Synchronization-at-Retirement for Pipeline Verification

Mark D. Aagaard[1], Nancy A. Day[2], and Robert B. Jones[3]

[1] Electrical and Computer Engineering, University of Waterloo, Waterloo, ON, Canada
markaa@swen.uwaterloo.ca
[2] Computer Science, University of Waterloo
nday@cs.uwaterloo.ca
[3] Strategic CAD Labs, Intel Corporation, Hillsboro, OR, USA
rjones@ichips.intel.com

Abstract. Much automatic pipeline verification research of the last decade has been based on some form of "Burch-Dill flushing" [BD94]. In this work, we study *synchronization-at-retirement*, an alternative formulation of correctness for pipelines. In this formulation, the proof obligations can also be verified automatically but have significantly-reduced verification complexity compared to flushing. We present an approach for systematically generating invariants, addressing one of the most difficult aspects of pipeline verification. We establish by proof that synchronization-at-retirement and the Burch-Dill flushing correctness statements are equivalent under reasonable side conditions. Finally, we provide experimental evidence of the reduced complexity of our approach for a pipelined processor with ALU operations, memory operations, stalls, jumps, and branch prediction.

1 Introduction

Many different strategies to verify pipelines have been documented in the literature. The capacity of verification strategies tends to decrease with the amount of automation. Yet, automation makes formal verification more practical and extends its "reach" in practice. A seminal paper in 1994 by Burch and Dill [BD94], introduced *flushing*, an approach that computes automatically an abstraction function from a pipelined implementation to a specification that executes instructions as atomic operations. The essential idea is that partially-completed instructions in the pipeline can finish executing by introducing *bubbles* (NOPs that don't increment the program counter) into the front of the pipeline until the pipeline is empty. At that point, the architectural state of the pipeline can be compared against the specification.

Two primary difficulties with Burch-Dill flushing have emerged. First, the computational complexity of flushing a pipeline full of symbolic instructions quickly becomes prohibitive as pipelines grow in size and complexity. The complexity arises because the implementation must be stepped until all in-flight instructions have retired. Second, finding invariants to characterize the reachable pipeline states is a difficult and mostly manual process. Much of the recent research on pipeline processor verification has used manual decomposition to circumvent the computational complexity of flushing. A common approach is to prove a collection of invariants that guarantee *flushpoint equality*: when the implementation is in a flushed state, the architectural state agrees with the specification state.

Specifications execute instructions atomically, while pipelined implementations execute several instructions simultaneously. When verifying a single step of the implementation, embedded in the correctness statement is a choice about which one of the in-flight instructions should be compared against the specification. This choice defines the *synchronization point* between the implementation and specification. The difficulty in choosing a synchronization point is that an instruction writes to different pieces of architectural state at different stages in the pipeline. For example, the program counter for fetching instructions and the register file are consistent only if the pipeline is flushed.

The two common synchronization points are fetch (*sync-at-fetch*) and retire (*sync-at-retire*). Burch-Dill flushing is a sync-at-fetch approach: the effects of the instruction that is fetched are compared to the result of executing the instruction sequentially in the specification. In this paper, we study sync-at-retire: only instructions that retire will be compared to the specification. At first glance, this may appear to be vacuous – how can a pipeline be verified if the correctness statement ignores the pipeline contents? The answer lies in the assumptions that we must make to prove sync-at-retire.

Determining the assumptions and proving that they are invariant is where the reasoning about the rest of the pipeline occurs. We present an approach in which these invariants are generated systematically based on the structure of the pipeline using history variables and completion functions [HGS03]. The verification of the individual invariants is modular and regular, and the validity of the decomposition is visible by inspection.

Using sync-at-retire as a correctness statement addresses both of the weaknesses of Burch-Dill flushing. First, because the sync-at-retire correctness statement and the necessary invariants do not require the entire pipeline be flushed, the verification complexity is significantly reduced. Second, our approach provides a method for the systematic creation of invariants.

This paper contains three main contributions:

1. A systematic technique for generating and proving the invariants necessary to prove sync-at-retire so that all proof obligations involve taking only a single-step of the implementation, extending the reach of automatic verification tools,
2. Proving that sync-at-retire is an equivalent correctness statement to sync-at-fetch under reasonable conditions about the flushing function, and
3. Demonstrating on several pipeline variants that verifying sync-at-retire (and the necessary invariants) is computationally more efficient than verifying sync-at-fetch.

Our approach is applicable to safety verification for the control logic of pipelines. By safety, we mean that any step of the implementation corresponds to a step of the specification. We use a simple processor and processor-related terms to illustrate our approach. Processor pipelines provide good pedagogical examples for this class of verification techniques, but these techniques are certainly not limited to processors. Realistic applications for our approach include simple embedded processors or sub-pipelines within processors.

The paper is organized as follows. Section 2 provides formal definitions for sync-at-fetch and sync-at-retire. In Section 3, we detail our sync-at-retire approach and demonstrate how we derive and verify the needed invariants in a systematic way. In Section 4, we prove that, under reasonable side conditions, verifying sync-at-retire is equivalent to verifying sync-at-fetch. Section 5 reports results of verifying example pipelines with both sync-at-fetch and sync-at-retire. The discussion of related work is deferred to Section 6, where it is presented in the context of our results. Section 7 summarizes the paper and considers future directions.

2 Synchronize, but Where?

The specification of a processor is described by an instruction set architecture (ISA) that executes instructions atomically. The goal of pipelined processor safety verification is to show that all execution traces of a pipeline correspond to a trace of the specification. A commonly-used method of demonstrating this correspondence is to find a simulation relation between pipeline and specification states such that one step of the pipeline corresponds to one step of the specification [Mil71].

