Formal Verification of Out-of-Order Execution Using Incremental Flushing

Jens U. Skakkebæk[1], Robert B. Jones[1,2], and David L. Dill[1]

[1] Computer Systems Laboratory, Stanford University, Stanford, CA 94305, USA
{jus,dill}@cs.stanford.edu
[2] Strategic CAD Labs, Intel, JFT-104, 2111 NE 25th Ave., Hillsboro, OR 97124, USA
rjones@ichips.intel.com

Abstract. We present a two-part approach for verifying out-of-order execution. First, the complexity of out-of-order issue and scheduling is handled by creating an in-order abstraction of the out-of-order execution core. Second, *incremental flushing* addresses the complexity difficulties encountered by automated abstraction functions on very deep pipelines. We illustrate the techniques on a model of a simple out-of-order processor core.

1 Introduction

Formal verification of microprocessor designs using theorem proving aims at proving that a processor model behaves as defined by an *instruction-set architecture* (ISA). The ISA captures the programmer-level view of the machine. This approach requires an *abstraction function* that relates the state of the processor model with the corresponding state of the ISA. Finding this abstraction function manually for pipelined designs is tedious and time consuming. In response, Burch and Dill devised an approach that automatically generates the abstraction function by *flushing* the implementation state [3]. The technique has been extended to dual-issue and super-scalar architectures [7, 2, 15].

While formal verification techniques exist for pipelined and super-scalar architectures, experience verifying out-of-order architectures is minimal. The distinct features of out-of-order architectures challenge existing verification approaches. First, the extended instruction parallelism in out-of-order architectures results in many complex interactions between executing instructions. This greater complexity makes it very difficult to devise an abstraction function. Second, large ($\geq$ 40 element) buffers are used to record and maintain the program order of instructions. Burch and Dill's automated pipeline flushing approach does not work for out-of-order architectures in practice because the number of cycles required to empty the buffer completely is so large. The logical formulas are too complex to manipulate in proofs and often too complex even to construct.

We present a two-part approach that deals with the out-of-order scheduling logic and the in-order buffering mechanisms separately. First, the implementation is modified to derive an in-order abstraction. These modifications bypass the out-of-order logic and result in instructions executing in order. By exploiting domain-specific knowledge, we are able to establish a functional equivalence relation between the out-of-order implementation and the abstraction. The second step of our technique shows that the in-order

abstraction is functionally equivalent to the ISA. This is accomplished via a technique introduced in this paper that we call *incremental flushing*, based on the Burch-Dill automatic flushing approach and the self-consistency technique of Jones *et al.* [8]. Incremental flushing reduces the verification complexity associated with flushing lengthy pipelines. This technique is also applicable to verification of other deeply-pipelined hardware designs, not just out-of-order microarchitectures.

We have created a simple model of an out-of-order execution core that we use to illustrate our approach. Although our example is not representative of industrial-scale designs, it captures essential features of out-of-order architectures: large queuing buffers, resource allocation within the buffers, and data-path scheduling of execution resources. However, using the techniques presented here, we were able to verify it using the Stanford Validity Checker (SVC) [1]. In particular, we have verified its correctness for any (reasonable) scheduling algorithm.

2 Related Work

Sawada and Hunt's theorem-proving approach uses a table of history variables, called a *micro-architectural execution trace table* (MAETT) [14, 13]. The MAETT is an intermediate abstraction that contains selected parts of the implementation as well as extra history variables and variables holding abstracted values. It includes the ISA state and the ISA transition function. A predicate relating the implementation and MAETT is found by manual inspection and proven by induction to be an invariant on the execution of the implementation. In our approach, the intermediate abstraction does not include the ISA state, but is closer to the implementation in abstraction level. This minimizes the manual work needed to find the relation between the implementation and abstraction. We then use an incremental flushing technique to automatically generate the abstraction function, significantly reducing the manual work required to relate the intermediate abstraction to the ISA.

Damm and Pnueli generalize an ISA specification to a non-deterministic abstraction [4]. It is then verified that the implementation satisfies the abstraction by manually establishing and proving the appropriate invariants. They have applied their technique to the Tomasulo algorithm [5], which has out-of-order instruction completion. In contrast, our out-of-order model features in-order retirement. In our approach, the intermediate abstraction executes instructions in-order. Damm and Pnueli's abstraction represents all possible instruction sequences which observe dataflow dependencies. Applying their method to architectures with in-order retirement would require manual proof by induction that the intermediate abstraction satisfies the ISA. We automate this proof by incremental flushing.

