A new synthesis procedure for atomic rules containing multi-cycle function blocks

Michal Karczmarek
NVIDIA Corporation (michal@karcza.com)

Arvind, Muralidaran Vijayaraghavan
MIT (arvind, vmurali@csail.mit.edu)

Abstract—A new method for hardware synthesis from atomic rules where rules can take unknown number of cycles is presented. Some complex functions, especially the ones involving data-dependent control, are more easily expressed as loops and take much less area when implemented as multi-cycle folded circuits. Multicycle rules also provide a high-level method for the designer to deal with the timing-closure problem by treating a long combinational path as a multicycle path. Our synthesis procedure uses minimal extra storage and executes all rules eagerly. It resets all those rules whose read sets are affected by a committing rule. It also makes use of rule reservations to avoid a short rule from resetting a long rule repeatedly. This technique automatically takes advantage of different timings of different conditional branches. Our syntax-directed synthesis procedure composes signals indicating when a computation is done and when inputs to a computation have changed. Preliminary results from our implementation look very promising.

I. INTRODUCTION

Guarded Atomic Actions (GAA) is an untimed model for expressing concurrent computations. However, when GAA is used as a model for hardware synthesis, it is assumed that each rule executes in one clock cycle [1]. This assumption has proven to be very useful in practice because it provides the designer a method for what to do in case a design does not meet the timing. The designer essentially looks for rules with long combinational paths and decomposes them into multiple rules, where each rule has a smaller combinational path. For example, pipelined designs are often expressed as multiple rules where each rule corresponds to a stage in the pipeline. One high-level method to meet timing in such designs is to split a pipeline stage into two or more stages. Powerful as this method is it does entail additional verification work to ensure that the overall design still has the desired functionality [2].

Is there an alternative? In [3] Karczmarek and Arvind presented a hardware synthesis technique for GAA whereby an infrequently taken long-latency path does not determine the overall circuit timing. The essential idea in their paper is to treat a long-latency combinational function as a multi-cycle function and then schedule the atomic rules embodying such combinational functions in a manner where one pays the penalty of longer delays only when the function is actually used. Even when the multi-cycle function is actually used, the effect of latency may be mitigated by the fact that other “non conflicting” rules can be executed concurrently with the slower rule. GAA semantics plays a crucial role because functionalities of such systems are invariant under different timing assumptions. This paper is about generalizing the multi-cycle rule synthesis by allowing combinational functions whose time to compute depends upon the input. The synthesis procedure presented here is syntax directed and quite different from the Karczmarek-Arvind (KA) procedure [3].

It is important to understand the difference between hardware synthesis from GAA and other untimed models such as C, C++, SystemC etc. There is a rich body of literature (see, for example, [4]–[12]) as well as number of commercial tools (see, for example, [13]–[15]) for hardware synthesis from imperative descriptions of algorithms. The goal of such tools is to derive a suitable microarchitecture for a sequential algorithm, to meet some performance goal and under some cost (area and timing) assumptions about hardware. For example, the tool, with some user guidance, chooses the appropriate number of computing elements and pipeline stages etc. In contrast GAA descriptions express microarchitectures, although at a very high-level. A user has much more control over synthesized hardware in GAA synthesis than in C-based synthesis. It is because of this lack of control over synthesized hardware that sometimes GAA or Verilog synthesis produces significantly more efficient hardware than C-based synthesis [16].

Our multicycle synthesis still provides a great degree of control over generated hardware to the user. For example, we provide a way for the user to insert delays in expression evaluation. Thus delay(e) takes one cycle longer than just computing e but it has no effect on the results or the semantics of the atomic actions embodying e. Delays can be inserted by the designer or by a tool that estimates the time it takes to compute a function. Conditional expressions, combined with delays, let the designer specify mutually exclusive paths with widely varying latencies. While-loops allow the designer to express certain functions in a more direct imperative style. For synthesizing loops we introduce registers to hold the values of circulating variables until the while-loop terminates. Thus insertion of delays is only for hardware efficiency and does not exacerbate the verification problem.

