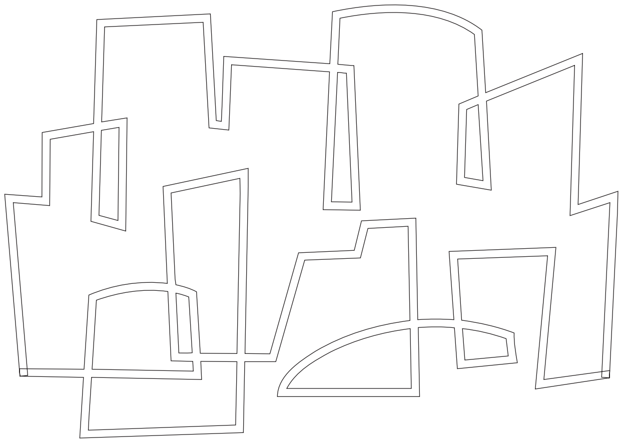
The Ephemeral History Register: Flexible Scheduling for Rule-Based Designs

In the proceedings of Formal Methods and Models
for Codesign (MEMOCODE'2004) ,
San Diego, California, June 22-25, 2004

Computation Structures Group
Memo 479

June, 2004

Daniel Rosenband



The Ephemeral History Register: Flexible Scheduling for Rule-Based Designs

Daniel L. Rosenband
Computer Science and Artificial Intelligence Lab
Massachusetts Institute of Technology
Cambridge, MA 02139
danlief@csail.mit.edu

Abstract

The quality of high-level synthesis results is strongly dependant on the concurrency that can be found in designs. In this paper we introduce the Ephemeral History Register (EHR), a new primitive state element that enables concurrent scheduling of arbitrary rules in a rule-based design framework. The key properties of the EHR are that it allows multiple operations to write to the same state simultaneously, and that the EHR maintains a history of all writes that occur within a clock-cycle. Using the EHR, we present an algorithm that takes as input a design and a desired schedule, and produces a functionally equivalent design that satisfies the desired concurrency and ordering of operations. A processor pipeline is used to illustrate the effectiveness of the EHR and scheduling algorithm, and shows how this approach significantly improves on previous synthesis algorithms for rule-based designs.

1. Introduction

There is a need for a new hardware design and synthesis approach to address the growing complexity of hardware designs. Desired properties of such an approach are (i) the input language must have well-defined execution semantics that bridge the gap between specification, design and formal verification (ii) the methodology should encourage correct-by-construction designs and (iii) performance must match the designer's expectations. We use guarded atomic actions, which we also refer to as rules, as a basis for our synthesis framework because they address all three of these properties. Guarded atomic actions have a strong semantic foundation[1-4] and previous work has shown how they can be applied to hardware specification[5-7], synthesis[8-10] and verification[11, 12].

Much of the research related to hardware synthesis from rule-based descriptions has focused on achieving maximal concurrent scheduling of rules within each clock cycle. Although many of the synthesis results have produced hardware that is comparable to handcoded RTL Verilog, large designs sometimes exhibit unpredictable and at times poor performance due to scheduling inefficiencies. Additionally, oftentimes the only way a designer can achieve a desired schedule is to assert scheduling properties that carry proof obligations. If the designer makes an error in this process, then the design will not only exhibit poor performance, but it might also become functionally incorrect. This paper presents scheduling for predictable performance by introducing new scheduling algorithms that are based on a new primitive state element, the Ephemeral History Register (EHR). Besides improving performance, the scheduling algorithms also remove the possibility of introducing functional errors when guiding the rule scheduling process. If the designer does not request the correct scheduling properties, then the design might not achieve the desired performance, but it will still produce a behavior that can be explained as some sequential firing of rules. This is important towards achieving correct-by-construction design and to remain within the formal framework of guarded atomic actions.

The basic idea behind the new algorithms and the EHR came from the realization that none of the current rule-based synthesis algorithms allow values that are written by one rule to be forwarded to another rule that executes within the same cycle. For values that are computed in the datapath, this is often undesirable as it can increase the critical path of the design. But, as we show in later sections, there are many cases where forwarding of control values from one rule to another can significantly improve the performance of the design. The EHR is a new primitive state-element that allows values to be forwarded from one rule to another while remaining in the semantic framework of guarded atomic actions. What distinguishes the EHR from a standard register is that it maintains a history of all values that are written to it within a clock cycle. Rather than return the current state as a conventional register would return, reads to the EHR can access the history and return (forward) one of the values being written to it. Designers can directly instantiate the EHR in their designs to improve the scheduling efficiency. But more importantly, the EHR allows more efficient scheduling algorithms to be developed. These algorithms accept as input a design and associated scheduling constraints. The design is not required to already use the EHR, but can be constructed from the standard register primitive. The scheduling constraints specify the desired concurrency and ordering of rules and interface methods. As output, the algorithms produce a transformed design which uses the EHR in place of the standard register primitive. The transformed design is guaranteed to be functionally equivalent to the original design, and the interface methods and rules are guaranteed to satisfy the scheduling constraints. In general, given sufficient resources, any scheduling constraints can be satisfied. This is a powerful mechanism that produces high-quality hardware for large designs and allows the designer to easily manipulate schedules without breaking the semantics of guarded atomic actions or risking incorrect functional behavior.

