, so the "second" rule to execute always knows what the "new" value in *data_0* will be and hence can directly look at the *data_1* register. Again, by simply adding this line of code which clearly doesn't change FIFO functionality we embed some high-level knowledge into the design. The result is that a MUX stage for one of the FIFO's is removed from the critical path. Note: this optimization works in the processor execute stage where the "first" execute rule always executes. However, this optimization does not improve timing for the decode rules because the "first" decode rule might stall.

```
clear = full_0 := 0;  full_1 := 0;
        data_0 := data_1;
      when (true)
```

These types of changes allowed us to reduce the cycle time from 4.6 to 3.0ns. The remaining 0.5ns can be obtained through similar changes but they become counterintuitive since one needs to keep track of when data is available and how mux's are introduces. Instead, at that point it would be more reasonable to rewrite the design as a superscalar design. It is important to recognize that a decision to rewrite the design with "superscalar" in mind is not due to a short-coming in the synthesis methodology that we present here. As designers we simply have high-level knowledge that the compiler does not have. Without this knowledge, the compiler must be conservative. An interesting future approach to this work might be to use user-assertions to guide the compilation process. For example, an assertion could be added that if FIFO slot "0" is empty, then FIFO slot "1" is also empty.

6. CONCLUSION

We presented a new synthesis algorithm for guarded atomic actions based on conditional rule composition and analyzed the efficiency of such an approach on several designs. We leveraged previous research on design transformation through rule composition as well as the EHR to create a practical framework that has a well founded theoretical foundation, but is also practical in that it eliminates the rule explosion that previously was required for such transformations. Our algorithms create efficient implementations that satisfy multiple performance constraints for both rules and modules. The experimental results show that interesting architectures can be rapidly generated by simply changing scheduling constraints. Additionally, many of the resulting designs have efficient circuits. In the cases where circuits are non-optimal, the designer can usually use high-level knowledge to make minor changes to the design to achieve the expected circuit timing.

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

[3] Bluespec, Inc., Benchmarking of Bluespec Compiler Uncovers No Compromises in Quality of Results (QoR) *www.bluespec.com/images/pdfs/InterraReport042604.pdf*

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

[5] 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.

[6] 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.

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

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

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