The novelty of the Burch-Dill approach was in creating a simulation relation automatically using an abstraction function that maps pipeline state to ISA state. The abstraction function first completes the execution of every operation in the pipeline (flushing) and then projects only the ISA state elements from the implementation to compare with the specification. We characterize this correctness statement as synchronization-at-fetch (*sync-at-fetch*) because it compares the effects of executing the *fetched* instruction with ISA execution of the same instruction. Figure 1 shows the commuting diagram for the sync-at-fetch correctness statement. Table 1 shows the notation used in our descriptions. For a deterministic implementation and specification, sync-at-fetch reduces to:

$$\forall\, q_i.\ \pi_f(\mathit{flush}\ (n_i\ q_i)) = n_s\ (\pi_f\ (\mathit{flush}\ q_i)).$$

To facilitate verification, the processor datapath is usually abstracted away with *uninterpreted functions*, leaving the datapath verification to be handled separately. The correctness statement is typically checked using decision procedures such as UCLID [LSB02], CVC Lite [BB04], and SVC [BDL96]. For deep and/or complex pipelines, Burch-Dill flushing suffers from rapid growth in the complexity of the terms that arise from completing the execution of every operation in the pipeline.

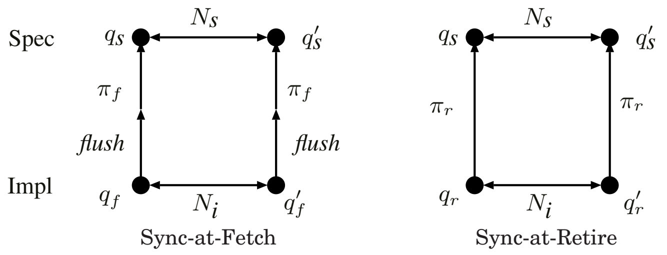

Fig. 1. Correctness Statements

An alternative correctness formulation, synchronization-at-retirement (*sync-at-retire*), compares only the result of executing the instruction about to *retire* (complete) with the specification executing the same instruction. The benefit of this correct-

Table 1. Notation

$Q_i,\ Q_s$	Sets of implementation and specification states
$N_s : Q_s \rightarrow Q_s \rightarrow \mathit{Bool}$	Next-state relation for specification (Relations are shown as two-way arrows in commuting diagrams.)
$n_s : Q_s \rightarrow Q_s$	Next-state function for specification
$N_i : Q_i \rightarrow Q_i \rightarrow \mathit{Bool}$	Next-state relation for implementation
$n_i : Q_i \rightarrow Q_i$	Next-state function for implementation
$\pi_f : Q_i \rightarrow Q_s$	Project at fetch
$\mathit{flush} : Q_i \rightarrow Q_i$	Flush an implementation state
$\mathit{isFlushed} : Q_i \rightarrow \mathit{Bool}$	Implementation state has no in-flight instructions
$\mathit{numFetch} : Q_i \rightarrow Q_i \rightarrow \mathit{nat}$	Number of instructions fetched in a step
$\pi_r : Q_i \rightarrow Q_s$	Project at retire
$\mathit{kill} : Q_i \rightarrow Q_i$	Kill all in-flight instructions
$\mathit{numRetire} : Q_i \rightarrow Q_i \rightarrow \mathit{nat}$	Number of instructions retired in a step
$\mathit{numWillRetire} : Q_i \rightarrow \mathit{nat}$	The number of in-flight instructions that will eventually retire.
π	$= \pi_f = \pi_r$ when in a flushed state
$\mathit{pc},\ \mathit{rf},\ \mathit{mem},\ \mathit{pc}_r$	Fetch program counter, register file, memory, retirement program counter

ness statement is that the simulation relation involves simply discarding all partially-completed operations, and projecting the specification state elements from the implementation to compare with the specification. To implement precise exceptions, pipelines can only write results to programmer-visible (specification) state when instructions complete. Hence, the contents of specification state elements in the pipeline, excepting the program counter, should correspond exactly to the contents of specification state elements prior to the execution of the instruction about to retire. In the sync-at-retire approach, the program counter for the specification is set to be the program counter for the instruction about to retire, denoted pc_r. Figure 1 also shows the sync-at-retire correctness statement. Specialized for a deterministic implementation and specification, it is:

$$\forall\, q_i.\ \pi_r\ (n_i\ q_i) = n_s\ (\pi_r\ q_i)$$

where π_r is similar to π_f except that it matches the implementation's pc_r with the specification pc.

At first glance, sync-at-retire seems like a dubious correctness statement, for it only compares the completion of a single instruction with one step of the specification. In fact, the above formula simply ensures only that the last stage of a pipeline's execution is correct. The key, however, is that to prove this correspondence, we must assume that the instruction has been correctly executed up to the final stage of the pipeline. This assumption is an invariant that characterizes the reachable state space of the last stage of the implementation. To prove this invariant, we must make assumptions about the reachable state space of the previous stage, and so on. This *back-chaining* continues until the first stage of the pipeline, where invariants must hold under environmental constraints. In this way, a sync-at-retire correctness statement sets up a natural decomposition of the verification problem into proving that each stage executes correctly. The resulting invariants can all be verified automatically and are significantly less complex than using flushing to prove sync-at-fetch. In the next section, we describe in detail how the invariants for sync-at-retire can be systematically constructed using history variables and completion functions.

For implementations that can stall, both of these correctness statements add a case verifying that when no instructions is fetched (or retired), then the externally visible state does not change.

