High-level Automatic Pipelining for Sequential Circuits [*]

Maria-Cristina V. Marinescu
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139
cristina@lcs.mit.edu

Martin Rinard
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139
rinard@lcs.mit.edu

ABSTRACT

This paper presents a new approach for automatically pipelining sequential circuits. The approach repeatedly extracts a computation from the critical path, moves it into a new stage, then uses speculation to generate a stream of values that keep the pipeline full. The newly generated circuit retains enough state to recover from incorrect speculations by flushing the incorrect values from the pipeline, restoring the correct state, then restarting the computation.

We also implement two extensions to this basic approach: stalling, which minimizes circuit area by eliminating speculation, and forwarding, which increases the throughput of the generated circuit by forwarding correct values to preceding pipeline stages. We have implemented a prototype synthesizer based on this approach. Our experimental results show that, starting with a non-pipelined or insufficiently pipelined specification, this synthesizer can effectively reduce the clock cycle time and improve the throughput of the generated circuit.

Keywords

Pipeline, modular, speculation, stall, forward

1. INTRODUCTION

This paper presents a new algorithm for automatically pipelining sequential circuits. The algorithm is based on speculation and uses state retention and recovery to respond to incorrect speculations. The paper also presents two extensions to the basic approach: generating stall logic to avoid incorrect speculations and the associated area penalty, and generating forwarding logic to increase the throughput of the resulting circuit.

Our algorithm starts with a non-pipelined or insufficiently pipelined specification of a circuit and repeatedly shortens its clock cycle by extracting a computation from the critical path and moving it into a new pipeline stage. The new stage precomputes the result of the selected expression and passes it to the computation of the next stage that uses it. To keep the pipeline full, the new stage must produce the next value of the expression before the final values of the variables it accesses become available. Our algorithm achieves this goal by speculating on the values of these variables. If the speculation is incorrect, the circuit restores its state to match the state before the speculation, flushes the incorrect values from the pipeline, then restarts the computation.

Our algorithm uses several techniques to improve the quality of the pipelined circuit. If the amount of state necessary to recover from an incorrect speculation is excessive, our algorithm can generate stall logic that causes the pipeline stage to stall until the new values are available. This technique eliminates the need for retaining recovery state, as the execution of the pipeline stage will never need to roll back. Our algorithm also generates circuits that forward the correct value to preceding pipeline stages. This technique increases the throughput of the circuit by reducing the amount of time that the circuit spends recovering from incorrect speculations or waiting for correct values to become available.

We have built a prototype implementation of our algorithm. Using our synthesizer [13] as backend, this implementation generates synthesizable Verilog at the RTL level. We have used our implementation to automatically generate pipelined versions of several circuits. Our results show that our automatically generated pipelined circuits are competitive with hand-generated versions.

This paper makes the following contributions:

- **Approach:** It presents a new approach for automatically pipelining sequential circuits. This approach repeatedly extracts a computation from the critical path and moves it into a new stage. This stage uses speculation to generate a stream of values that keep the pipeline full. The approach reduces the clock cycle and increases the throughput of a circuit.
- **Algorithm:** It presents a pipelining algorithm that implements our approach. It also presents two extensions to the approach: stalling, which reduces the amount of area that would otherwise be required to respond to incorrect speculations; and forwarding, which increases throughput either by replacing values produced by incorrect speculations with correct values or by making new values available earlier to the stall logic.
- **Experimental Results:** It presents experimental results that prove the viability of the approach in practice.

The remainder of the paper is organized as follows. Section 2 discusses related work. Section 3 illustrates how a system is specified using rewrite rules and gives an example

[*]This research was supported in part by NSF Grant CCR-9702297.

of what the starting and derived circuit specifications may look like. Section 4 presents the pipelining algorithm. Section 5 presents the experimental results. Section 6 draws the conclusions.

2. RELATED WORK

Many high-level synthesis systems focus on the automatic generation of highly efficient pipelined designs. Most of this work is primarily concerned with functional pipelining. Many synthesis tools target instruction-set architectures[10, 4, 3, 14, 15, 1, 9, 2]; our tool, on the other hand, targets the more general class of sequential circuits. Other approaches start with a C program [8, 5].