Related work appears in the context of high-level hardware synthesis that is based on control-data flow graphs (CDFG's). For example, chaining is presented in[13] as a mechanism to improve performance by forwarding the value from one operation to another without storing an intermediate result. Dynamic renaming is used in[14] to eliminate data dependencies that limit code motion, and hence allows more aggressive compiler optimizations to be implemented. The major difference between the CDFG synthesis flows and synthesis from atomic actions is that CDFG's focus on generating an efficient *static* schedule of operations over a sequence of control steps. In contrast, rule-based synthesis generates a scheduler that *dynamically* determines which rules fire in every cycle. We believe dynamic scheduling is important in hardware systems because many designs have 1. a large number of data dependent conditional paths, each with its own timing and resource requirements, 2. have subsystems with variable and unpredictable latencies (due to caching and interference from other processes, etc.), and 3. have input events whose timing is often unpredictable. Because schedules are dynamically generated in our synthesis framework, it cannot always be determined in advance what the source of a value is. The value might be the stored register value, or it could be one of several forwarded values depending on what other operations execute. Although in some respects similar to the chaining and renaming ideas in the CDFG synthesis context, dynamic scheduling requires a different approach to forwarding of values from one operation to another.

Paper organization: The next section reviews previous synthesis methodologies for guarded atomic actions and illustrates an example where this approach is not sufficient to produce the desired performance. Section 3 introduces the Ephemeral History Register and shows how it can be used to achieve the desired scheduling performance. Section 4 presents a new scheduling algorithm that accepts scheduling constraints as input, and using the EHR, transforms the design to meet the scheduling requirements. We conclude in Section 5.

2. Rule-based hardware synthesis

This section reviews the execution model of atomic actions and outlines the synthesis approach of Hoe and Arvind[8, 9]. We then present a processor pipeline that uses a FIFO built from primitive registers for its pipeline stages. This example demonstrates the need for a new scheduling approach because the previous algorithms are unable to derive sufficient concurrency in rule firings.

2.1. Atomic Action Execution Model

Each 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:

$$\text{rule } R_i\text{:} \quad \text{when } \pi_i(s) \text{ ==> } s := \delta_i(s)$$

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 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* $R_i$ , s.t. $\pi_i(s)$ is true
  2) s := $\delta_i(s)$
```

2.2. Synthesizing Rules into RTL Hardware

There is a straightforward translation from rules into hardware. Assuming all state is accessible (no port contention), each π and δ can be easily implemented as combinational logic. A hardware scheduler and control circuit then needs to be added so that in every cycle the scheduler dynamically picks one δ function whose corresponding π condition is satisfied. At the end of the cycle the control circuit updates the state of the system with the result of the selected δ function. The cycle time in such a synthesis is determined by the slowest π and the slowest δ functions.

Although correct, such an implementation has unsatisfactory performance because it is often possible to execute several rules simultaneously such that the result of the execution continues to match an execution in which the selected rules are applied in some sequential order. Thus, the challenge in generating efficient hardware from sets of atomic actions is to generate a scheduler which in every cycle picks a maximal set of rules that can be executed simultaneously. In this paper we assume that each rule executes within a single cycle but we are also investigating implementations where the execution of a rule may stretch over multiple cycles.

Figure 1 shows the circuit that is generated in Hoe's synthesis flow. The predicates (π_i's) are computed for each rule using a combinational circuit. The scheduler is designed to select a maximal subset of applicable rules with the constraint that the outcome of a scheduling step can be explained as atomic firing of rules in some sequence. Based on which rules the scheduler chooses to enable (φ_i's), the selector block then combines the update functions (δ_i's) from the chosen rules and updates the current state with the resulting values.

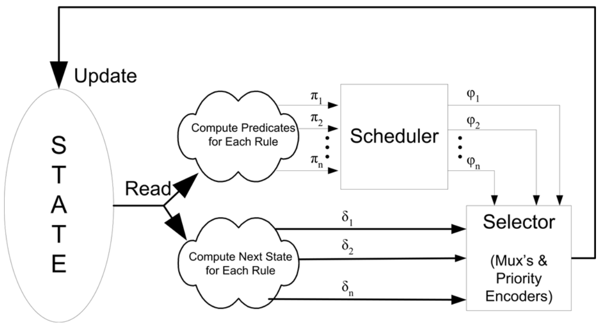

Update
S T A T E
Read
Compute Predicates for Each Rule
π1
π2
πn
Scheduler
φ1
φ2
φn
Compute Next State for Each Rule
δ1
δ2
δn
Selector
(Mux's & Priority Encoders)

Figure 1: Synthesized Atomic Actions

With some small differences that are pointed out in Section 4, we use the same model for circuit generation in our new synthesis flow. The main difference in our flow lies in the fact that we introduce the EHR state element and that we use a new scheduling algorithm, not in the circuit generation.

2.3. Rule-based processor pipeline

The example we use throughout this paper is derived from a standard 5-stage processor pipeline[15]. In a rule-based design environment such a processor is typically expressed using one rule to specify the behavior of each of the five pipeline stages. FIFO's are used as pipeline stages so that the stages can be described and scheduled independently of one another. In contrast to traditional hardware descriptions, this description-style lends itself to proving the correctness of the processor implementation[5].

In this paper we focus on the interaction of rules that access the write-back (wb) FIFO – the FIFO between the memory and the write-back stage. The rules that interact with this FIFO are R_{decode} (accepts a bypassed value), R_{mem} (enqueues the next value into the FIFO), and the write-back rule itself, R_{wb} (reads an element from the FIFO and dequeues it). These interactions are illustrated in Figure 2 and the following code snippet. We note that *wb.first* and *wb.bypass* are purely combinational circuits that return information about the state of the FIFO, whereas *wb.enqueue* and *wb.dequeue* both modify the state of the FIFO.

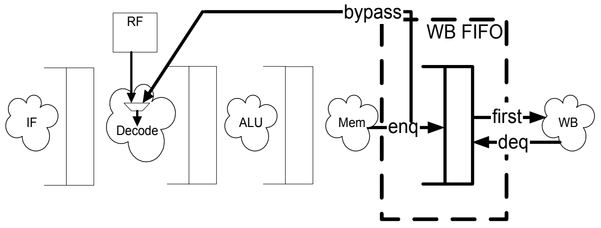


Figure 2: Processor Pipeline