Henzinger *et al.* use Tomasulo's algorithm to illustrate a method for manually decomposing the proof obligation [6]. They provide abstract modules for parts of the implementation. These modules correspond to implementation internal transactions. Similar to our approach, the abstractions are invariants on the implementation and are extended with auxiliary variables. Again, our approach automates part of the abstraction process.

McMillan model checks the Tomasulo algorithm by manually decomposing the proof into smaller correctness proofs of the internal transactions that together form

one step of execution [11]. Furthermore, he uses symmetry reduction technique to extend the proof to a large number of execution units. Our proofs are also decomposed into properties of internal transactions. In contrast to an automated model checking approach, our theorem-proving based method is able to handle internal buffers of arbitrary size.

Incremental flushing is related to the distributed systems work of Katz [10]. His formalization deals with atomic, concurrent transactions which can be reordered into a more convenient form for formal analysis—without affecting the soundness of the final result. However, the framework of distributed transactions cannot be directly applied to verification microprocessor architectures where the control logic dictates the sequencing of internal transactions.

3 Preliminaries

The desired behavior of a processor is defined by an *instruction-set architecture* (ISA). The ISA represents the programmer-level view of the machine where instructions execute sequentially. The ISA for our example is shown in Figure 1a. The simple state

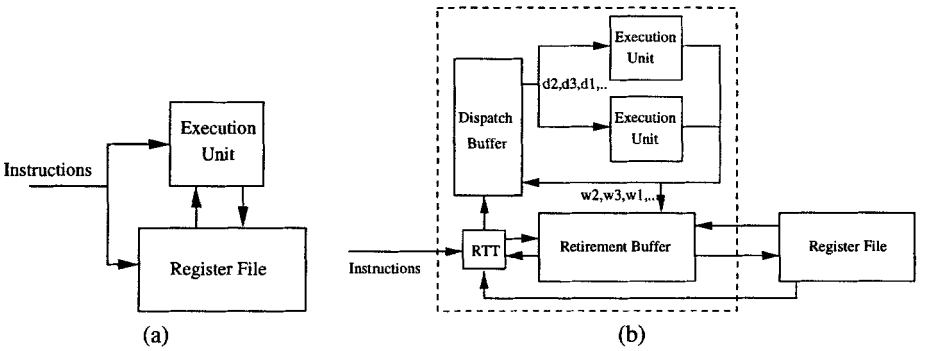

Fig. 1. (a) The simple ISA model. (b) Instruction flow in our out-of-order execution core IMPL.

consists of a register file (RF), while the next-state function is computed with an execution unit (EU) that can execute any instruction. The ISA also accepts a `bubble` input that leaves the state unchanged. Note that our ISA model does not include a program counter or memory state—as these are also omitted from our simplified out-of-order model.

Modern processors do not implement the ISA in this manner, because the performance would be abysmal. In out-of-order architectures, instructions are fetched, decoded, and sent to the execution core in program order. Internally, however, the core executes instructions out-of-order, as allowed by data dependencies. This allows independent instructions to execute concurrently. Finally, instruction results are written back to architecturally-visible state (the register file) in the order they were issued.

Consider our example out-of-order execution core (IMPL) shown in Figure 1b. The architectural register file (RF) contains the current state of the ISA-defined architectural

registers. When an instruction is *issued*, new entries are allocated in both the dispatch and retirement buffers, and the register translation table (RTT) entry for the logical register corresponding to the instruction destination is updated. The RTT is used to locate the instruction's source data. Instructions are *dispatched*, possibly out-of-order, from the dispatch buffer (DB) to individual execution units when their operands are ready and an execution unit is available. When an instruction finishes execution, the result is *written back* to the retirement buffer (RB). This data is also bypassed into the DB for instructions awaiting that particular result. Finally, the RB logic must ensure that instruction results are *retired* (committed to architectural state) in the original program order. When an RB entry is retired, the RTT is informed so that the logical register entry corresponding to the instruction's destination can be updated if necessary. IMPL also accepts a special `bubble` flushing input in place of an instruction. Intuitively, a `bubble` is similar to a NOP instruction but does not affect any state or consume any resources after being issued.