A potential difficulty occurs with both sync-at-retire and sync-at-fetch if architectural state is committed after instruction retirement. For example, some microarchitectures delay committing store operations. In this situation, a structural decomposition with a separate proof about the memory hierarchy can be needed.

3 Synchronization-at-Retirement

In this section, we describe verifying sync-at-retire for deterministic implementations and specifications, and present a systematic process for creating and verifying the necessary invariants. The overall strategy is to prove that the invariants imply sync-at-retire and then that the invariants are indeed invariant.

The experience of the authors, along with results detailed in the literature, indicates that invariant finding is likely the most difficult aspect of this type of verification. We describe a novel invariant-generation process based on the use of history variables and completion functions. Our approach decomposes the proof by pipeline stages; each stage is further decomposed into individual data elements of instructions.

3.1 Parcels and History Variables

For the systematic creation of invariants, we capture the movement of an instruction through the pipeline using *parcels* – information about the instruction at each stage of the pipeline. After instruction decode, every pipeline register contains a parcel with the following fields:

`valid`	`pc`	`pc_next`	`opcode`	`src1`	`data1`	`src2`	`data2`	`dest`	`result`

The `valid` field indicates whether or not the parcel is a bubble. The `pc` and `pc_next` fields contain the program counter (address) for the current instruction and the program counter for the next instruction. The `opcode`, `src1`, `src2`, and `dest` fields are the opcode, and register addresses for the operand sources and instruction result. The `data1`, `data2`, and `result` fields contain the actual operands and result. When used, immediate data is placed in the `data2` field.

In the early stages of the pipeline, many of these fields contain don't cares. For example, the value of `data1` does not matter until after the instruction operands have been read from the register file. As an instruction moves through the pipeline, many parcel fields become history variables. For example, once values have been read, the source addresses are never read again. Our approach only requires history variables for in-flight instructions; we need no record of instructions that have retired.

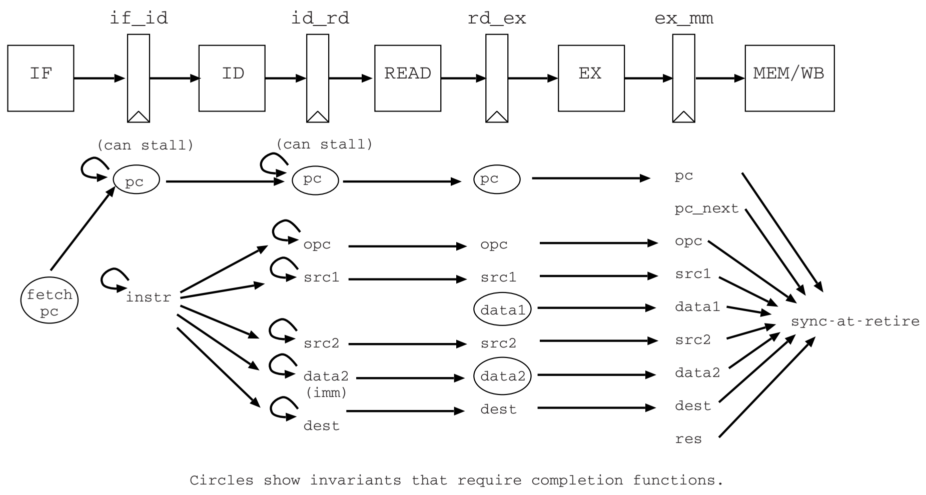

Fig. 2. Example Pipeline, Invariants, and Invariant Dependencies

3.2 Proof Decomposition

We decompose the necessary set of invariants by stage and by parcel element. Each invariant states that a parcel element is correct in a given stage. We prove sync-at-retire by assuming that the parcel contents in the last pipeline register have been computed correctly. Then we chain backward, proving each invariant by relying only on the invariant describing the correctness of the corresponding parcel element in the previous stage.

We explain our approach for generating invariants using the pipeline in Figure 2. This pipeline executes NOPs, ALU instructions, loads, stores, jumps with direct addressing, and branches with offset addressing. We use uninterpreted functions to represent operations on data.

Figure 2 also shows the parcel fields at each stage in the pipeline that have invariants associated with them, and the dependencies between invariants (more on this later). We provide an overview of the invariants, starting with the last stage of the pipeline.

For the sync-at-retire correctness statement to be valid, the parcel in `ex_mm` must have been correctly calculated and the `MEM/WB` (memory/writeback) logic must write correct values to memory and the register file. The table below describes the invariants that characterize a correct `ex_mm` parcel; all have as an antecedent that the parcel is valid. These invariants are sufficient to prove sync-at-retire.

ex_mm_pc_ok	pc matches pc_r
ex_mm_pc_next_ok	pc_next is either ex_mm.pc + 4 or the branch/jump target
ex_mm_opc_ok	opcode matches instruction memory at ex_mm.pc
ex_mm_src1_ok	src1 matches instruction memory at ex_mm.pc
ex_mm_data1_ok	data1 matches ex_mm.src1 in register file
ex_mm_src2_ok	src2 matches instruction memory at ex_mm.pc
ex_mm_data2_ok	data2 matches ex_mm.src1 in register file or immediate data
ex_mm_dest_ok	dest matches instruction memory at ex_mm.pc
ex_mm_res_ok	result matches branch target or result of ALU operation

Moving back, we prove the ex_mm invariants by assuming that the parcel in the previous stage (rd_ex) was computed correctly. Proofs of the invariants are sliced by data element: to prove ex_mm_data1_ok, we need only rd_ex_data1_ok. These verification obligations check that the EX logic is correct. The EX logic computes branch targets and the results of ALU instructions. If the branch is mispredicted, earlier stages will squash parcels in its shadow and reset the fetch program counter.