Kroening and Paul [11] describe a method of automating the generation of stall and forward logic starting from a given sequential machine. Starting from hardware that is initially partitioned into pipeline stages, the algorithm produces a circuit that can stall in any arbitrary stage while keeping the other stages running, if possible. To implement forwarding, the designer has to specify the registers holding intermediate results that need to be forwarded to previous stages in the pipeline. The goal of our research, in contrast, is to completely automate the pipelining transformation.

Retiming [12] optimally pipelines combinatorial circuitry. Architectural retiming [6] adds a negative/normal register pair on a latency-constrained path, effectively pipelining the logic without adding latency. It implements the negative register by either precomputation or prediction of the value that it produces.

Research by Hoe and Arvind [7] and Shen and Arvind [16] has introduced an approach to describe, verify and synthesize processors based on term rewriting systems (TRS). They do not implement automatic pipelining, but their specification language offers comparable capabilities in this direction as our language.

3. EXAMPLE

The text inside the boxes in Figure 2 and Figure 1 presents a specification written in our high-level description language. The figures also contain a graphical representation that we find useful in explaining our example. We first present a non-pipelined datapath, then a simple, three-stage, linear pipelined datapath that our algorithm can automatically derive from the non-pipelined specification. Section 4 shows the intermediate specifications at each step of the algorithm. We chose to present this three-stage linear pipeline because of its simplicity; our algorithm is capable of generating deep pipelines and is not specific to this particular class of circuits.

The designer specifies the circuit using two kinds of information:

- **State Declarations:** The designer specifies the state of the system as a set of typed variable declarations.
- **Module Specification:** The designer specifies the behavior of each module as a set of update rules. Modules communicate by reading and writing shared state and particularly using FIFO queues.

3.1 Modules

Figure 2 and Figure 1 show the functional modules in our non-pipelined and respectively, pipelined, examples and the queues that interconnect them. Each module consists of a set of *update rules*. An update rule has an enabling condition and a set of updates to the state. When the enabling condition evaluates to `true`, the rule is enabled and can execute, in which case its updates are *atomically* applied to the state. Conceptually, the execution of the system repeatedly chooses an enabled rule and executes it. In practice, our backend synthesizer analyzes the specification to execute multiple rules in parallel in the same clock cycle [13].

Queues provide buffered, first-in, first-out connections between modules. Modules can perform several operations on a queue `q`:

- `head(q)`: Retrieves the first element in the queue.
- `tail(q)`: Returns the rest of `q` after the first element.
- `insert(q,e)`: Returns the queue `q` after inserting the element `e` at the end of `q`.
- `replace(e1,e2,q)`: Returns `q` after replacing all entries `e1` by `e2`.
- `notin(q,e)`: Returns *true* if the element `e` is not in `q`; otherwise returns *false*.
- `q = nil`: Resets the queue to be empty.

We next illustrate the conceptual model of execution in our system by discussing the operation of the rules in our example. The rules describe the structure of the hardware pipeline and *not* the program that executes on it.

3.1.1 *Non-pipelined Speci£cation*

```
<INC r> = im[pc] ->
    rf = rf[r->rf[r]+1], pc = pc+1;
<JRZ r l> = im[pc] and rf[r] = 0 ->
    pc = l;
<JRZ r l> = im[pc] and rf[r] != 0 ->
   pc = pc+1;
```