The techniques for the synthesis of while-loops into FSMs are well known and have been implemented in numerous behavioral synthesis tools [13]–[15]. What is new in our technique is how to do so within the framework of atomic actions. Our method draws a distinction between state elements (e.g. registers) that represent architectural state and those that hold temporary values which do not carry over from one rule execution to the next. Changes in the architectural state always correspond to some sequential execution of atomic
rules. Without such a distinction one would have to create
shadow registers or imitate locks to avoid reading a register
before it contains the right value and writing a register before it
has been read by its consumers. These atomicity considerations
make the loop synthesis problem quite different from the
traditional loop synthesis in C-like languages.

This paper makes the following contributions:

1) A new way of expressing atomic rules which include
multi-cycle functions. It can be easily incorporated as
an extension in existing languages like Bluespec System
Verilog (BSV) [17].
2) A new synthesis procedure for a set of multi-cycle
atomic rules incorporating combinational functions with
data-dependent computation time.
3) Preliminary results showing that our technique takes
advantage of different timing in different branches of
conditionals. The results also show that expressing an
iterative computation as a loop provides significant area
savings over explicitly unrolling the same computation.

In contrast to the earlier work, our synthesis procedure
is syntax directed and compositional. We also show that
our procedure, after optimization, produces exactly the same
hardware as the current BSV synthesis tool for single-cycle
rules [1]. Unlike KA, our procedures also includes Ephemeral
History Registers (EHRs) [18], [19], which are very important
for expressing efficient designs.

Paper Organization: In Section II, using examples, we
present BMC, a kernel language of atomic actions where
an action can take multiple clock cycles. In Section III we
review the KA synthesis procedure [3] because our procedure
uses a key ideas from the KA scheduling. In Section IV
we present the details of the syntax directed translation of
BMC into hardware circuits. In Section V we introduce EHRs
and show how they can be incorporated in our synthesis.
In Section VI we give some informal arguments why our
synthesis procedure preserves one-rule-at-a-time semantics.
In Section VII we show synthesis results from our preliminary
implementation of the algorithm in Section IV. In Section VIII
we present some of the related work and finally we discuss
our conclusions in Section IX.

II. BMC: A LANGUAGE OF ATOMIC ACTIONS WITH
MULTI-CYCLE FUNCTIONS

We extend the simple language of atomic actions of Dave et
al [20] to let the users write multi-cycle rules explicitly. The
grammar of the language (BMC) is given in Figure 1 where
multi-cycle rules are written using delay or a while-loop. We
have included EHRs and associated operators (x.r0, x.r1, x.w0,
x.w1) in the grammar though we will discuss EHRs only in
Section V. To simplify the presentation we have eliminated
nested modules and allowed guards only at the top level in a
rule. We take some liberties with this syntax and use tuples
like (e1, e2) and function expressions like f(e) in our
examples with the understanding that the tuples and function
are always inlined by the compiler and play no role in the
synthesis procedure. We explain the language by writing the

Module main
[Register r v]
[EHR x v]
[t = e]
[Rule R a when e]
a ::= r := e || x.w0 := e || x.w1 := e
|| if e then a
|| a | a
|| (t = e in a)
e ::= r || c || t || x.r0 || x.r1
|| op (e, e)
|| (t = e in e)
|| if e then e else e
|| delay(e)
|| (while e do
[ t = e ]
return t)

Fig. 1: The Grammar of BMC

Module gcd
Register x, y
Rule GCD
(if (x ≥ y) then x := x - y
  else (x := y | y := x))
when (y ≠ 0)
method start(a, b) =
  (x := a; y := b) when (y = 0)
method result() = x when (y = 0)
endmodule

function gcd(x,y) =
  while (y ≠ 0) do
    (x,y) = (if (x ≥ y) then (x - y, y)
             else (y, x))
  return x

(a) GCD module

(b) GCD function

Fig. 2: GCD in two different ways

GCD in two different ways as shown in Figure 2. The first way,
shown in Figure 2a, is the normal way of writing GCD using
atomic actions while Figure 2b expresses GCD using a typical
imperative loop expression. The actual code also contains type
and size declarations but these are not important to understand
our compilation scheme for BMC.