Because instruction memory never changes, the invariants for opc, src1, src2, and dest are the same in every stage after instruction decode. These invariants are proved by back-chaining one stage at a time until the front of the pipeline is reached. In contrast, pc_next does not have to be correct until it is determined by the EX logic. In stages prior to ex_mm, no invariants are required for pc_next, because it may not be correct. In general, as we move back in the pipeline, fewer invariants about parcel fields are required. The next table describes the invariants necessary for the rd_ex pipeline register, omitting the ones related to instruction memory.

rd_ex_pc_ok	pc matches pc_next of completing ex_mm
rd_ex_data1_ok	data1 matches rd_ex.src1 in register file *after* instruction in ex_mm is completed
rd_ex_data2_ok	original immediate data, or data2 matches rd_ex.src2 in register file *after* instruction in ex_mm is completed

To state some of the invariants about the rd_ex stage, we need to refer to a future specification state, namely the specification state that results from completing the instruction in the ex_mm stage. For example, if the pipeline's bypass logic is correct, the data1 field should be the value in register src1 after the instruction in ex_mm is completed. To calculate the specification state that results from completing the parcels downstream from a stage, we use completion functions [HGS03]. The completion function for a stage describes the result that the instruction currently in that stage will eventually have on the specification state.

A completion function takes an specification state and a parcel and returns the specification state that results from committing this parcel. A completion function is similar to running the specification except that some of the results (such as fetching the operands) have already been computed and are present in the parcel. Figure 3 shows the register file completion function for the ex_mm parcel and illustrates how the invariant rd_data1_ok uses this completion function. Hosabettu *et al.* [HGS03] used completion functions as a decomposition technique for a proof of sync-at-fetch; our use differs significantly as we use completion functions to construct a set of invariants decomposed by stage and parcel elements.

The other invariants are constructed similarly. All invariants are proved inductively; Figure 2 shows the dependencies between invariants. When a stage can stall, the proof of an invariant must assume both the invariant for the parcel element in the previous stage and for the parcel element in the current stage. The circled invariants in Figure 2 use completion functions. We compose the completion function of a given stage with the composition of the completion functions from the downstream stages. Note that in the if_id stage, the instruction has not yet been decoded, so there are no parcel fields.

The most complex invariant for our pipeline is the one about fetch_pc. This is because the pipeline might contain bubbles in any stage. This invariant includes a com-

```
// instruction in ex_mm updates rf of specification state correctly
ex_mm_cf_rf ex_mm (_,rf,mem) =
  if (valid ex_mm)
  then if (is_ld ex_mm)
    // if load op, read data from memory and write to register file
    then write rf (ex_mm.dest) (read mem (ex_mm.data1))
    else (is_alu ex_mm)
      // write computed result for ALU operations
      (write rf (ex_mm.dest) (ex_mm.result))
      // otherwise, no change to register file
      rf
  // otherwise, no change to register file
  else rf

// if instruction in rd_ex is valid, then data1 field matches src1 in rf
rd_ex_data1_ok rd_ex (_,rf,_) =
  (valid rd_ex) ==> (rd_ex.data1 == read rf rd_ex.src1)

// invariant instantiation
rd_ex_data1_ok rd_ex (ex_mm_cf ex_mm (pc_r,rf,mem))
```

Fig. 3. Example Completion Function, Invariant Definition, and Invariant Instantiation

position of completion functions to determine the program counter that results from executing all the instructions currently in the pipeline. These completion functions must determine whether a given instruction in the pipeline will retire, which in turn depends on whether there are any mispredicted branches ahead of this instruction.

We conclude this section with a brief, but formal description of our approach for a three stage pipeline with no stalls. We use the following sets and functions:

P_j	Type of parcel at state j
$\pi_j : Q_i \to P_j$	Project pipeline register j
$c_j : P_j \to Q_s \to Q_s$	Completion function for stage j
$I_j : P_j \to Q_s \to Bool$	Invariant for stage j

Using the definitions $q_s = \pi_r\ q$, $q'_s = \pi_r\ (n_i\ q)$, $p_j = \pi_j\ q$, and $p'_j = \pi_j\ (n_i\ q)$, the proof steps are:

$$\begin{array}{rcl} I_3\ p_3\ q_s & \vdash & n_s\ q_s = q'_s \\ I_2\ p_2\ (c_3\ p_3\ q_s) & \vdash & I_3\ p'_3\ q'_s \\ I_1\ p_1\ (c_2\ p_2\ (c_3\ p_3\ q_s) & \vdash & I_2\ p'_2\ (c_3\ p'_3\ q'_s)) \\ & \vdash & I_1\ p'_1\ (c_2\ p'_2\ (c_3\ p'_3\ q'_s)) \end{array}$$

Note that the conclusion of the first obligation is the sync-at-retire correctness statement. These proof obligations are further decomposed by parcel element as described in Figure 2. The history variables used in this approach do not add to verification complexity because the pipeline never reads history variables.

In summary, we create an invariant for each computed parcel field in each stage of the pipeline. Earlier in the pipeline, fewer fields have been computed, so fewer invariants are needed. Our decomposition checks the correctness of each stage of the pipeline individually, significantly reducing the complexity of the formula to be verified and usually making it easier to isolate an error for debugging. All of the proof obligations require taking only a single-step of the implementation. *No flushing of the implementation pipeline ever occurs with our approach.* The only multi-step computation contained in our approach is the composition of completion functions, a much simpler task than reasoning about flushing the entire pipeline.