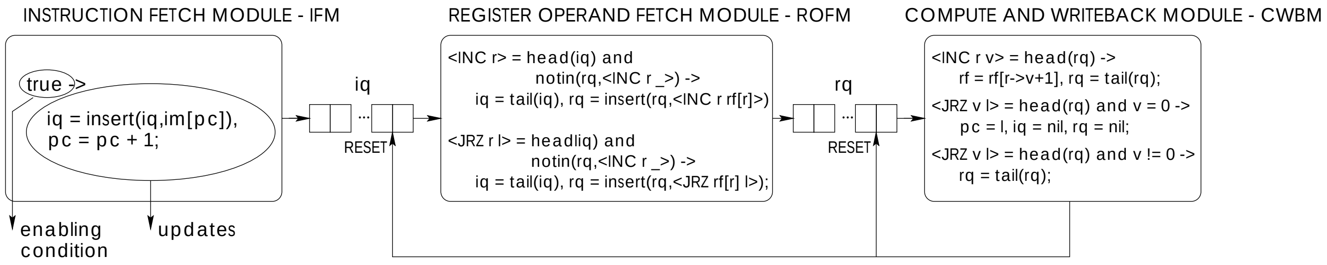
Figure 2: Non-pipelined Specification

The first rule in Figure 2 processes `INC` instructions. The rules use a form of pattern matching similar to that found in ML and Haskell. If the rule's enabling condition is true, the clause matches and binds the variable `r` to the register name argument of the `INC` instruction. The rule can then use `r` to refer to this argument. If enabled, the rule atomically executes the block in the right-hand-side of the arrow. The update `rf = rf[r->rf[r]+1]` sets element `r` of the register file to be `rf[r]+1`. The other rules perform similar actions. To keep the example clear, the instruction set contains only an `INC` instruction, which increments the value in its single register argument, and a `JRZ` instruction, which tests the value in its register argument and, if the value is zero, jumps to the location in its location argument.

3.1.2 *Three-stage Pipelined Speci£cation*

The condition for the rule in module IFM of Figure 1 is `true`, which means that the rule is always enabled. When it executes, it fetches an instruction from the instruction memory and inserts it into the instruction queue `iq`. It also increments the program counter `pc` to set up the next fetch.

The two rules in the module ROFM remove instructions from `iq`, fetch the register operands, and insert them into `rq`. The enabling condition of the first rule is `<INC r> = head(iq) and notin(rq, <INC r _>)`. If the instruction at


INSTRUCTION FETCH MODULE - IFM
REGISTER OPERAND FETCH MODULE - ROFM
COMPUTE AND WRITEBACK MODULE - CWBM
true ->
iq
rq
RESET
RESET
enabling condition
updates

```
iq = insert(iq,im[pc]),
pc = pc + 1;

<INC r> = head(iq) and
            notin(rq,<INC r _>) ->
    iq = tail(iq), rq = insert(rq,<INC r rf[r]>)

<JRZ r l> = headliq) and
            notin(rq,<INC r _>) ->
    iq = tail(iq), rq = insert(rq,<JRZ rf[r] l>);

<INC r v> = head(rq) ->
    rf = rf[r->v+1], rq = tail(rq);
<JRZ v l> = head(rq) and v = 0 ->
    pc = l, iq = nil, rq = nil;
<JRZ v l> = head(rq) and v != 0 ->
    rq = tail(rq);
```

Figure 1: Three-stage Pipelined Specification

the head of `iq` is an `INC`, the clause matches and binds `r` to the register name argument of the `INC` instruction. The second clause, `notin(rq, <INC r _>)` uses the binding `r` to check for a read after write (RAW) hazard caused by a pending instruction in `rq` that will write the register `r`. In this case, the machine delays the operand fetch so that it fetches the value after the write (this translates into `stalling`[1]). The clause `notin(rq, <INC r _>)` checks to make sure that there is no such instruction in `rq`, and the rule as a whole is enabled and can execute only if there is no hazard.

The other rules perform similar actions. The update `iq/rq = nil` clears the queue(s) `iq/rq`.

3.2 State

3.2.1 Non-pipelined Speci£cation

Figure 3 presents the state and type declarations for Figure 2.

```
1 type reg = int(3), val = int(8), loc = int(8);
2 type ins = <INC reg> | <JRZ reg loc>;

4 var pc :  loc, im :  ins[N], rf :  val[8];
```

Figure 3: State Variables and Type Declarations for Example in Figure 2

Line 4 in Figure 3 presents the state declarations, which consist of the following state variables: a program counter `pc`, an instruction memory `im` and a register file `rf`. Lines 1 and 2 contain the type declarations for these variables. The type declarations include a 3 bit register name type `reg`, an 8 bit integer type `val`, an 8 bit integer type `loc` which represents the locations of instructions in the instruction memory and an instruction type `ins`. The instruction type is a tagged union type, similar to those found in ML and Haskell and contains only an `INC` and a `JRZ` instruction.

3.2.2 Three-stage Pipelined Speci£cation

Figure 4 presents the state and type declarations for Figure 1. Line 5 in Figure 4 declares the two queues, `iq` and `rq`. Line 3 contains the type declaration for instructions whose register operands have been fetched from the register file. The rest of the declarations are identical to the ones in Figure 3.