The rule in Figure 2a operates on two registers x and y and
has the guard when y ≠ 0. In each step the GCD rule either
swaps the value of x and y or subtracts x from y. “|” represents
parallel composition of two actions and is a commutative
operator. Thus, if we write (x := y | y := x) the values of x
and y would be swapped at the end of the action. Also parallel
assignments to a state variable, i.e., architectural register, in
an action are illegal. The source language allows conditional
assignments to registers (as in this case) but provides no ability
to sequence actions in a rule. The guard when y ≠ 0 which
must be true for the rule to execute. Thus the GCD rule will
fire repeatedly until y becomes 0 in which case the register x
would have the GCD result. This rule is a single-cycle rule and
will need to be executed multiple times to compute the result.
The user of this GCD module has to first start it by calling
gcd.start with two arguments. This call will succeed only
when the method is ready, i.e., y=0. Then from a different
to write but immediately introduces multi-cycle rules

The imperative version shown in Figure 2b is more natural
a function and not its most efficient pipelined implementation.
overkill when one is simply interested in the value produced by
produces results. However, such interfaces appear to be an
between the initiation of function call and the time when it
a module. It also allows other computations to take place
example, it may be possible to pipeline the computation inside
our implementation (presented later) will make it clear, each
iteration of this loop will take one cycle and the total number
of the cycles needed to complete the loop will depend upon
the initial values of x and y. A rule that uses this expression
will look like a multi-cycle atomic rule, though the internal
state of the GCD function will not be visible to the caller of
the function.

Figure 3 shows the circuits that would be generated for
each of these versions. There are two main reasons for the
difference between the two circuits. The first point is that the
interfaces of the two circuits are different. Figure 2a works as
described earlier—a new pair of x and y is given whenever the
method to start GCD is ready; the GCD result can be obtained
when the result method is ready. In Figure 2b, the GCD is
recomputed whenever the values of x or y changes. And the
result can be obtained whenever the value of the while loop
expression is valid. We will describe more about these signals
later in the paper. The second point of difference is the fact
that in Figure 2b, one iteration of GCD is completed before
putting any values in the registers, while in Figure 2a, the
inputs are first registered. The rest of the circuit in Figure 2b
deal with book-keeping, and will be described in detail later.

It should be noted that if the language does not have while-
loop expressions then the only way to write GCD is as shown
in Figure 2a. For many pure functions, an implementer may
want to start with the simpler loop version and defer the
decision of writing it some other way until later.

III. KA SYNTHESIS PROCEDURE REVISITED

We briefly describe the KA procedure to compile multi-
cycle rules [3]. The procedure assumes that there are no loop-
expressions and the maximum number of cycles to compute
all the expressions in a rule is known at compile time. This
is represented by \( L_A \) for rule A. The basic schema for generating
circuits for a set of guarded atomic actions is shown in
Figure 4. This scheduler is based on the idea that all the rules
compute all the time but only a subset of rules that are ready
to commit are actually allowed to commit. Furthermore, the
architectural state (registers) that a rule reads must not change
when the rule is computing and the rule does not affect its
output registers until the cycle when it actually commits.

The schema associates a Cycles Left register with each rule,
which is set to the maximum latency of the rule \( (L_A) \) whenever
the rule starts computing. It is decremented at each clock
cycle and generates a ready-to-commit (rc\(_A\)) signal whenever
the Cycles Left counter reaches zero. The counter is reset
to \( L_A \) whenever the inputs of the rule are affected by the
commitment of another rule. The compiler does analysis to
determine if the write-set of a rule (i.e., the registers a rule
writes) and read-set of another rule (i.e., the registers a rule
reads) intersect. The compiler can allow multiple rules among
the rules that are ready-to-commit to commit provided these
rules are not in “conflict” [1], [3]. As discussed by Kaczmarek
and Arvind, such greedy scheduling can be very unfair; a fast
rule whose output is read by many other rules can essentially
cause repeated abortion of slower affected rules.