4 Equivalence of Correctness Statements

We have proved the equivalence of the sync-at-retire and sync-at-fetch correctness statements for superscalar non-deterministic implementations and deterministic specifications. Our proof establishes that sync-at-retire implies sync-at-fetch, and that sync-at-fetch implies sync-at-retire. Our proof relies on minor conditions describing the relationship between *flush*, *kill*, π, and other functions used in the correctness statements.

Of the two directions, the proof that sync-at-retire implies sync-at-fetch was substantially more difficult. The difficulty arose in discovering conditions that do not require flushing an implementation state, so they are computationally less expensive to check than the sync-at-fetch correctness statement. Because sync-at-fetch relies on flushing, we used conditions that rely on flushing when proving sync-at-fetch implies sync-at-retire. In this section, we focus on the more difficult of the two proofs (sync-at-retire implies sync-at-fetch, Theorem 1), but limit our presentation to scalar implementations for clarity. The details of the other proofs appear in a technical report [ADJ04].

Theorem 1. *Sync-at-retire implies sync-at-fetch*

$$[\forall\, q_r, q_r'.\ N_i\ q_r\ q_r' \Longrightarrow \pi_r\ q_r' = n_s\ (\pi_r\ q_r)]$$
$$\Longrightarrow$$
$$\left[\forall\, q_f, q_f'.\ N_i\ q_f\ q_f' \Longrightarrow \pi_f(\mathit{flush}\ q_f') = n_s\ (\pi_f(\mathit{flush}\ q_f))\right]$$

Each of sync-at-fetch and sync-at-retire compare an implementation step against a specification step. But, for the same implementation step, the two correctness statements choose different specification steps. Sync-at-fetch chooses the specification step that executes the instruction that the implementation fetches, while sync-at-retire chooses the specification step that executes the instruction that the implementation retires. The essence of our proof is to establish a relationship between the implementation step that fetches an instruction and the implementation step that retires the instruction. We introduce the notion of a *serial execution* of the implementation to bridge the gap between the fetching step and retiring step. In a serial-execution step, the implementation starts in a flushed state, takes a single step to fetch an instruction, then flushes the single fetched instruction, to result in a flushed state.

Figure 4 outlines the proof that sync-at-retire implies sync-at-fetch. The rightmost column illustrates the justification for each step. Solid lines denote relations that are known (e.g., the left-hand-side of an implication); dashed lines denote relations on the right-hand-side of an implication; solid circles denote universally quantified states; and hollow circles denote existentially quantified states.

Theorem 1 is of the form $(A \Longrightarrow B) \Longrightarrow (C \Longrightarrow D)$. The proof of such a theorem proceeds by assuming C, then using $A \Longrightarrow B$ and other lemmas to prove D. For Theorem 1, we begin in Step 1 by assuming the antecedent of sync-at-fetch. In Step 2, we use Lemma 1 to prove that the implementation step from q_r to q_r', which fetches the instruction i, corresponds to a serial-execution step from q_e to q_e'. In Step 3, we use Lemma 3 to prove that the serial-execution step from q_e to q_e' corresponds to the step from q_r to q_r', which retires the instruction i. The correspondence is achieved by killing all in-flight instructions (denoted as *kill* in the diagram). In Step 4, we use sync-at-retire to prove that q_e' is equivalent to the specification state q_s'. Step 5 concludes the proof by showing that sync-at-fetch holds between $q_f \rightarrow q_f'$ and $q_s \rightarrow q_s'$.

Lemma 1 says that each step of fetching an instruction corresponds to a serial-execution step. The lemma could be proved easily by flushing q_f and q_f', but in the proof of sync-at-retire implies sync-at-fetch, we wish to avoid flushing. The purpose of Lemma 1 is to prove that the two paths of the commuting diagram, $q_f \rightarrow q_e \rightarrow q_e'$ and $q_f \rightarrow q_f' \rightarrow q_e'$, both result in the same state q_e'. The proof proceeds in two phases: first, we prove that both paths retire the same number of instructions; then we prove that any pair of paths that start from the same state and retire the same number of instructions will result in the same state. The proof relies on Lemma 2 and Condition 1.

Lemma 2 says that each step that retires an instruction corresponds to a serial-execution step. The proof of this lemma involves applying sync-at-retire correctness to the state one step prior to the flushed state q_e'. Applying the *kill* function to q_e and all states prior to q_e' results in the same implementation step because no instructions are retired in these steps.

Lemma 2. *Retire implies serial execution*

$$
\begin{array}{l}
\forall\, q_r, q_r' .\\
\quad \left[\wedge \begin{array}{l} N_i\ q_r\ q_r' \\ \textit{numRetire}\ q_r\ q_r' = 1 \end{array} \right]\\
\quad \Longrightarrow\\
\quad \left[\begin{array}{l} \exists\, q_e'' . \\ \quad \wedge \begin{array}{l} N_i\ (\textit{kill}\ q_r)\ q_e'' \\ \pi(\textit{flush}\ q_e'') = \pi(\textit{kill}\ q_r') \end{array} \end{array} \right]
\end{array}
$$

$$\begin{tikzcd}
q_e \arrow[r, dashed, leftrightarrow, "N_i"] & q_e'' \arrow[rr, dotted, "\textit{flush}"] & & q_e' \\
& & q_r \arrow[ull, "\textit{kill}"] \arrow[r, leftrightarrow, "\textit{retire}\ i"'] & q_r' \arrow[u, "\textit{kill}"']
\end{tikzcd}$$

Condition 1 says that from a flushed state, a fetched instruction will never be killed (i.e., its execution is not speculative and it should eventually retire). Such a condition may not be necessary in practice for the correctness of a pipeline, but we require it to prove that sync-at-retire implies sync-at-fetch. The condition can be verified by proving invariants about *numRetire* and *numWillRetire*. In Section 5, we demonstrate that checking Condition 1 is computationally less expensive than using sync-at-fetch.