We have made significant simplifying assumptions in our processor model: instructions have only one source operand, and only one issue and one retire can occur each cycle. Our model is out-of-order because the execution units have variable latency. We also omit a "front-end" with fetch, decode, and branch prediction logic. Omitting these features allowed our efforts to focus on the features which make the out-of-order verification problem difficult: the out-of-order execution and the large effective depth of the pipeline. The SVC verification reported in this paper used a model with unbounded buffers.

4 The Approach

The goal of our verification approach is to prove that the out-of-order implementation IMPL (as described by an HDL model) satisfies the ISA model. We define δ_i to be the implementation next-state function, which takes a state q_i and an input instruction i and returns a new state q_i', i.e., $q_i' = \delta_i(q_i, i)$. We extend δ_i in the obvious way to operate over input sequences $w = i_0 \ldots i_n$. We define δ_s similarly for ISA.

Let σ be a *size* function that returns the number of currently executing instructions, i.e., those that have been issued but not retired. We require that $\sigma(q_i^\circ) = 0$ for an initial implementation state q_i°. We define an instruction sequence w to be *completed* iff $\sigma(\delta_i(q_i^\circ, w)) = 0$, i.e., all instructions have been retired after executing w. We use the projection function $\pi_{\text{RF}}(q_i)$ to denote the register file contents in state q_i. For clarity in presentation, we define $q_{i1} \stackrel{\text{RF}}{=} q_{i2}$ to be $\pi_{\text{RF}}(q_{i1}) = \pi_{\text{RF}}(q_{i2})$, and we will sometimes use $\stackrel{\text{RF}}{=}$ when the projection π_{RF} is redundant on one side of the equality.

The overall correctness property for IMPL with respect to ISA is expressed as:

Correctness *For every completed instruction sequence w and initial state q_i°,*

$$\delta_i(q_i^\circ, w) \stackrel{\text{RF}}{=} \delta_s(\pi_{\text{RF}}(q_i^\circ), w).$$

That is, the architecturally visible state in IMPL and ISA is identical after executing any instruction sequence that retires all outstanding instructions in the implementation. This is the same commuting property used by several approaches, including [3]. Note that because our model is only an execution core, we are only checking the correctness

of the register file. A (future) verification of a more complete processor model could check the program counter and memory.

We verify the correctness property by dealing with the out-of-order and in-order parts of IMPL separately. First, we derive an in-order intermediate abstraction (ABS) from IMPL. We then establish an equivalence relation between ABS and IMPL. In the second step, we demonstrate functional equivalence between ABS and ISA. By transitivity of equality of the final register file values, this establishes functional equivalences between IMPL and ISA.

5 First Step: Functional Equivalence of IMPL and ABS

ABS is derived directly from IMPL by removing the "out-of-orderness" while preserving the in-order buffering mechanism (Figure 2). In ABS, the DB has been removed: instructions are executed immediately upon issue. However, the results are queued and not written to architectural state until later. In the ABS model for this paper, instructions are issued, executed, and written into an annotated RB in one clock. The write-only annotated state in the RB contains some of the information lost with the DB removal and aids in finding invariants. ABS accepts the same `bubble` input as IMPL. We add an extra input to ABS called the *retirement flag* that signals when to retire the oldest instruction. ABS thus has more possible behaviors than IMPL: while instruction results are computed immediately in ABS, they may be buffered indefinitely in the annotated RB before being committed to architectural state.

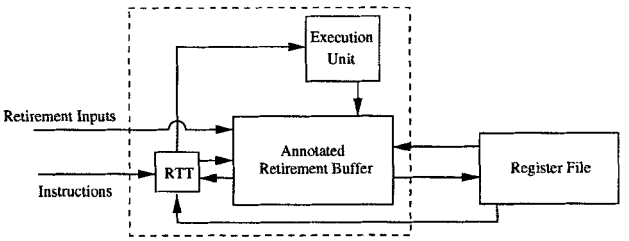

Fig. 2. Instruction flow in the intermediate abstraction.