This problem is solved by the KA scheduler by allowing
rules to hold reservations. A new reservation is granted to
a rule only if its read-set does not intersect with the write-
sets of rules already holding reservations. Once a rule holds
a reservation, the reservation cannot be revoked until the rule
commits. This is achieved by associating another bit called
Still Active (sA) with each rule. This bit is set to 1 when a
rule is given a reservation and reset to 0 whenever the rule
commits.

We mention in passing that there is a subtle bug in the KA
scheduler. Suppose a rule is ready-to-commit but its write-
set intersects with the read-sets of rules that concurrently hold
reservations with this rule. Committing this rule can reset some
rules holding reservations. This will violate the invariant we
IV. A SYNTACTIC DIRECTED TRANSLATION OF BMC

In this section we describe a syntax-directed translation scheme to convert a specification in BMC into a hardware circuit.

A. Preliminaries

We first present some concepts on which the whole compounding scheme is built.

1) Architectural vs delay registers: There are two types of registers in a BMC design: architectural registers and delay registers. Architectural registers are present in the original design and introduced by the designer. Delay registers are used only to hold the temporary results of a combinational function and unlike an architectural register, are never assigned by multiple actions. Whenever the designer writes the expression delay(e), a delay register is inserted by the compiler at the output of expression e. Thus, r := f1(delay(f2(e))) will take two cycles to update architectural register r whereas r := f1(f2(e)) will take only one cycle to update r. The delay registers are also inserted by the compiler to implement loops. Delay registers are updated every clock-cycle and not directly controlled by the rule scheduler. Architectural registers on the other hand are updated only when a rule completes and commits. All the architectural registers in the write-set of a rule are updated atomically, i.e., in the same clock cycle. For an architectural register to be updated, some rule that writes into it must be ready to commit, and must hold a reservation. The scheduler ensures that two rules that update the same register cannot hold reservations simultaneously.

2) The (Valid, Change, Data) triples: We associate a 3-tuple (x_v, x_{ch}, x_d) with every variable x that represents an expression. x_d represents the actual value computed by the expression. The Valid signal x_v indicates that the expression has finished computing, and that the results are consistent with the current architectural state of the system. The Changed signal x_{ch} indicates that the state has changed and the expression has to be recomputed. Every time an architectural register in the system changes, all the expressions whose values are affected by this register have to be recomputed, and hence the corresponding Changed signals of the affected expressions have to be set to true.

We also provide an extra bit, called the valid bit with every delay register. The Changed signal invalidates the valid bit of each delay register of the expressions that are affected by the change of a particular architectural state. This forces the recomputation of all the affected expressions. We later show how to systematically set the Changed signal for each expression. Once an expression becomes valid, it remains valid until the architectural state which affects this expression changes.

3) The ready-to-commit signals for actions and rules: Architectural state registers are updated using the action r := e. Whenever the expression e becomes valid, the action r := e becomes ready to commit, indicated by the signal rc. We will later describe the procedure to compose the rc signals for all the actions in a rule to produce the ready-to-commit signal for the entire rule. The rule’s rc signal goes back to the scheduler, which decides whether to commit the rule (signal cnt) as described in Section III and [3].

4) Predicated register-assignments: If a register update is written as a conditional action if p then a (Figure 1), where a, for example, is r := e, then the update for the architectural register r is predicated based on the expression p being true. Therefore, we introduce a 2-tuple (rp, rd) to carry all the information needed to update an architectural register. Signal rp represents the predicate value which determines if the register has to be updated and signal rd represents the 3-tuple expression which is used to update the register. In every rule, the (rp, rd) signals are composed for all the actions in the rule to produce a single (rp, rd) pair for each register. We will describe this composition later.

Since many rules can update an architectural register, the (rp, rd) pairs for a register from all the rules have to be combined using muxes controlled by the commit signals from the scheduler to determine the enable signal for each register. Figure 5 shows the circuit for combining (rp, rd) pairs for each of the n registers from all the m rules. It produces a 2-tuple (wen, rd) for each n architectural registers, where wen is the Register Enable signal, and rd, as before, is the 3-tuple expression that supplies the value to the register. Because of the way the scheduler generates the commit signal for the rules, no two rules can commit in the same cycle if they are updating the same register.