4. PIPELINING ALGORITHM

[1]We chose here to present stalling for the sake of simplicity of the presentation.

```
1 type reg = int(3), val = int(8), loc = int(8);
2 type ins = <INC reg> | <JRZ reg loc>;
3 type irf = <INC reg val> | <JRZ val loc>;

4 var pc :  loc, im :  ins[N], rf :  val[8];
5 var iq = queue(ins), rq = queue(irf);
```

Figure 4: State Variables and Type Declarations for Example in Figure 1

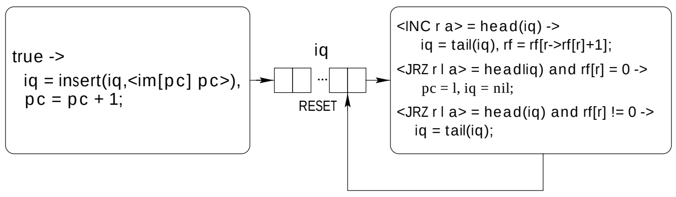
4.1 Basic Approach

The pipelining algorithm starts with a non-pipelined or insufficiently pipelined specification and automatically generates a highly-pipelined, functionally equivalent specification. The algorithm repeatedly extracts an expression from a target module and creates a new module to compute the value of the expression at each clock cycle. It then uses a stream to transport the computed values from the new module into the target module from which the expression was extracted. The length of the stream is conceptually unbounded. The synthesis algorithm we developed in [13] implements all the streams in the final specification as finite hardware buffers. This algorithm operates on the resulting specification once the pipelining algorithm has finished. Our pipelining algorithm transforms the target module so that it reads the value of the expression from the stream instead of computing its value. Because this transformation splits computations across multiple clock cycles, it may reduce the clock cycle of the circuit and increase its throughput.

In general, the extracted subcomputation may depend on values that are not available until after it must produce the new value. The compiler therefore speculates on the values that the subcomputation uses. If the speculation is incorrect, the circuit restores the values of any incorrectly updated variables and restarts the computation from the restored state. To enable the restoration, the transformed specification inserts the old values of any potentially incorrectly updated variables into the new stream. When the circuit encounters an incorrect speculation, it extracts these values from the stream and uses them to restore any incorrectly updated variables to their correct values. The algorithm consists of seven phases for each further pipelining decision. The steps are illustrated using Figure 2, Figure 1 and Figure 6. Figure 6 is the intermediate two-stage pipelined specification derived from Figure 2 by applying the algorithm once.

- **Select Target Expression:** The selection of the target expression is driven by an analysis of the combinational path lengths in the circuit. The algorithm repeatedly determines the critical path, then chooses a target expression on this critical path. Inserting the computation of the target expression into a different stage of the pipeline removes the expression from the critical path, shortening its length. A more general approach could be implemented that uses a wider set of paths than the critical one(s) in deciding which expression to select as target. In addition to selecting the target expression automatically, our implemented system also allows the designer to drive the pipelining process by manually selecting the target expression. For Figure 2 the algorithm selects `im[pc]`; for Figure 6 the target expression is `rf[r]`.