We must prove that IMPL is a *refinement* of ABS. We define δ_a to be the ABS next-state function, which takes an initial state q_a and a pair consisting of an input instruction i and a Boolean-valued retirement input r, and returns a new state q'_a, i.e., $q'_a = \delta_a(q_a, \langle i, r \rangle)$. The retirement input r indicates in each step of execution whether or not to retire a result. A retirement input r is *allowed* by a state q_a and input i iff r never tells ABS to retire an instruction when one is not waiting. Note that it is allowable for r to *not* retire a waiting instruction. We extend the definition of δ_a to sequences of instruction inputs w and retirement inputs $w_r = r_0 \ldots r_n$ such that $q'_a = \delta_a(q_a, \langle w, w_r \rangle)$[1].

We define states q_i of IMPL and q_a of ABS to be *consistent* when $q_i \overset{RF}{=} q_a$. We must demonstrate that:

[1] The pair of sequences $\langle w, w_r \rangle$ is easily derived from the corresponding sequence of pairs $\langle i_0, r_0 \rangle, \ldots, \langle i_n, r_n \rangle$.

Impl-ABS Refinement *For every completed instruction sequence w and every pair of consistent initial states q_i°, q_a°, there exists a sequence of retirement inputs w_r allowed by q_a° and w such that*

$$\delta_i(q_i^\circ, w) \stackrel{\text{RF}}{=} \delta_a(q_a^\circ, \langle w, w_r \rangle).$$

We prove that IMPL is a refinement of ABS by induction: we show that for each step that IMPL makes, there exists an ABS step such that the register files are identical. Forcing ABS to retire instructions in lock step with IMPL is straightforward. ABS retirement inputs are generated from an oracle which observes whether or not the IMPL is retiring an instruction and instructs ABS to do the same thing. We establish $q_i \stackrel{\text{RF}}{=} q_a$ by proving a stronger property. We derive a relation $\mathcal{R}$ between IMPL and ABS states such that: $\mathcal{R}(q_i, q_a) \Rightarrow (q_i \stackrel{\text{RF}}{=} q_a)$. We demonstrate that $\mathcal{R}$ is a *simulation relation* [9]:

Proof Obligation 1 (IMPL-ABS Equivalence)

1. *(Base Case) For every initial implementation state q_i°, there exists an initial ABS state q_a°, such that:*

$$\mathcal{R}(q_i^\circ, q_a^\circ).$$

2. *(Induction Step) For every instruction i, for every pair of consistent initial states q_i°,q_a°, and for every instruction sequence w and retirement sequence w_r with resulting states $q_i = \delta_i(q_i^\circ, w)$, $q_a = \delta_a(q_a^\circ, \langle w, w_r \rangle)$, there exists a retirement input r such that*

$$\mathcal{R}(q_i, q_a) \Rightarrow \mathcal{R}(\delta_i(q_i, i), \delta_a(q_a, \langle i, r \rangle)).$$

Deriving $\mathcal{R}$ is non-trivial. One way to construct $\mathcal{R}$ is to mechanically derive the weakest invariant which implies $q_i \stackrel{\text{RF}}{=} q_a$. Of course, this technique blows up when applied directly to a complex circuit.

The relation $\mathcal{R}$ is formed as a conjunction of the IMPL reachability invariant, the ABS reachability invariant, and assertions relating the IMPL state with the ABS state. The difficulties associated with deriving invariants are ubiquitous. We used an *ad hoc* collection of domain-specific techniques we found to be quite effective. The process of deriving and proving the reachable-state invariant for IMPL was simplified by recognizing that the out-of-order mechanism in a given cycle consists of a number of *transactions*—each of which operate on only part of IMPL state. In IMPL, these are *issue*, *dispatch*, *writeback*, and *retire*. The ABS reachability invariant is easily derived from the IMPL reachability invariant, because ABS is essentially a simple IMPL. Some IMPL state is not present in ABS, and other IMPL state has been renamed and is now part of the annotated RB.

We added *link assertions* which relate partially executed instructions in the DB and RB of IMPL to their counterparts in the annotated RB of ABS. The link assertions ensure that the partially executed instructions in the implementation always have the correct value or the information needed (pointers or data) to eventually compute the correct value. Run times and memory usage for proving the proof obligations on our example are reported in Section 7.