5) Circuit to update an architectural register: Figure 6 shows the circuit associated with each architectural register. Each architectural register has a one bit register associated with it to indicate whether it has been updated in the last cycle. Initially this bit is set to one and after that whenever the architectural register is updated (wen is true), this bit remains
one for one cycle. As the figure shows, the Changed signal for the expression reading the architectural state becomes true for the next cycle, forcing it to be recomputed. By definition an architectural register always holds a valid value.

B. Syntax directed translation of expressions

We now describe the hardware circuit that each expression in our language generates. Every expression has the interface as shown in Figure 8a. The inputs and outputs are all 3-tuples \((x_d, x_v, x_{ch})\). In the following figures, \(T[e]\) represents the hardware circuit generated by expression \(e\).

1) Register, constant and variable \((r | c | t)\)

Figure 7b shows the hardware circuit generated by these reads of registers, constants or variables. The circuit simply returns the 3-tuple represented by the architectural register \(r\), the constant \(c\) or the variable \(t\). Note that for a constant \(c\), the Valid signal always remains true and the Changed signal always remains false.

2) Let expression \((t = e_1 \ in \ e_2)\)

Figure 7c shows the hardware circuit generated by a let expression. The expression \(e_1\) is bound to variable \(t\) and is sent as input to expression \(e_2\).

3) Primitive Operand \((op(e_1, e_2))\)

Figure 7d shows the hardware circuit generated by a primitive operator. The output can be computed only when both the expressions have been computed (conjunction of the Valid signals in Figure 7d). Any expression depending on this primitive operator must be recomputed whenever either of the expressions in the primitive operator are recomputed (disjunction of the Changed signals in Figure 7d).

4) Conditional expression \((if \ p \ then \ e_1 \ else \ e_2)\)

Figure 7e shows the hardware circuit generated by a conditional expression. The circuit waits for the predicate \(p\) to become valid. Then, depending on the value of the predicate, it waits for the either the true or the false expression to become valid. The circuit thus waits only for one of the two branches of the conditional to become valid, depending on the predicate. Any expression that depends on this conditional expression must be recomputed whenever \(p\) is recomputed or, depending on the value of the predicate, either the true or the false expression is recomputed.

5) Delay expression \((delay(e))\)

Figure 7f shows the hardware circuit generated by a delay expression. The value of expression \(e\) along with its Valid signal gets registered. Whenever \(e\) is recomputed, the Changed signal for \(e\) is set and hence the value produced by \(delay(e)\) becomes invalid. The Changed signal is passed on without being registered because it has to invalidate the values produced by all the expressions that use the value produced by \(delay(e)\). Once \(e\) is recomputed, \(delay(e)\) produces a valid value, which gets passed on to further expressions.

6) While expression \((while \ p \ do \ x=f; \ return \ x)\)

Figure 7g shows the hardware circuit generated by a while expression. The while expression introduces a register to store the values produced in each iteration of the loop as shown in the figure. In the first iteration of the loop, the value of \(x\) is taken from the input to the while expression; during the subsequent iterations, the value is taken from the loop iteration register (signal “nextData”). The signal “firstIter” denotes if the loop is executing the first iteration or not. The loop is repeated as long as the predicate \(p\) for the loop evaluates to false. Since the predicate itself can take \(x\) as input, the number of iterations of the loop will not be known at compile time. The Changed signal is used to repeat the computations in the predicate \(p\) and function \(f\) for every iteration of the loop. The signal “nextIter” behaves like a pulse in the sense that it remains true for one cycle only after every iteration, thus denoting whether the loop is starting the next iteration. This is used to trigger the Changed signal “inputChanged”. The loop iterations continue till the predicate becomes false, after which the loop produces the valid result.

C. Syntax directed translation of actions

We now describe the hardware circuit that each action in our language generates. Every action has the interface as shown in Figure 8a. The inputs are all 3-tuples \((x_d, x_v, x_{ch})\). Each action produces a ready-to-commit signal \(rc\) and a list of predicated register updates \((rp, rd)\), one for each register. In the following figures, \(T[e]\) represents the hardware circuit generated by expression \(e\) and \(T[a]\) represents the circuit generated by action \(a\).