- **Compute All Involved Variables:** The value of the target expression depends on the variables that it references. This set of variables is called the *set of involved variables.*
  For the specification in Figure 2, the set of all involved variables of `im[pc]` is {`im,pc`}. For Figure 6, the set of all involved variables of `rf[r]` is {`rf,head(iq)`}.
- **Speculate on New Values of Involved Variables:** The pipelining algorithm will move the computation of the target expression into a new module. This module will compute the value of the target expression in a clock cycle before the final values of the involved variables have been determined. The module therefore speculates on the final values of these variables, using the speculated values to compute the value of the target expression. There are two kinds of speculation:
  - **Control:** Speculate on which rule will fire. For the involved variable `pc` in Figure 2 we speculate that the first rule will fire. This choice implies that the new value of `pc` is `pc+1`. For `iq` in Figure 6 we speculate that the first rule in the rightmost box will fire, so `iq`'s new speculated value is `tail(iq)`.
  - **Data Hazard:** Speculate on the absence of data hazards. For involved variable `rf` in Figure 6, we speculate that there will be no writes to `rf[r]`.
- **Generate a Stream of Values:** Generate a stream containing the sequence of values of the target expression and transform the specification to use these values. For each rule that originally read the target expression:
  - Modify the rule so that it now reads from the head of the new stream.
  - Replace all occurrences of the expression with the value read from the stream.
- **Update Involved Variables:** Augment the new module to update the involved variables with their speculated values. This operation ensures that, during the next clock cycle, the specification will generate an appropriate next value for the target expression.
  Figure 6 presents the results of this transformation for the specification in Figure 2 and target expression `im[pc]`. Figure 1 presents the results of this transformation for the specification in Figure 6 and target expression `rf[r]`.
- **Augment Stream to Handle Failed Speculations:** If the speculation is incorrect, the specification must restore the *updated variables* (the set of all variables updated as a result of the speculation) to their correct values. The algorithm therefore augments the generated stream with the values of the updated variables before the speculation. The set of updated variables for `im[pc]` is {`pc`}. The set of updated variables for `rf[r]` is {`rf,iq`}.
  The algorithm transforms the specification to detect incorrect speculations and, when necessary, use the values in the stream to restore the correct state of the system and clear the stream. The system will therefore restart from a consistent state.
  For the non-pipelined example in Figure 2, Figure 5 shows the resulting specification after the algorithm executes this step.

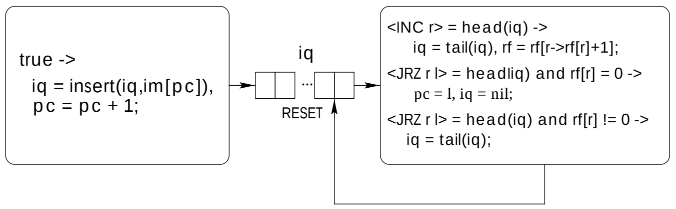


```
true ->
  iq = insert(iq,<im[pc] pc>),
  pc = pc + 1;

<INC r a> = head(iq) ->
    iq = tail(iq), rf = rf[r->rf[r]+1];
<JRZ r l a> = headliq) and rf[r] = 0 ->
    pc = l, iq = nil;
<JRZ r l a> = head(iq) and rf[r] != 0 ->
   iq = tail(iq);
```

Figure 5: Specification After Stream Augmentation for Handling Failed Speculations

- **Remove Unused Values from Stream:** After updating all the rules that read the target expression, eliminate all the fields of stream entries that were saved and never used again.
  As we notice in Figure 5, field `a` of `iq` is never used, so we can safely remove it from the stream to obtain the specification in Figure 6.



```
true ->
  iq = insert(iq,im[pc]),
  pc = pc + 1;

<INC r> = head(iq) ->
    iq = tail(iq), rf = rf[r->rf[r]+1];
<JRZ r l> = headliq) and rf[r] = 0 ->
    pc = l, iq = nil;
<JRZ r l> = head(iq) and rf[r] != 0 ->
   iq = tail(iq);
```

Figure 6: 2-stage Intermediate Specification

4.2 Optimizations

We next present how our algorithm generates logic that implements two techniques — stalling and forwarding — which can improve the quality and performance of the automatically pipelined circuit.

We first present the circuit that responds to incorrect speculations by restoring saved state. Figure 7 shows the transformation schema for a rule R_i

$$R_i\text{:}\quad \texttt{e = head(str) and } P_i \rightarrow A_i$$

that reads some target expression `TE`. `e = head(str) and` P_i is the enabling condition of R_i; both clauses are optional. A missing clause reads as a true clause. A_i consists of all the updates performed by rule R_i.

In Figure 7, `IV` stands for the set of involved variables of `TE` and `UV` for the corresponding set of updated variables. `IV` is the disjunct union of two subsets: IV_{ctrl} — the set of involved variables on which the algorithm applies control speculation, and IV_{DH} — the set of involved variables on which it applies data hazard speculation. The speculated value of `TE` is `TE'`, and the set of speculated values for the elements of IV_{ctrl} is IV'_{ctrl}. A_i`.upd(IV)` returns the set of values that A_i updates the involved variables in `IV` to.