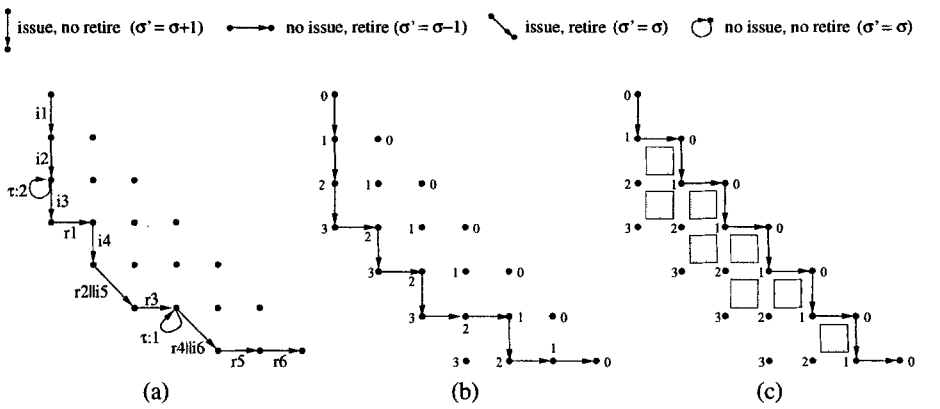

Fig. 3. (a) A *Max-n* execution ε_n. (b) An equivalent non-diagonal execution $\varepsilon_{\hat{n}}$. (c) An equivalent *Max*-1 execution ε_1. Labels in and rn denote the issue and retirement of instruction number n. The label $rn||in$ denotes simultaneous issue and retire. $\tau : n$ is a shorthand for n cycles where in each cycle, `bubbles` are issued and nothing is retired. The numbers indicate the sizes of each state. The squares indicate the distance between $\varepsilon_{\hat{n}}$ and ε_1.

6 Second Step: Functional Equivalence of ABS and ISA

In this section, we introduce incremental flushing, and use it to prove that ABS satisfies ISA. Formally, we desire to establish that:

ABS-ISA Equivalence *For every completed instruction sequence w, initial ABS state q_a°, and sequence of retirement inputs w_r allowed by w and q_a°:*

$$\delta_a(q_a^\circ, \langle w, w_r \rangle) \stackrel{\text{RF}}{=} \delta_s(\pi_{\text{RF}}(q_a^\circ), w).$$

ABS contains an annotated RB that queues instruction results before they are committed to architectural state. Recall that the Burch-Dill abstraction function *flushes* an implementation (by inserting `bubbles`) for the number of clock cycles necessary to completely expose the internal state. In the case of a simple five-stage pipeline, only five steps are required to complete the partially executed instructions. Following this approach with our model would compare a potentially full annotated RB with the ISA model. The Burch-Dill flushing technique would unroll ABS to the depth of the annotated RB, resulting in a logical expression too large for the decision procedure to check.

Our *incremental flushing* approach overcomes this unmanageable complexity. Instead of flushing the entire pipeline directly, a set of smaller, inductive flushing steps is performed. Taken together, these proof obligations imply the monolithic flushing operation. To illustrate, consider the graphical presentation of three different *executions* of ABS in Figure 3. We define the *execution* of a system as the sequence of states that the system passes through when executing a given pair of input sequences $\langle w, w_r \rangle$. For instance, the execution shown in Figure 3a is a result of executing the input sequence:

$\langle i_1, \text{F} \rangle, \langle i_2, \text{F} \rangle, \langle \texttt{bubble}, \text{F} \rangle, \langle i_3, \text{F} \rangle, \langle \texttt{bubble}, \text{T} \rangle, \langle i_4, \text{F} \rangle,$
$\langle i_5, \text{T} \rangle, \langle \texttt{bubble}, \text{T} \rangle, \langle \texttt{bubble}, \text{F} \rangle, \langle i_6, \text{T} \rangle, \langle \texttt{bubble}, \text{T} \rangle, \langle \texttt{bubble}, \text{T} \rangle$

Apart from self-loops, edges are only traversed when instructions are issued or retired.