1) Primitive action \((r := e)\)

Figure 8b shows the hardware circuit generated by a primitive register write action. The ready-to-commit signal \(rc\) is generated whenever the expression which produces the value for the register becomes valid. Since this action is not predicated, \(rp\) is set to true.
2) Conditional action (if \( p \) then \( a_1 \))

Figure 8c shows a conditional action. This action must wait for the predicate \( p \) to become valid. Then, if the predicate is false, the action is ready to commit irrespective of whether the action \( a_1 \) is ready to commit. But if the predicate is true, then this action must wait for \( a_1 \) to become ready to commit. The ready-to-commit (rc) signal shown in the figure reflects this. All the register updates of action \( a_1 \) must happen only if the predicate \( p \) of this action has been computed and it is true.

3) Parallel action (\( a_1 \mid a_2 \))

Figure 8d shows a parallel action which is the parallel composition of two actions. The whole action is ready to commit only when both the actions are ready to commit. The mux shown in the figure composes the register updates of the two actions. If both the actions are writing to the same register, then it is an error if the predicates for that register in both the actions are true. For each register, the mux passes on the predicate and the update-expression from the action where the predicate to update that register is true.

4) Let action (\( t = e \) in \( a \))

Figure 8e shows the hardware circuit generated by a let action. The expression \( e \) is bound to variable \( t \) and is sent as input to action \( a \).

D. Two phase scheduler

Figure 9 puts together all the pieces we have discussed so far. We have not discussed the compilation of guards but guards are like ordinary expressions and \( rdy \) lines simply correspond to the conjunction of valid and the data parts of a guard expression.

We have split the KA scheduler shown in Figure 4 into two schedulers in Figure 9, the rule reservation scheduler and the rule commit scheduler. Both the schedulers in Figure 9 are pure combinational circuits and if a box is drawn around them then the inputs and outputs of these combined schedulers is the same as the scheduler in Figure 4. The reservation scheduler looks at the guard values of rules and provides reservations for those rules whose guards are true (\( rdy \)) and which do not conflict with any of the rules already holding reservations (i.e., \( sa \) for still active). It never takes away reservation from a rule. The commit scheduler looks at the rules which are holding a reservation and which are ready to commit and generates the
commit signals for those rules. It also delays the commitment of a rule if there is some other sequentially composable rule holding reservation but has not committed yet (please see [3] for further details.)

E. Opportunities for Optimizations

In the case where the designer inserts no delay registers into the design, our circuits can be optimized easily to generate the same hardware as the single-cycle BSV designs. In the absence of delays, all values communicated through our 3-tuples are always valid. This is because even when an architectural register is updated, the updated value is visible throughout the circuit the very next cycle. That means that all the logic for computing valid can be optimized out. Similarly, because the only use of Changed signals is to compute multi-cycle Valid signal, those can also be optimized out, together with the one-bit registers that accompany architectural registers. Finally, actions are always ready to commit, because all expressions are always providing valid outputs. Thus all the $sa$ bits from KA will always hold 0, and are also subject to elimination. Without $sa$ bits, the KA scheduler becomes the standard Bluespec scheduler [1].

V. MULTI-CYCLE SYNTHESIS IN THE PRESENCE OF EPHEMERAL HISTORY REGISTERS

Ephemeral History Registers (EHRs) have been introduced as a way to schedule multiple rules to take place in the same clock cycle, one after the other [18], [19]. At some level, multi-cycle synthesis and EHR-based synthesis are duals of each other: one tries to use multiple clock cycles to execute one rule, while the other tries to execute multiple rules in one clock cycle. In this section, we propose a mechanism to combine these two ideas.

A basic 2-ported EHR register, as introduced for a single cycle synthesis, is shown in Figure 10a. It has four ports: two read ports $r0$ and $r1$, and two write ports $w0$ and $w1$ with their respective enables $w0en$ and $w1en$. There is a register $X$ which is the storage element backing the EHR. If write port $w0$ is enabled i.e., $w0en$ is True, then read port $r1$ reads the value written in write port $w0$, otherwise it reads the old value from $d$. $d$ is updated either by $w1$ or by $w0$ in that order, depending on which write ports are enabled. If neither are enabled, then the old value of $d$ is retained.