Condition 1. *From a flushed state, a fetched instruction will always retire*

$$
\begin{array}{l}
\forall\, q, q' .\\
\quad \left[\begin{array}{l} \textit{isFlushed}\ q \\ \wedge\ N_i\ q\ q' \\ \wedge\ \textit{numFetch}\ q\ q' = 1 \end{array} \right] \Longrightarrow \left[\vee \begin{array}{l} \textit{numRetire}\ q\ q' = 1 \\ \textit{numWillRetire}\ q' = 1 \end{array} \right]
\end{array}
$$

Lemma 3 is used in Step 3 of Figure 4. The lemma says that for each serial-execution step, there exists an implementation step that retires the instruction and is related to the serial-execution step via the *kill* function. This lemma is the opposite of Lemma 2, which says that each retiring step corresponds to a serial-execution step. To prove Lemma 3, note that a witness for the implementation state q_r can be computed by starting in the flushed state q_e, taking a step to fetch an instruction, and then stepping the pipeline until the instruction is about to retire.

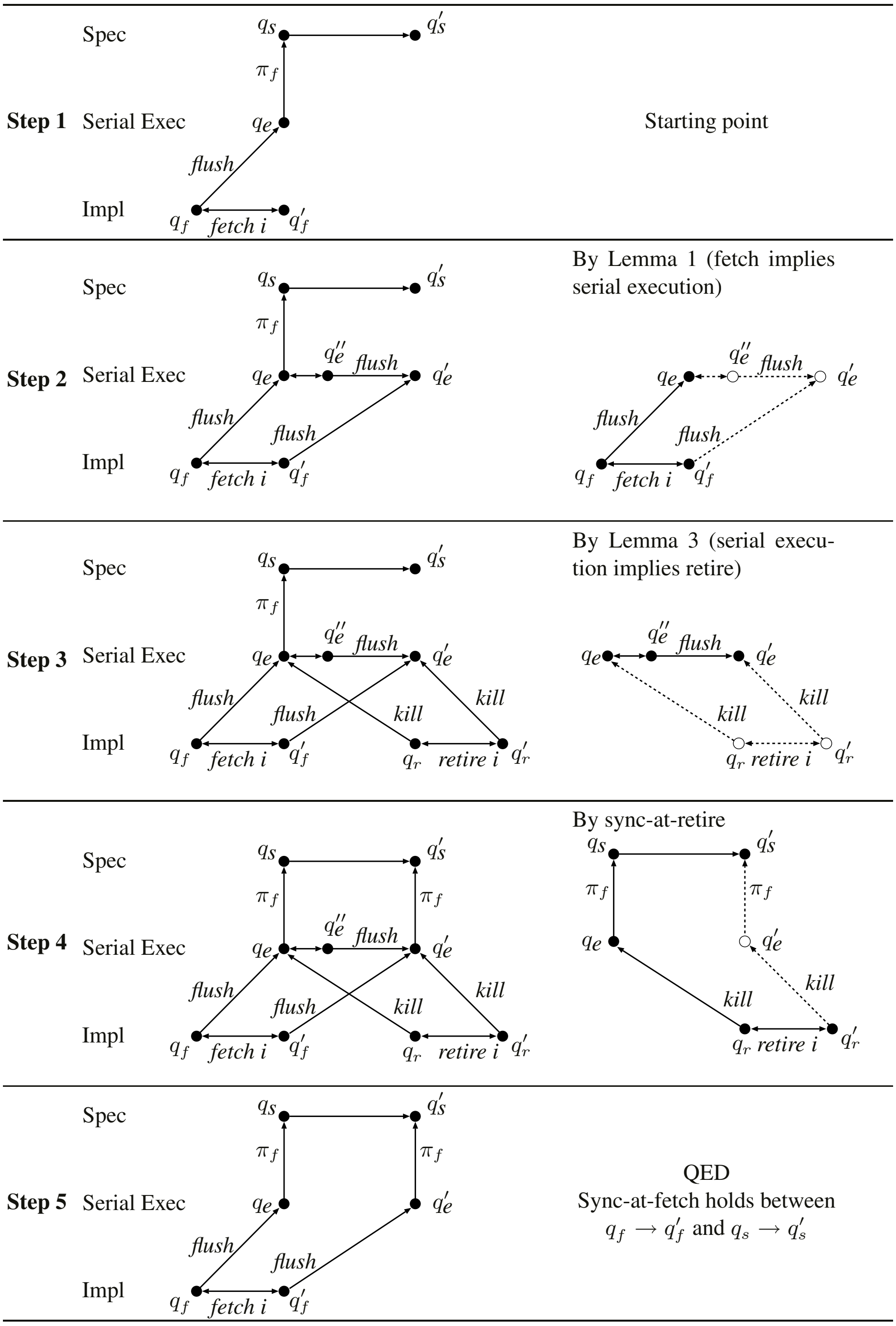
Step 1
Spec
Serial Exec
Impl
q_s
q_s'
π_f
q_e
flush
q_f
fetch i
q_f'
Starting point
Step 2
By Lemma 1 (fetch implies serial execution)
q_e''
q_e'
Step 3
By Lemma 3 (serial execution implies retire)
kill
q_r
retire i
q_r'
Step 4
By sync-at-retire
Step 5
QED
Sync-at-fetch holds between $q_f \longrightarrow q_f'$ and $q_s \longrightarrow q_s'$