```
Rdecode: … wb.bypass;
Rmem:    wb.enqueue …;
Rwb:     … wb.first; wb.dequeue;
```

In general, a designer would expect all three of these rules to execute concurrently, with the implied ordering:

$$R_{wb} < R_{mem} < R_{decode}$$

Here, $R_a < R_b$ means that R_a and R_b can execute concurrently and simultaneous execution will behave the same as sequentially executing R_a followed by R_b. This ordering would be required because we are using a single element FIFO and hence would have to read and dequeue the value (R_{wb}) from the FIFO before we can insert a new value (R_{mem}). After the new value has been inserted (R_{mem}), it can be bypassed to the decode rule (R_{decode}).

The question that needs to be examined is: do current scheduling algorithms allow these three rules to execute concurrently? Hoe and Arvind[9] showed that efficient hardware can be generated from such a description, but required that a special FIFO primitive be used. However, if the FIFO is constructed from primitive registers, then as we show in the next subsection, the compiler derives an unsatisfactory schedule. Bluespec[16] and [17] improved on Hoe's scheduling, but only by supporting scheduling annotations that carried proof obligations. An error in these annotations could easily lead to a functionally incorrect design. Thus, there clearly exists a need for improved scheduling algorithms. The new algorithms that we later present solve this scheduling problem while fitting into the semantic and language framework of [16] and [17]. We should note that we use the FIFO because it is a simple example that exhibits many of the problems with the current scheduling algorithms. As designs get larger, many similar problems have arisen. Introducing new primitives, as would be required in Hoe's flow, or requiring annotations with proof obligations as would be required in [17] is generally not a satisfactory solution.

2.4. Pipeline Analysis using CF and SC

Below we provide the code for a possible implementation of the processor pipeline FIFO. This FIFO contains only a single element and is constructed from two registers: *data* and *full*. The *data* register holds the contents of the FIFO element. The *full* register is true if the FIFO is full, that is, the *data* register contains valid data. The FIFO contains the standard *enqueue*, *dequeue*, and *first* methods, along with a *bypass* method which is intended for bypass logic. The *first* and *bypass* methods are written using exactly the same code since they both return to contents of the FIFO. We later show how we can move the *bypass* function to appear to execute later in time so that it observes values written by the *enqueue* method. Each method has a *when* condition, which must be true for a rule to be able to call the method. For example, a rule cannot call *enqueue* if the FIFO is full.

The FIFO is expressed as a set of module methods, which for the purpose of this paper can be assumed to be flattened into the rules that call them during the compilation process. This means that each method body is inlined into the body of the rule that calls it, and each method "when condition" is conjugated with the rule predicate. We use the method notation because it allows us to analyze the FIFO scheduling properties and then given the FIFO scheduling properties, show the impact on the processor rules.