This scheme can be extended to multi-ported EHR registers, where each read port reads the last enabled write port whose index is smaller than that of the read port, and the final update
value for the register is given by the overall last enabled write port (or it retains the old value if none of the write ports are enabled).

Originally, the conflict analysis between rules was based solely on read and write sets of the rules. In the presence of EHRs, the conflict analysis should take into account the EHR ports which each rule uses for reading and writing a register. A higher indexed port can only be scheduled after a lower indexed port, and for read and write ports with the same index, the read should happen before the write. This actually does not change the conflict analysis for concurrent scheduling of rules; it just requires that in the implementation the scheduler must ensure that the order in which the commit signals are generated for the rules is consistent with the order imposed by the ports of the EHRs that the rules use.

One naive way of extending the multi-cycle synthesis in the presence of EHRs is to make the rule reading from r1 port of the EHR wait till another rule writing into the w0 port of the EHR is committed. This is inefficient, which can be illustrated with the following example. Consider two rules A, and B each of which in isolation takes 10 cycles to finish its computations to commit the results. Rule A writes into w0 port of an EHR X, and rule B reads from r1 port of EHR X. B does not read any other register or EHR, and finally writes into register Y. Suppose A computes the value for the w0 port of X in just 1 cycle. If B can start computing immediately after A writes into port w0, i.e., after cycle 1, then both A and B would be able to commit correctly after cycle 11. However, if the mux controlling the input for the r1 port of X depended upon the commit signal of rule A (by using only the w0en signal) then rule B won’t be able to read the value of X until 10 cycles later and the two rules would take 20 cycles to commit! Notice that in the EHR shown in Figure 10a, the muxes are controlled by register enable signals (i.e., w0en and w1en) and these signals are dependent on the rule commit signals generated by the commit scheduler in Figure 9.

In order to avoid this inefficiency, we should not (re)start the computation for rule B after w0en signal arrives. Instead, if it is known that some rule is going to write into the w0 port of X, then the new value written into the w0 port can be used as soon as the expression writing into the w0 port becomes valid, without having to wait for rule writing into w0 port to commit. The reservation signal of A, along with the predicate for writing into the w0 port gives exactly the signal we need, which we call as w0rsv, i.e., w0rsv = rsvA ∧ rpxw0. The complete circuit is given in Figure 10. Note that in a system where all rules execute in one cycle, the reservation and the commit signals for a rule are always generated simultaneously i.e., signals w0rsv and w0en are the same in Figure 10b; the distinction arises only in the multi-cycle system because the reservation signal arrives early.

Figure 11 gives the details of the multicycle EHR, which in turn uses a special multicycle mux shown in Figure 12. The generation of data and valid signals are straightforward. The change signal for the r0 port is True if either w0en or w1en is true but is delayed by one cycle using a one bit register.

Fig. 11: An EHR primitive for multi-cycle synthesis

Fig. 12: Multi-cycle MUX for Register Enable

The change signal for the r0 port is more subtle. If port w0 is being written (w0rsv is asserted), we must use the 3-tuple values from w0. Otherwise we can use the 3-tuple values from r0. However, like a conditional expression (see Figure 7e), we must also correct for the change in w0rsv: if its value changes between cycles t and t+1, we must assert the change signal in cycle t+1 because the value is being selected from a different path of the mux.

VI. ONE-RULE-AT-A-TIME SEMANTICS

We now proceed to state an important theorem about our multi-cycle synthesis procedure.

**Theorem 1.** The multi-cycle synthesis procedure described in the paper preserves the one-rule-at-a-time semantics, i.e., any state reached by a system synthesized using the multi-cycle procedure can also be reached by a system which executes at most one rule every clock cycle.