Fig. 4. Overview of proof that sync-at-retire implies sync-at-fetch

Lemma 3. *Serialized implementation implies retire*

$$\forall\, q_e, q_e''.\quad \begin{bmatrix} \wedge\; N_i\; q_e\; q_e'' \\ \quad isFlushed\; q_e \end{bmatrix} \Longrightarrow \begin{bmatrix} \exists\, q_r, q_r'. \\ \quad \wedge\; N_i\; q_r\; q_r' \\ \quad \wedge\; q_e = kill\; q_r \\ \quad\quad \pi(flush\; q_e'') = \pi(kill\; q_r') \end{bmatrix}$$

The proof that sync-at-fetch implies sync-at-retire proceeds symmetrically to Figure 4: we show that every step that retires an instruction corresponds to a step in serial execution, and that every step in serial execution corresponds to a step that fetches that same instruction. In the superscalar proofs [ADJ04] multiple instructions might be fetched or retired in a single clock cycle. This generalization means that a single fetching or retiring step corresponds to multiple steps of serial execution and multiple steps of the specification.

5 Results

We used SVC [BDL96] to compare the verification complexity and time of three example pipelines to demonstrate that sync-at-retire is computationally more efficient than sync-at-fetch. Complete proof scripts are available at [Mic]. Table 2a shows the case splits, expressions generated and run-times for the three pipelines, each of which is an extension of the previous, and has the stages shown in Figure 2 and the following features:

- **pipe1:** executes loads and stores; no branches; stalls for dependencies on load instructions; includes icache misses and bubble squashing.
- **pipe2:** adds branch and jump instructions with branch prediction; the branch target is resolved in the `READ` stage meaning two instructions can be in the shadow of a mispredicted branch.
- **pipe3:** moves branch target resolution to the `EX` stage, meaning that three instructions can be in the shadow of a mispredicted branch.

The row labeled “total time” is the sum of the time for checking sync-at-retire and all invariants needed for sync-at-retire. No invariants were required by sync-at-fetch for these simple pipelines. Both methods require environmental assumptions that opcodes are distinct.

We expect that this result is independent of proof engine – similar results should be found using a different validity checker such as UCLID [LSB02] or CVC Lite [BB04]. Sync-at-retire does involve the manual work of creating the invariants, however these are systematically created and not iteratively discovered based on counter-examples. While in theory our approach requires more manual work (to construct the invariants), we found in practice that debugging the pipeline was often significantly easier because of the systematic nature of the invariant construction, which isolate bugs to a single pipeline stage.

To guarantee that proving sync-at-retire is equivalent to proving sync-at-fetch, we also checked the proof obligation required by the proof outlined in Section 4. Table 2b

Table 2. Run-time results

Pipeline	pipe1	pipe2	pipe3
Sync-at-retire			
Case splits	44	78	78
Exprs. generated	252	530	532
Time (s)	<1	<1	<1
Largest invariant:			
Case splits	4.5k	374k	527k
Exprs. generated	15.0k	7.72M	9.28M
Time (s)	<1	44	54
Total time (s)	7	115	141
Sync-at-fetch			
Case splits	10.4k	615k	852k
Exprs. generated	1.48M	72.6M	77.6M
Time (s)	9	674	651

(a) sync-at-retire vs. sync-at-fetch

Pipe3	Condition 1
Case splits	95.0k
Exprs. generated	624k
Time (s)	4.5

(b) sync-at-retire implies sync-at-fetch

shows the complexity of checking Condition 1 for pipe3. It is considerably less expensive than checking sync-at-fetch. Note again that sync-at-retire is an acceptable correctness criteria by itself, and this condition is only necessary to guarantee that sync-at-fetch is also satisfied.

6 Related Work

In earlier work, we surveyed many of the different approaches to safety verification of microprocessors, and compared their correctness statements [ACDJ03]. In the body of work that we surveyed, most verification efforts used sync-at-fetch. In this section, we limit the discussion to prior work that has used sync-at-retire.

Fox and Harman [FH96,FH03] suggested the idea of relating a pipeline to a specification at the time of retirement with *retiming functions*. A retiming function maps an implementation state to the specification state that corresponds to the retiring instruction. There is a separate data abstraction function. They verify a single-step commuting diagram, but there is no discussion of invariants. They use term rewriting systems for verification.

Arons and Pnueli [PA98,AP00] have used synchronize-at-retire with theorem proving. They use a program counter similar to ours, where the externally-visible program counter is the address of the next instruction to be retired. They do not discuss invariant generation other than to mention that it was the most difficult part of the proof.

Arvind and Shen [AS99] reported using term rewriting systems (TRS) in the verification of pipelined processors. In their paper, they discuss applying rewrites until the system reaches a *drained* (flushed) state, an approach similar to flushing. They also mention the possibility of "rolling back" pipeline execution by killing all partially-completed instructions, but do not discuss it in enough detail for a meaningful comparison.

Manolios [MS04] uses a *commitment* approach that maintains history variables with parcels. Unlike our application, Manolios uses history variables that keep track of the state of the pipeline before a given operation executed. He uses the history variables to "rewind" the pipeline to revert to a previous state. He reports that using commitment to verify safety properties is computationally more complex than using flushing. However, with commitment, the incremental cost of verifying liveness is less than with flushing.