```
enqueue x =
     data.write x;
     full.write 1;
when (full.read == 0)

dequeue =
     full.write 0;
when (full.read == 1)

first =
     return data.read;
when (full.read == 1)

bypass =
     return data.read;
when (full.read == 1)
```

Past rule-based synthesis approaches recognized that some rules can execute concurrently and appear to still execute sequentially and atomically. Both Staunstrup[10] and Hoe[8, 9] made the observation that two rules can execute simultaneously if they are "Conflict Free" (CF), that is, they do not update the same state and neither updates the state read by the other rule. Arvind and Hoe further observed that two rules (R_1 and R_2) can execute simultaneously if one rule (R_2) does not read any of the state that the other rule (R_1) writes. In this case, simultaneous execution of R_1 and R_2 appears the same as sequential execution of R_1 followed by R_2. For this to hold, R_2 writes must take precedence over writes to the same state by R_1. Such rules are called "Sequentially Composable" (SC) in[9]. Hoe showed that from the pair-wise CF and SC relationships between rules one can deduce if a group of rules can be scheduled concurrently.

If we apply the CF and SC analysis to the FIFO we obtain the following FIFO scheduling matrix. A "*c"* in this table indicates that the two methods (m_1 and m_2) do not appear to execute in the order m_1 followed by m_2 when simultaneously enabled. A "< " in the table indicates that the two methods can be scheduled to execute simultaneously and it will appear as though m_1 executed before m_2. In general, two methods conflict if they are neither CF nor SC. Two methods have the "<" property if they are sequentially composable in that direction or if they are conflict free.

$m_1 \backslash m_2$	enqueue	dequeue	first	bypass
enqueue	c	C	c	c
dequeue	c	C	c	c
first	<	<	<	<
bypass	<	<	<	<

Given these FIFO scheduling properties we can derive the scheduling constraints that the FIFO imposes on the processor pipeline. If all the methods that two rules call are sequentially composable in the same direction, then the rules can execute concurrently and appear to execute sequentially. Thus, we write $R_1 < R_2$ if all the methods that R_1 calls are "<" all the methods that R_2 calls. If neither $R_1 < R_2$ nor $R_2 < R_1$ holds, then the rules conflict and cannot execute simultaneously (R_1 *c* R_2). Thus, given the FIFO scheduling properties we obtain the following constraints on the processor pipeline:

R_{decode}	<	R_{mem}
R_{decode}	<	R_{wb}
R_{mem}	c	R_{wb}

Clearly, SC and CF analysis are not sufficient to create a FIFO that works as a proper pipeline register. Sufficient concurrency is not found and the value being inserted into the FIFO cannot be forwarded to another rule that executes within the same cycle. We have experimented with other FIFO designs and believe that no FIFO can be constructed from existing primitives and achieve the desired properties using the past rule-based synthesis framework. Similar issues arise in more complex designs. This problem is what motivated us to find an alternate scheduling algorithm. The reader will note that neither CF nor SC permit values to be forwarded from one rule to another. Hence, using the SC and CF scheduling methods it is impossible to correctly schedule or create a bypass function that returns the value being enqueued into the FIFO.

3. The Ephemeral History Register (EHR)

The basic idea behind the new scheduling approach is to permit the forwarding of values between rules that execute within the same cycle. If one rule is writing to a register, and another rule is reading the same register, then the value that is written can be forwarded to the rule that is reading the register. In this case it appears as though the rule that is reading the register executes after the rule that is writing the register, even though they are executing within the same cycle. In essence, we are dividing each clock cycle into sub-cycles and assigning rules to particular sub-cycles. The values of registers that are being written in one sub-cycle can be read by rules that read the register in a later sub-cycle. For example, in the above FIFO implementation, the *enqueue* method should observe any writes that the *dequeue* method makes to the *full* register. If the FIFO is initially full, then *dequeue* clears the *full* register, which when forwarded to the *enqueue* method would in turn allow it to execute within the same cycle. If *enqueue* does not observe the forwarded value, then as we have previously shown, it cannot execute concurrently with *dequeue*. Thus, *dequeue* should execute in an earlier sub-cycle than *enqueue.* Looking at the processor rules, we would assign R_{wb} to sub-cycle 0, R_{mem} to sub-cycle 1, and R_{decode} to sub-cycle 2, since each of these rules is forwarding a value to the rule in the next sub-cycle.