The formal proof for the above theorem is beyond the scope of this paper. Informally, the proof goes as follows. The scheduler ensures that when multiple rules are reserved, then committing those rules do not create a conflict. In other words, two rules which do not lead to the same final state as executing one of the rules followed by the other is never reserved together. Similarly, the scheduler ensures the same criterion while committing multiple rules together. Moreover, the scheduler also ensures that, when a rule is ready to commit, all other reserved rules that are yet to commit will appear to have taken place after this rule is committed (otherwise the rule which is ready to commit will not be committed).
The circuits generated using our technique have some similarity with asynchronous circuits [23]. Both methods convey similarities with the termination signal propagation in dataflow compilers [21], [22].

The synthesis problem discussed in this paper has a different goal than the usual goal of high-level synthesis. The use of delays and data-dependent loop expressions make atomic rules multi-cycle. For correctness the rules must appear to execute in some sequential order but for performance, as many rules as possible should execute concurrently. Concurrent execution of multi-cycle rules is tricky because improper scheduling can easily destroy the atomicity of rules. Both the KA scheme [3] and the scheme presented here speculatively execute all the rules but only let those rules commit that do not destroy atomicity. The problem addressed in this paper is harder than the one addressed in [3] because it is not known in advance how long a rule will take to complete. We think our valid and changed signal propagation scheme is a new way of approaching the synthesis problems. It has some similarities with the termination signal propagation in dataflow compilers [21], [22].

As can be expected, a single cycle division circuit created a very long combinational path which caused more than a six fold slowdown of the clock. The area of the circuit doubled, as the added unrolled logic now constitutes a large portion of our very simple design. Our unrolled multi-cycle divider matches the results presented in [3]. By inserting 15 delay registers, one between each step of division, the clock cycle recovers to close the speed of the original design with no divider. But the area is now more than double of the original. The divider implemented using a while loop fixes that flaw as well. The area is now only 25% larger than the original design while the clock remains almost as fast. We have succeeded in adding a new multi-cycle instruction without changing any other part of the design, without increasing the area significantly and without slowing down our clock.
the concept of completion of computation. The difference between our approach and some asynchronous design methodologies (such as dual-rail logic) is that in asynchronous circuits data is often treated as a token that needs to be consumed before more data can be processed. Our circuits are aggressive in that they always try to compute the results with newest available data, throw away partial or complete results as soon as new data is available.

Carloni et al. [24] has worked on synthesizing latency insensitive modules from synchronous module specifications with no combinational paths between inputs and outputs of each module. Vijayaraghavan et al. [25] expanded this work to remove the restriction on combinational paths, and also to enable further manual refinement of the generated latency insensitive module, calling the generated circuits as Latency-Insensitive Bounded Dataflow Networks (LI-BDNs). LI-BDNs enable multicycle refinement of a previously single-cycle design. Our work is different from LI-BDNs because our correctness criteria is based on atomic actions whereas LI-BDNs’ correctness criteria is based on the ability to simulate a synchronous specification (albeit in an asynchronous manner).

IX. SUMMARY AND FUTURE WORK

Multi-cycle synthesis of rules has been shown as an effective tool for solving timing closure for infrequently used paths. We have presented a new technique which gives total control to the designer (or another tool designed for this purpose) to partition the time over which an expression evaluates, thereby eliminating long combinational paths. The resulting circuits can include paths with data-dependent computation patterns, where different branches of computation require different number of cycles to complete. We have also introduced a looping construct which allows us to build actions with no pre-determined cycle latency. Our while-construct does not duplicate the guard and body of the loop, thus providing significant area savings over an unrolled version of the circuit. We have designed our circuits to fit into the KA scheduler without modifications, thus leveraging a proven, practical scheduling technique. We have also extended the KA multi-cycle technique to allow for multi-cycle guards.

The logical extension of this work is to enable our compiler to permit separate compilation of modules. Modular compilation of single-cycle rules has been shown to be practical [26], [27]: we intend to leverage that framework, but need to extend the method call protocol. The technique presented here has laid important ground work for that extension. Another interesting extension of this work would be to allow for inclusion of an action outside of our while loop. This would allow an easy expression of a complex action, such as writing several registers of a processor with a single instruction with only one register write port available.

REFERENCES