Arons and Pnueli [AP98], Jhala and McMillan [JM01], and Lahiri and Bryant [LSB02] use a set of invariants (refinement maps), rather than a simulation relation, as their correctness statement. The typical approach is to augment the implementation with history variables. When an instruction is fetched, issued, or dispatched, the specification is executed to compute the correct values for any architectural state variables that the instruction will write. When the instruction writes to architectural state, the actual values written are compared to those in the history variables. Each architectural state variable has its own refinement map, and so there is no unique synchronization point in the verification. The lack of a synchronization point usually prevents the direct comparison of an implementation state with the specification state. In some cases, it is possible to do a direct comparison when the implementation is in a flushed state.

7 Conclusion

In this paper, we demonstrated that verifying a correctness formulation based on sync-at-retire can have significant advantages over a formulation based on sync-at-fetch because no verification step requires more than a single step of the implementation. Further, we proved that the two formulations are equivalent in the bugs that they will detect. We did not attempt to show that sync-at-retire can verify a more complex pipeline than any previously verified with sync-at-fetch, but instead focused on the comparison of the approaches. We are interested in comparing our work with Hosabettu's approach to decomposing the sync-at-fetch correctness statement with completion functions [HGS03], but must first map his approach to an automated verification environment.

For processor pipelines, a significant advantage of sync-at-retire as opposed to sync-at-fetch is dealing with precise interrupts, where instructions ahead of the interrupt in the pipeline may be killed. The sync-at-fetch approach would require a complex "informed flushing" function to decide how many instructions should be flushed before comparing with the specification state. In contrast, sync-at-retire requires no such function, and any reasonable interrupt handling scheme can be encoded in the invariants. We are currently studying and quantifying this observation.

We plan to study sync-at-retire with other kinds of pipelines: very deep pipelines used in digital signal processors and parallel pipelines, such as those used in graphics engines. We are also interested in exploring the applicability of our approach to superscalar pipelines with out-of-order instruction execution and in-order retirement.

Acknowledgments

We thank the anonymous reviewers for their insightful comments. We thank Jun Sawada for an idea that helped us discover proofs for two out of the conditions in the paper. The first and second authors are supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC). Aagaard is supported in part by Intel Corporation, and the Semiconductor Research Corporation. The third author thanks Intel Corporation's Strategic CAD Labs for providing a work environment that supports external research collaboration.

References

[ACDJ03] M. D. Aagaard, B. Cook, N. A. Day, and R. B. Jones. A framework for superscalar microprocessor correctness statements. *Software Tools for Technology Transfer*, 4(3):298–312, 2003.

[ADJ04] M. D. Aagaard, N. A. Day, and R. B. Jones. Equivalence of sync-at-fetch and sync-at-retire correctness for pipeline circuits. Technical Report CS2004-31, University of Waterloo, School of Computer Science, September 2004.

[AP98] T. Arons and A. Pnueli. Verifying Tomasulo's algorithm by refinement. Technical report, Dept. of Computer Science, Weizmann Institute, October 1998.

[AP00] T. Arons and A. Pnueli. A comparison of two verification methods for speculative instruction execution with exceptions. In *TACAS*, vol 1785 of *LNCS*, pp 487–502. Springer, 2000.

[AS99] Arvind and X. Shen. Using term rewriting systems to design and verify processors. *IEEE Micro*, 19(3):36–46, 1999.

[BB04] C. Barrett and S. Berezin. CVC Lite: A new implementation of the cooperating validity checker. In *CAV*, vol 3114 of *LNCS*, pp 515–518. Springer, 2004.

[BD94] J. Burch and D. Dill. Automatic verification of pipelined microprocessor control. In D. L. Dill, editor, *CAV*, vol 818 of *LNCS*, pp 68–80. Springer, 1994.

[BDL96] C. Barrett, D. Dill, and J. Levitt. Validity checking for combinations of theories with equality. In *FMCAD*, vol 1166 of *LNCS*, pp 187–201. Springer, 1996.

[FH96] A. Fox and N. Harman. Algebraic models of correctness for microprocessors. In *FMCAD*, vol 1166 of *LNCS*, pp 346–361. Springer, 1996.

[FH03] A. Fox and N. Harman. Algebraic models of correctness for abstract pipelines. *The Journal of Algebraic and Logic Programming*, 57:71–107, 2003.

[HGS03] R. Hosabettu, G. Gopalakrishnan, and M. K. Srivas. Formal verification of a complex pipelined processor. *Formal Methods in System Design*, 23(2):171–213, September 2003.

[JM01] R. Jhala and K. L. McMillan. Microarchitecture verification by compositional model checking. In *CAV*, vol 2102 of *LNCS*, pp 396–410. Springer, 2001.

[LSB02] S. K. Lahiri, S. A. Seshia, and R. E. Bryant. Modeling and verification of out-of-order microprocessors in UCLID. In *FMCAD*, vol 2517 of *LNCS*, pp 142–158. Springer, 2002.

[Mic] `http://www.cs.uwaterloo.ca/˜nday/microbox`.

[Mil71] R. Milner. An algebraic definition of simulation between programs. In *Joint Conf. on AI*, pp 481–489. British Computer Society, 1971.

[MS04] P. Manolios and S. K. Srinivasan. Automatic verification of safety and liveness for Xscale-like processor models using web refinements. In *DATE*, pp 168–175, 2004.

[PA98] A. Pnueli and T. Arons. Verification of data-insensitive circuits: An in-order-retirement case study. In *FMCAD*, vol 1522 of *LNCS*, pp 351–368. Springer, 1998.