Neither Hoe and Arvind's rule-based synthesis flow nor the more advanced Bluespec language support a construct that allows a value to be forwarded from one rule to another. We introduce the Ephemeral History Register (EHR) as a new primitive state element that supports the forwarding of values. A circuit diagram of the EHR is show in Figure 3. We call it the 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 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 method *write*. The reader will note that each *write* method has two signals associated with it (*x* and *en*). The *x* input is the data input and could be a bus. The *en* input is a control input that indicates the method is being called and should execute. A value is not written unless the associated *en* signal is asserted.

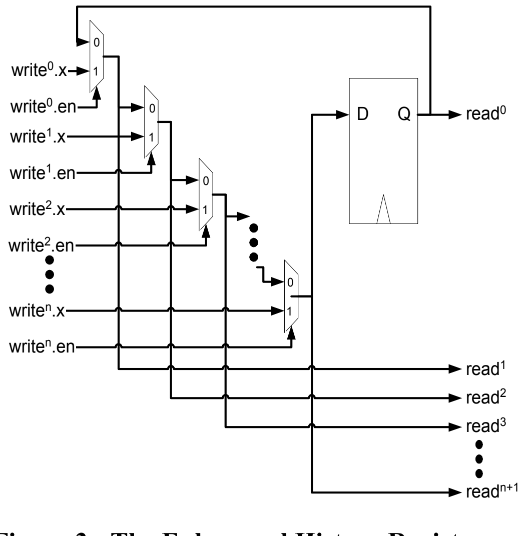


Figure 3: The Ephemeral History Register

A few key properties of the EHR are:

- *read*^{0} returns the current value of the register.
- if *write*^{i} is not enabled for any i, then the register value does not change.
- *write*^{i} takes precedence over *write*^{j} if $i > j$.
- *read*^{i} returns the value being written by *write*^{j}, where j is the largest value less than i for which a write is enabled. It returns the current value of the register if no such write is enabled.

Any number of *read* and *write* method versions can be used in each EHR instance. For example, an EHR with only a *read*^{0} and a *write*^{0} method behaves the same as the standard register primitive. An EHR with a single *read* and two *write* methods is equivalent to the circuit that is required for sequential composition of rules since it gives preference to the *write* method with the higher version. If an EHR has multiple *read* interfaces, then values are forwarded from *write* methods with a lower version to *read* methods with a higher version number. Unused method versions can clearly be optimized away during compilation.

Just as we derived a scheduling matrix for the FIFO implementation, as shown below, we can create a scheduling matrix for the EHR. The reader will recall that a "<" entry in the scheduling matrix indicates that m_1 and m_2 can execute concurrently and that such execution is equivalent to the sequential execution of m_1 followed by m_2. The "*c*" indicates that this is not possible. For example, the *write*^{0} \ *write*^{2} entry is "<" because if both methods are enabled simultaneously, then the *write*^{2} value takes precedence and is the value that is written to the register. Hence, it appears as though *write*^{2} executes after *write*^{0}. Similarly, the *read*^{1} \ *write*^{0} entry is a "*c*" because the *read*^{1} method observes the value written by *write*^{0}. Thus, it is impossible to appear as though *read*^{1} executes before *write*^{0} if both methods are called simultaneously.

m_1\\m_2	write^0	write^1	write^2	read^0	read^1	read^2
write^0	c	<	<	c	<	<
write^1	c	c	<	c	c	<
write^2	c	c	c	c	c	c
read^0	<	<	<	<	<	<
read^1	c	<	<	<	<	<
read^2	c	c	<	<	<	<

In general, the EHR has the scheduling constraints:

- *write*^{i} < *write*^{j} for all versions where $i < j$
- *write*^{i} < *read*^{j} for all versions where $i < j$
- *read*^{i} < *write*^{j} for all versions where $i \leq j$
- *read*^{i} < *read*^{j} for all versions i, j
- *c* for all other method pairs

3.1. FIFO implementation using the EHR

This section shows how the EHR can be used to implement a FIFO with the desired scheduling properties. The FIFO *bypass* method also has the desired property of returning the value being inserted into the FIFO. The implementation is based on the FIFO code from section 2. However, rather than use the standard register primitive for the *full* and *data* registers, this implementation uses the EHR for both state elements. Otherwise, the only changes that are made to the code are that version numbers of the register method calls are changed. In section 4 we show how these changes can be automatically derived through new scheduling algorithms.

In order to achieve the proper processor pipeline schedule we require the FIFO methods to satisfy the following scheduling properties:

(first < dequeue) < enqueue < bypass

These constraints are based on the processor pipeline rule ordering that we derived in section 2. An implementation that satisfies these FIFO scheduling requirements is:

```
enqueue x =
    data.write$^0$ x;
    full.write$^1$ 1;
when (full.read$^1$ == 0)

dequeue =
    full.write$^0$ 0;
when (full.read$^0$ == 1)

first =
    return data.read$^0$;
when (full.read$^0$ == 1)

bypass =
    return data.read$^1$;
when (full.read$^2$ == 1)
```

Based on pairwise analysis of the methods each of the FIFO's methods calls, we can derive the scheduling matrix for this FIFO. Entries that are unchanged from the original FIFO implementation are shaded.

m_1 \ m_2	enqueue	dequeue	first	bypass
enqueue	c	c	c	<
dequeue	<	c	c	<
first	<	<	<	<
bypass	c	c	<	<

It is important to note that each method individually behaves precisely as it did in the original implementation. This is important towards maintaining the semantic model of guarded atomic action execution. Only when methods are called together does the behavior change, and such behavior is valid in both cases only when it can be explained as a sequential execution of the two methods (or rules). For example, the bypass method returns the current state of the *data* register if the FIFO contains valid data (the *full* register is set) and no other FIFO methods are called. However, if the *enqueue* method is called simultaneously, then the *bypass* method now returns (forwards) the value being enqueued. Previously, these two methods could not be called simultaneously.

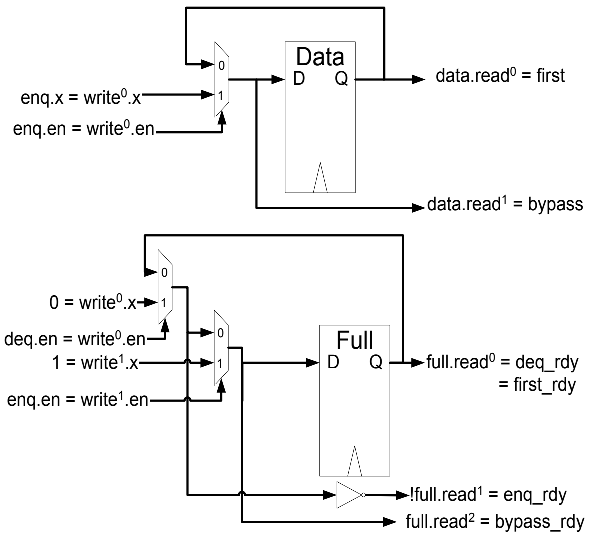


Figure 4: EHR FIFO Implementation

The circuit that is synthesized from the new FIFO description is shown in Figure 4. The *rdy* signals correspond to the values of the method *when* conditions. This circuit behaves precisely as the designer would expect and allows the processor pipeline to be scheduled with the desired performance. After optimizing the constant inputs to the mux's, the circuit is also equivalent to what a designer would have implemented as a pipeline stage in a RTL-level implementation. Thus, using the EHR as a new primitive state element, we are able to build designs that are more efficient than was previously possible in a rule-based synthesis environment. The use of the EHR is not only advantages when designing FIFO's, but improves scheduling in more complex designs as well.

4. Flexible Scheduling Algorithm

We have shown that the Ephemeral History Register is a powerful new primitive element that allows the scheduling of designs to be improved. However, as the example in section 3 showed, the designer still has to alter the design to achieve a desired schedule. In this framework the changes a designer has to make to satisfy scheduling constraints are limited to changing the version of method calls to EHR instances. However, this is a tedious process, and if not performed correctly can lead to designs with poor performance, or even functionally incorrect designs. The ability to easily experiment with different schedules is an important component of architectural and implementation exploration. This led us to develop a new scheduling algorithm that takes a set of rules (or methods) along with a set of scheduling constraints as input. The scheduling constraints specify desired concurrency and execution ordering among rules. As output the algorithm produces a transformed design that satisfies the scheduling constraints and is functionally equivalent to the original design. In general, any scheduling constraint can be satisfied, provided that sufficient resources (ports) are available.

There are two key components to our scheduling algorithm. The PROMOTE procedure takes a rule as input and transforms it into a new rule that is functionally equivalent to the original rule, but appears to execute later in time relative to other rules – it executes in a later sub-cycle. The PROMOTE procedure accomplishes this by selectively increasing the version of method calls to EHR instances. The TSCHEDULE function is the top level procedure that takes a set of rules and the scheduling constraints as input and produces a new design that meets the scheduling requirements. It achieves this by repeatedly calling PROMOTE on select rules so as to advance them in time until they meet the scheduling constraints. After a finite number of calls to PROMOTE, the design is guaranteed to satisfy the constraints.

The next two subsections explain these two procedures. We then show how the procedures can be used to automatically derive the design that satisfies the designer's expectation for the original processor pipeline.

4.1. PROMOTE

The idea behind the PROMOTE procedure is that we can increase the version of calls to EHR methods without altering the behavior of a rule or method call. By increasing the version we can reduce conflicts with other rules that are calling methods from the same

EHR. For example, if R_1 calls $write^0$ of an EHR and R_2 calls $read^0$ of the same EHR, then we can achieve $R_1 < R_2$ without altering the behavior of either rule by promoting the $read^0$ call to $read^1$. However, if R_2 also calls $write^0$, then promoting $read^0$ to $read^1$ would alter the behavior of the rule since it would now forward the value from $write^0$ to its own call to $read^1$. The way to avoid this would be to also increase the version of $write^0$ to $write^1$.

The PROMOTE procedure, as described below, accepts a rule (R) or method as input. It also accepts as input one of the method calls (x^i) that appear in the rule. The goal of the algorithm is to increase the version of x^i to x^{i+1} without altering the behavior of the rule. It returns a new rule that increases the version of x^i along with a minimal number of other method calls while maintaining the same functionality as the original rule.

One simple approach to increasing the version of an EHR method call without altering the rule behavior is to increase by the same amount all versions of calls to the same EHR within the rule. However, this is not efficient since there are cases where only a subset of the calls to an EHR need to have their version increased. Step 3 in the PROMOTE algorithm spells out the precise requirements for increasing the version numbers. The guiding principle is that the version number should only be increased if it helps scheduling or if it needs to be increased to avoid altering the behavior of the rule.