A_i(IV) returns the updates in A_i that write the involved variables in IV. newstr is the newly generated stream of values for target expression TE. dataHazard(E) returns true if head(newstr) writes the expression E in the current clock cycle and at least one entry in tail(newstr) reads it.

```
R_i1:  e1 = head(str) →
    newstr = insert(newstr,<e1 TE[IV'ctrl/IVctrl] UV>),
    IVctrl = IV'ctrl;
If control speculation:
If Ai.upd(IV) = IV'ctrl then:
  Ri2:  <e2 TE2 UV2> = head(newstr) and
                  Pi[UV2/UV,TE2/TE] →
      newstr = tail(newstr),
      Ai[UV2/UV,TE2/TE] \ Ai(IV);
Otherwise:
  Ri2:  <e2 TE2 UV2> = head(newstr) and
                  Pi[UV2/UV,TE2/TE] →
      UV = UV2,         // restore UV
      newstr = nil,      // clear stream
      Ai;               // update
If data hazard speculation:
  R'i2:  <e2 TE2 UV2> = head(newstr) and
                  Pi[UV2/UV,TE2/TE] and
                  no dataHazard(TE ∪ IVDH) →
      newstr = tail(newstr),
      Ai[UV2/UV,TE2/TE] \ Ai(IV);
  R''i2:  <e2 TE2 UV2> = head(newstr) and
                  Pi[UV2/UV,TE2/TE] and
                  dataHazard(TE ∪ IVDH) →
      UV = UV2,         // restore UV
      newstr = nil,      // clear stream
      Ai;               // update
```

Figure 7: Restoration Schema

Assume we apply this transformation to the specification in Figure 6, for TE = rf[r]. Figure 8 presents the INC instruction that this transformation generates. The first clause of each newly derived rule reads the head of rq and binds r, x and s to the arguments of the INC instruction. The clause notin(tail(rq),<_ r _ _>) checks if there is an entry in tail(rq) that reads rf[r]. This is the check for a data hazard on rf[r] for the INC instruction. In case of a hazard — second rule in Figure 8 — all the updated variables, in our case iq, have to be restored to their old values. We would like to avoid having to store and carry the whole instruction stream iq along, up to the restoration point. In this case, a viable approach is stalling the pipeline stage that reads rf[r] until all data hazards are cleared.

```
<INC r x s> = head(rq) and
              notin(tail(rq),<_ r _ _>) →
    rf = rf[r->x+1], rq = tail(rq);
<INC r x s> = head(rq) and
              ~notin(tail(rq),<_ r _ _>) →
    iq = tail(s), rq = nil, rf = rf[r->x+1];
```

Figure 8: Roll-back Scheme for INC instructions

4.2.1 Stalling

Let S be the set of all target expressions on which the algorithm speculates. For TE ∈ S, let $\{R_i\}$ be the set of rules that generate the speculated values of TE and let this stream be Q_1. Let $\{Q_n\}$ be the set of streams that the rules that write TE read from. To eliminate the need to restore state in case of a failed speculation on TE, the algorithm modifies the preconditions of all rules in $\{R_i\}$ to check that either 1) no item in any stream from Q_1 to $\{Q_n\}$ will generate a write to TE or 2) all items that update TE write the same known value. This approach implements the *stalling* mechanism; Figure 1 presents its results for our example.

Let R_{21} and R_{22} be the two resulting rules from splitting the rule in Figure 6 handling INC instructions. The check for data hazards is now handled by R_{21}, which will not fire and read the target expression rf[r] until all the previous rules writing rf[r] did so:

```
R21:<INC r> = head(iq) and notin(rq,<INC r _>) →
        iq = tail(iq),
        rq = insert(rq,<INC r rf[r]>);
```

Rule R_{22} does not anymore need to check for data hazards for the current target expression — in our example rf[r]. There is no speculation on fly regarding the absence of a data hazard and therefore no need to save the instruction queue in the newly generated stream of values for rf[r]. The derived R_{22} will have the form below:

```
R22:<INC r v> = head(rq) →
        rf = rf[r->v+1], rq = tail(rq);
```

Stalling trades the potentially higher throughput of speculative execution for a smaller circuit area. This trade-off requires a policy to decide when it is better to stall the pipeline or when it is better to speculate. The decision depends primarily on the accuracy of the predictions and the amount of state that needs to be saved for restoration purposes. Also, if all all the rules ahead in the pipeline will update TE with the same known value, the circuit can safely generate the next value of TE even if some rule will write TE. Therefore, a stalling check replacing a data hazard speculation waits until TE is hazard-free; a stalling check replacing a control speculation waits until all the previous rules in the pipeline can only update TE with the same known value.