We use $\varepsilon(q_a, \langle w, w_r \rangle)$ to denote the execution (sequence of states) resulting from the application of δ_a to q_a and $\langle w, w_r \rangle$. We define $last(\varepsilon(q_a, \langle w, w_r \rangle))$ as the last state of the execution. Note that by definition:

$$last(\varepsilon(q_a, \langle w, w_r \rangle)) = \delta_a(q_a, \langle w, w_r \rangle)).$$

Each state in an execution is associated with the number of active instructions—defined earlier as the *size* function σ. This is illustrated in Figure 3c. We call an execution where for all states $\sigma \leq n$ a *Max-n* execution (denoted ε_n). Accordingly, completely serialized executions with at most one outstanding element are *Max*-1 executions (denoted ε_1).

Our verification of ABS-ISA equivalence proceeds in two steps. First, we establish that:

Incremental Flushing *For every initial state* q_a° *and* Max-n *execution* $\varepsilon_n(q_a, \langle w, w_r \rangle)$, *there exists* $\langle w^1, w_r^1 \rangle$ *(derived from* w, w_r *by reordering issues and retires) and a corresponding* Max-1 *execution* $\varepsilon_1(q_a, \langle w^1, w_r^1 \rangle)$ *such that:*

$$last(\varepsilon_n(q_a^\circ, \langle w, w_r \rangle)) = last(\varepsilon_1(q_a^\circ, \langle w^1, w_r^1 \rangle)).$$

A *Max*-1 execution is derived from a *Max-n* execution by reordering the issues and retires. This notion is based on the concept of *self-consistency*: execution results should be equivalent for certain classes of inputs [8]. The final results of *Max-n* and *Max*-1 executions will be identical if we can prove inductively that reordering issue and retires for distinct instructions does not change the resulting state. Section 6.1 details the proof obligations for this step.

The second ABS-ISA verification step shows that all *Max*-1 executions produce the same result as the ISA model.

***Max*-1 ABS-ISA Equivalence** *For every initial state* q_a°, *and for every* Max-1 *execution* ε_1 *corresponding to an instruction sequence* w^1 *and allowed retirement sequence* w_r^1*:*

$$last(\varepsilon_1(q_a^\circ, \langle w^1, w_r^1 \rangle)) \overset{\text{RF}}{=} \delta_s(\pi_{\text{RF}}(q_a^\circ), w).$$

Proving this is much simpler than the original problem of directly proving ABS-ISA equivalence, since only one instruction is present in ABS at a time. The proof is carried out by induction on the length of instruction sequences, as described in Section 6.2.

6.1 Incremental Flushing

Space limitations prevent us from presenting the complete proofs justifying the incremental flushing approach. We will, however, state the verification steps and resulting proof obligations. We also include a proof sketch for the inductive step of incremental flushing.

The incremental flushing proof step can be split up into three proof obligations, as illustrated in Figure 4a-c. Recall that δ_a takes a state, an input, and a retirement input flag. We use T and F for the values of the retirement input flag, where T forces ABS to retire an instruction, and F prevents it from doing so. The first proof obligation demonstrates the independence of inserting and removing elements from the system:

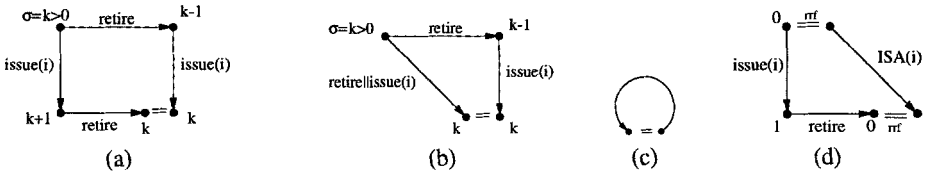

Fig. 4. (a) Proof Obligation 2, the nodes are labeled with their sizes. (b) Proof Obligation 3. (c) Proof Obligation 4. (d) Proof Obligation 5, the ISA induction step.