```
PROMOTE(R, $x^i$) =
  • Assume R is a rule (or method) that makes calls to the set of methods X = {$x_0$, $x_1$, $x_2$, …}, $x^i \in X$, and $x^i$ is a call to an EHR method.
  1) Let B be the minimal subset of X such that $x^i \in$ B and such that all method calls y in (R – B) satisfy one of the properties:
       • $y \text{ c } x^i$
       • $y < x^i$
       • $x^i < y$ and $x^{i+1} < y$
  2) For each method call $x^j \in$ B, replace the call to $x^j$ in R by a call to $x^{j+1}$.
  3) Return the new R.
```

4.2. TSCHEDULE

The TSCHEDULE procedure takes a sequence of rules as input and transforms them into new rules that are functionally equivalent to the original rules. The new rules have the scheduling property that they can all execute simultaneously and appear to execute in the order they were listed in the input. We first show how the TSCHEDULE algorithm can be applied to a pair of rules and then show how it can be generalized to an arbitrary number of rules.

In the two input case, the TSCHEDULE procedure accepts two rules (R_x and R_y) as input and produces a new rule R_y' that satisfies the property $R_x < R_y'$. The procedure always succeeds at achieving the scheduling requirement and guarantees that R_y' has the same behavior as R_y.

Functional correctness is guaranteed since the only transformation that is applied to R_y is PROMOTION, which we showed above does not alter the behavior of a rule. It is also clear that we eventually achieve the desired scheduling relationship between R_x and R_y. The reason for this is that we repeatedly promote elements (either calls to EHR methods, or the methods of modules that both R_x and R_y call). If sufficient promotion occurs, then everything in R_y' must appear to execute "later" than R_x, and hence $R_x < R_y$ must be true eventually.

```
TSCHEDULE($R_x$, $R_y$) =
  • Assume $R_x$ is a rule (or method) that makes calls to the set of methods X = {$x_0$, $x_1$, $x_2$, …}
  • Assume $R_y$ is a rule (or method) that makes calls to the set of methods Y = {$y_0$, $y_1$, $y_2$, …}
  1) If $R_x < R_y$ return $R_y$
  2) There exists an $x \in$ X and $y \in$ Y such that $x < y$ does **not** hold. (If no such element exists, then $R_x < R_y$ must hold and we would have exited in step 1.)
  3) If x and y are calls to EHR methods, then $R_y$ = PROMOTE($R_y$, y).
  4) If x and y are **not** calls to EHR methods, then TSCHEDULE(x, y).
  5) Go to step 1.
```

The TSCHEDULE procedure can easily be generalized for an arbitrary set of rules. By successively achieving pairwise scheduling requirements, we end up with the desired schedule for the entire set of rules.