4.2.2 Forwarding

Regardless of whether the algorithm speculates on the value of TE or stalls the pipeline, waiting for its correct value to become available, generating *forwarding* logic may increase the throughput of the circuit. To implement forwarding, the algorithm replaces the obsolete values of TE in Q_1 with their correct, updated values. Figure 10 shows how the technique updates a rule to implement the bypass. updatedTE stands for the newly computed, correct value of TE.

```
<INC r x> = head(rq) →
    rq = replace (<INC r _>,<INC r x+1>,
         replace (<JRZ r _ _>,<JRZ r _ x+1>,
         tail(rq))),
    rf = rf[r->x+1];
```

Figure 9: Forward Scheme for INC instructions

Forwarding transforms the two rules handling INC instructions in Figure 8 into the single new rule in Figure 9. This rule produces a circuit that updates all of the entries in **rq** produced by rules that accessed `rf[r]` with the new correct value `x+1`.

```
$R^{'}_{i1}$:  $e_1$= head(str) and dataHazard(TE) $\longrightarrow$
    newstr = insert(newstr,<$e_1$ updatedTE UV>),
    $IV_{ctrl}$ = $IV^{'}_{ctrl}$;
$R^{''}_{i1}$:  $e_1$ = head(str) and no dataHazard(TE) $\longrightarrow$
    newstr = insert(newstr,<$e_1$ TE[$IV^{'}_{ctrl}$/$IV_{ctrl}$] UV>),
    $IV_{ctrl}$ = $IV^{'}_{ctrl}$;
If control speculation:
If $A_i$.upd(IV) = $IV^{'}_{ctrl}$ then:
  $R_{i2}$:  <$e_2$ $TE_2$ $UV_2$> = head(newstr) and
                  $P_i$[$UV_2$/UV,$TE_2$/TE] $\longrightarrow$
      newstr = tail(newstr)[updatedTE/$TE_2$],
      $A_i$[$UV_2$/UV,$TE_2$/TE] $\setminus$ $A_i$(IV);
Otherwise:
  $R_{i2}$:  <$e_2$ $TE_2$ $UV_2$> = head(newstr) and
                  $P_i$[$UV_2$/UV,$TE_2$/TE] $\longrightarrow$
      UV = $UV_2$,         // restore UV
      newstr = nil,        // clear stream
      $A_i$;               // update
If data hazard speculation:
  $R^{'}_{i2}$:  <$e_2$ $TE_2$ $UV_2$> = head(newstr) and
                  $P_i$[$UV_2$/UV,$TE_2$/TE] and
                  no dataHazard($IV_{DH}$) $\longrightarrow$
      newstr = tail(newstr)[updatedTE/$TE_2$],
      $A_i$[$UV_2$/UV,$TE_2$/TE] $\setminus$ $A_i$(IV);
  $R^{''}_{i2}$:  <$e_2$ $TE_2$ $UV_2$> = head(newstr) and
                  $P_i$[$UV_2$/UV,$TE_2$/TE] and
                  dataHazard($IV_{DH}$) $\longrightarrow$
      UV = $UV_2$,         // restore UV
      newstr = nil,        // clear stream
      $A_i$;               // update
```

Figure 10: Forward Scheme

Forwarding reduces the amount of time that the circuit spends recovering from incorrect speculations or waiting for correct values to become available. It may therefore increase the throughput of the circuit.

5. EXPERIMENTAL RESULTS

We have implemented the pipelining algorithm within our prototype synthesizer, which generates synthesizable Verilog implementations at the RTL level. We then by compared the results obtained by our algorithm against a hand-written version that implements the same basic functionality. We wrote a non-pipelined specification of a 32-bit datapath processor with a complete instruction set[2] and ran it through our pipelining algorithm. The resulting pipelined specification was then fed into the synthesizer to obtain a Verilog model for it. This model was then synthesized using the Synopsis Design Compiler to an industry standard .25 micron standard cell process. To serve as a reference point, we also synthesized, in the same environment, the Santa Clara University SCU RTL 98 DSP, a hand-written (in Verilog), standard 32-bit fixed point DSP that implements the same basic functionality. Our automatically pipelined version had a cycle time of 88.9 MHz as opposed to a 90.9 MHz cycle time for the hand-pipelined version; the synthesized areas were virtually identical.