```
TSCHEDULE($R_x$, $R_y$, $R_z$, …) =
  1) $R_X$ = TSCHEDULE($R_x$, $R_y$).   // satisfy $R_x < R_y$
  2) $R_X$ = TSCHEDULE($R_x$, $R_z$).   // satisfy $R_x < R_z$
  3) …                                 // satisfy $R_x < …$
  4) $R_Y$ = TSCHEDULE($R_y$, $R_z$).   // satisfy $R_y < R_z$
  5) …
// satisfies $R_x < R_y < R_z < …$
```

4.3. Processor pipeline scheduling

The TSCHEDULE and PROMOTE algorithms are best illustrated through an example. In Table 1 we show a simulation of the algorithm applied to the processor pipeline. As input the TSCHEDULE algorithm takes the three rules we were concerned with (R_{wb}, R_{mem}, and R_{decode}), with scheduling requirement:

$$R_{wb} < R_{mem} < R_{decode}$$

The simulation highlights all TSCHEDULE and PROMOTE procedure calls and indicates when a rule or interface method is altered. All transformations are changes to version numbers of calls to methods of the state elements inside the FIFO. The outcome is precisely the description that we manually created in section 3.

Table 1: Algorithm Simulation

Step	Algorithm
1	TSCHEDULE(R_{wb}, R_{mem}, R_{decode});
2	TSCHEDULE(R_{wb}, R_{mem});
3	TSCHEDULE(dequeue, enqueue);
4	PROMOTE(enqueue, full.read^0); **Change in enqueue:** • full.read^0 to full.read^1 • full.write^0 to full.write^1 (dequeue < enqueue now true) ($R_{wb} < R_{mem}$ now true)
5	TSCHEDULE(R_{wb}, R_{decode});
6	TSCHEDULE(dequeue, bypass);
7	PROMOTE(bypass, full.read^0); **Change in bypass:** • full.read^0 to full.read^1 (dequeue < bypass now true) ($R_{wb} < R_{decode}$ now true)
8	TSCHEDULE(R_{mem}, R_{decode});
9	TSCHEDULE(enqueue, bypass);
10	PROMOTE(bypass, full.read^1); **Change in bypass:** • full.read^1 to full.read^2
11	TSCHEDULE(bypass, data.read^0);
12	PROMOTE(bypass, data.read^0); **Change in bypass:** • data.read^0 to data.read^1 (enqueue < bypass now true) ($R_{mem} < R_{decode}$ now true)

4.4. Circuit generation

For the most part, circuit generation when using the EHR and associated scheduling algorithms is equivalent to the circuit generation that Hoe used. One exception arises because there is the possibility of generating combinational loops with the EHR. This arises when rule R_1 forwards a value to R_2's predicate. If R_2 conflicts with rule R_1, then a combinational loop arises if the scheduler gives strict preference to R_2 over R_1 since R_2 would disable R_1. R_1 in turn would no longer produce the value that enabled R_2, and so R_2 would get disabled. R_1 could then be enabled again, etc. The way to avoid such loops is to break the loop in the scheduler – either by giving preference to R_1 or detecting that R_1 is the rule that enables R_2 to execute.

5. Conclusion

In this paper we presented a new scheduling algorithm for rule-based designs that significantly improves on previous methods. These are general algorithms that can be applied to many designs. They are particularly useful for large designs that require flexibility in scheduling without risking incorrect functional behavior.

The algorithms are made possible through the use of a new state element, the Ephemeral History Register. This new primitive element makes forwarding of values from one rule to another possible, while maintaining the semantics of guarded atomic actions. As an example of the power of the EHR and scheduling algorithm, we presented a processor pipeline and showed that we were able to build the pipeline FIFO's using the EHR – something that previously could not be done using only primitive elements. This FIFO was interesting because it was implemented using only a single storage element, allowed simultaneous enqueue and dequeue, and allowed the value that was being enqueued to be bypassed to another rule.

The scheduling algorithms are useful because they allow a designer to precisely specify how rules should be scheduled. The compiler then takes these requirements and transforms the design to meet the constraints. By providing incorrect constraints, the designer might not achieve the desired performance, but will never cause the design to become functionally incorrect. This contrasts with the previous compilation flow where a designer had to compile a design and then observe the scheduling results. It was often difficult to understand what was limiting the scheduling performance and once the scheduling problem was discovered, the code had to be rewritten to achieve the desired performance. This was a time-consuming and error-prone process.

Both the EHR and the scheduling algorithms are powerful new mechanisms that we have shown to be practical through the processor pipeline example.

Acknowledgements: We would like to thank Arvind for his support and help in developing the ideas for this paper. We also thank Compaq for their financial support.

References

[1] E. W. Dijkstra, "Guarded commands, nondeterminacy and formal derivation of programs," *Commun. ACM*, vol. 18, pp. 453-457, 1975.
[2] K. M. Chandy and J. Misra, *Parallel program design : a foundation*. Reading, Mass.: Addison-Wesley Pub. Co., 1988.
[3] N. A. Lynch, *Atomic transactions*. San Mateo, Calif.: Morgan Kaufmann Publishers, 1994.
[4] R. J. R. Back and R. Kurki-Suonio, "Decentralization of process nets with centralized control," in *Proceedings of the second annual ACM symposium on Principles of distributed computing*: ACM Press, 1983, pp. 131-142.
[5] Arvind and X. Shen, "Using term rewriting systems to design and verify processors," *Micro, IEEE*, vol. 19, pp. 36-46, 1999.
[6] X. Shen, "Design and verification of adaptive cache coherence protocols," in *Dept. of Electrical Engineering and Computer Science*: Massachusetts Institute of Technology, 2000, pp. 178 p.
[7] J. Plosila and K. Sere, "Action systems in pipelined processor design," presented at Proceedings Third International Symposium on Advanced Research in Asynchronous Circuits and Systems, 1997.
[8] J. C. Hoe, "Operation-centric hardware description and synthesis," in *Dept. of Electrical Engineering and Computer Science*: Massachusetts Institute of Technology, 2000, pp. 139 p.
[9] J. C. Hoe and Arvind, "Synthesis of operation-centric hardware descriptions," presented at IEEE/ACM International Conference on Computer Aided Design (ICCAD), 2000.
[10] J. Staunstrup and M. R. Greenstreet, "From High-Level Descriptions to VLSI Circuits," *BIT*, vol. 28, pp. 620-638, 1998.
[11] D. L. Dill, "The Murphi verification system," in *Proceedings of the Eigth International Conference on Computer-Aided Verification*, vol. 1102, *Lecture Notes in Computer Science*: Springer-Verlag, 1996.
[12] J. Stoy, X. Shen, and Arvind, "Proofs of Correctness of Cache-Coherence Protocols," in *Formal Methods Europe*, vol. 2021, *Lecture Notes in Computer Science*, J. N. Oliveira and P. Zave, Eds.: Springer-Verlag, 2001, pp. 43-71.
[13] D. D. Gajski, *High-level synthesis : introduction to chip and system design*. Boston: Kluwer Academic, 1992.
[14] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau, "SPARK: a high-level synthesis framework for applying parallelizing compiler transformations," presented at VLSI Design, 2003. Proceedings. 16th International Conference on, 2003.
[15] J. L. Hennessy and D. A. Patterson, *Computer Architecture a Quantitative Approach*, Second Edition ed: Morgan Kaufman, 1996.
[16] L. Augustsson and others, "Bluespec: Language definition," Sandburst Corp., 2001.
[17] D. L. Rosenband and Arvind, "Modular Scheduling of Guarded Atomic Actions," Proceedings of the 41st Design Automation Conference (DAC), 2004.