It took us approximately fifteen minutes to write the specification for the non-pipelined processor and less than one minute to run it through our pipeline algorithm. Our specification contains 7 lines for state declarations and 10 lines of rule definitions for module specifications. Our automatically generated implementation consists of about 1200 lines of synthesizable Verilog. We tested the generated Verilog model using the Cadence NCVerilog simulator.

[2]The instruction set contains load, store, jump, ALU, multiply and variable shift operations, but no division.

6. CONCLUSIONS

This paper presents a new approach for automatically pipelining sequential circuits: repeatedly extract a computation from the critical path, move it into a new stage, then use speculation to generate a stream of values that keep the pipeline full. We also present extensions that integrate stalling and forwarding into this basic approach. Our experimental results provide encouraging evidence that the approach can deliver efficient pipelined implementations.

7. REFERENCES

[1] M. Breternitz and J. P. Shen. Architecture synthesis of high-performance application-specific processors. In *Proceedings of the 27th ACM/IEEE Design Automation Conference*, 1990.

[2] F. Chang and A. Hu. Fast specification of cycle-accurate processor models. *To appear in ICCD 2001*, 2001.

[3] R. J. Cloutier and D. E. Thomas. Synthesis of pipelined instruction set processors. In *Proceedings of the 30th ACM/IEEE Design Automation Conference*, 1993.

[4] H. D. M. G. Goossens, J. Rabaey and J. Vandewalle. An efficient microcode-compiler for custom DSP-processors. *IEEE Transactions on Computer-Aided Design*, 9:925–937, 1990.

[5] G. Goossens, J. Vandewalle, and H. DeMan. Loop optimization in register-transfer scheduling for DSP-systems. In *Proceedings of the 26th ACM/IEEE Design Automation Conference*, 1989.

[6] S. Hassoun and C. Ebeling. Architectural retiming: Pipelining latency-constrained circuts. In *Proceedings of the 33rd ACM/IEEE Design Automation Conference.*

[7] J. Hoe and Arvind. Hardware synthesis from term rewriting systems. In *VLSI: Systems on a chip*, Lisbon, Portugal, Dec. 1999.

[8] U. Holtmann and R. Ernst. Combining MBP-speculative computation and loop pipelining in high-level synthesis. In *ED & TC*, 1995.

[9] I.-J. Huang and A. M. Despain. High level synthesis of pipelined instruction set processors and back-end compilers. In *Proceedings of the 29th ACM/IEEE Design Automation Conference*, 1992.

[10] C.-T. Hwang, Y.-C. Hsu, and Y.-L. Lin. Scheduling for functional pipelining and loop winding. In *Proceedings of the 28th ACM/IEEE Design Automation Conference.*

[11] D. Kroening and W. Paul. Automated pipeline design. In *Proceedings of the 38th ACM/IEEE Design Automation Conference*, Las Vegas, 2001.

[12] C. E. Leiserson, F. Rose, and J. Saxe. Optimizing synchronous circuitry by retiming. In *Proceedings of the 3rd Caltech Conference on VLSI.*

[13] M.-C. Marinescu and M. C. Rinard. High-level specification and efficient implementation of pipelined circuits. In *Proceedings of the ASP-DAC*, 2001.

[14] N. Park and A. C. Parker. Sehwa: A software package for synthesis of pipelines from behavioral specifications. *IEEE Transactions on Computer-Aided Design*, 7:356–370, 1988.

[15] P. B. Paulin and J. P. Knight. Force-directed scheduling for the behavioral synthesis of ASIC's. *IEEE Transactions on Computer-Aided Design*, 8:661–679, 1989.

[16] X. Shen and Arvind. Using term rewriting systems to design and verify processors. *IEEE Micro Special Issue on "Modeling and Validation of Microprocessors"*, 1999.