Proof Obligation 2 (Reordering Step) *For every reachable state q_a s.t. $\sigma(q_a) \geq 1$, and for every input i:*

$$\delta_a(\delta_a(q_a, \langle i, \mathrm{F}\rangle), \langle \mathtt{bubble}, \mathrm{T}\rangle) = \delta_a(\delta_a(q_a, \langle \mathtt{bubble}, \mathrm{T}\rangle), \langle i, \mathrm{F}\rangle).$$

In other words, we must show that the relative order of retirement and issue is immaterial for distinct instructions. The next proof obligation requires that simultaneous issue and retirement of distinct instructions yields the same result as a sequential retirement and issue:

Proof Obligation 3 (Parallel Correctness) *For every reachable state q_a s.t. $\sigma(q_a) \geq 1$, and for every input i:*

$$\delta_a(q_a, \langle i, \mathrm{T}\rangle) = \delta_a(\delta_a(q_a, \langle \mathtt{bubble}, \mathrm{T}\rangle), \langle i, \mathrm{F}\rangle).$$

The final proof obligation illustrates that `bubble` inputs without retirement do not change ABS state:

Proof Obligation 4 (Correctness of Self-Loops) *For every reachable state q_a:*

$$\delta_a(q_a, \langle \mathtt{bubble}, \mathrm{F}\rangle) = q_a.$$

Taken together, these three proof obligations establish the *Incremental Flushing* step of our verification, i.e., that every *Max-n* execution has a functionally equivalent *Max*-1 execution. We next give a brief sketch of the proof.

Proof Sketch: We assume the three Proof Obligations shown above and must show that for every *Max-n* execution ε_n there exists a corresponding *Max*-1 execution ε_1 such that

$$last(\varepsilon_n(q_a^\circ, \langle w, w_r\rangle)) = last(\varepsilon_1(q_a^\circ, \langle w^1, w_r^1\rangle)).$$

We perform the proof in two steps, as illustrated in Figure 3. Given an execution ε_n (Figure 3a) we first show that we can construct a "non-diagonal" execution $\varepsilon_{\hat{n}}$ (Figure 3b) from ε_n that does not have any diagonals nor self-loops, and such that

$$last(\varepsilon_n(q_a^\circ, \langle w, w_r\rangle)) = last(\varepsilon_{\hat{n}}(q_a^\circ, \langle \hat{w}, \hat{w_r}\rangle)).$$

This is proved by induction on the length of ε_n. We use Proof Obligation 3 to replace any diagonal edge with horizontal and vertical edges. Proof Obligation 4 is used to remove the self-loops.

The second step shows that we can derive a *Max*-1 sequence ε_1 (Figure 3c) such that

$$last(\varepsilon_{\hat{n}}(q_a^\circ, \langle \hat{w}, \hat{w}_r \rangle)) = last(\varepsilon_1(q_a^\circ, \langle w^1, w_r^1 \rangle)).$$

We prove this by induction on the distance between the non-diagonal *Max*-$\hat{n}$ execution $\varepsilon_{\hat{n}}$ and the *Max*-1 execution ε_1, where distance is the number of "squares" that separate the two executions. For example, eight squares separate the executions in Figures 3b and 3c. We repeatedly apply Proof Obligation 2, "folding" the *Max*-$\hat{n}$ execution $\varepsilon_{\hat{n}}$ back to the corresponding *Max*-1 execution ε_1. This is possible because the input sequences resulting in ε_n and $\varepsilon_{\hat{n}}$ are completed (defined in Section 4). Each folding is a reordering of independent retires and issues.
End Proof Sketch.

Note that each folding is a rewrite of the execution. It is easy to see that Proof Obligations 2–4 together are a confluent (Church-Rosser) set of rewrite rules, where the *Max*-1 execution is the unique normal form.

6.2 *Max*-1 ABS-ISA Equivalence

The final verification step is to show that all *Max*-1 executions of ABS are functionally equivalent with ISA. We can divide the *Max*-1 execution up into issue-retire fragments that are simple "steps" in the graphical illustration. The proof is a simple induction on the number of these fragments, comparing the execution and retirement of an arbitrary instruction from an arbitrary ABS *Max*-1 state with the result that is retired by ISA. This is illustrated in Figure 4d. Formally:

Proof Obligation 5 (ABS-ISA Induction) *For every initial IA state q_a° and every instruction i:*

$$\delta_a(\delta_a(q_a^\circ, \langle i, \mathtt{F} \rangle), \langle \mathtt{bubble}, \mathtt{T} \rangle) \stackrel{\mathrm{RF}}{=} \delta_s(\pi_{\mathrm{RF}}(q_a^\circ), i).$$

Because we have previously shown that a functionally equivalent *Max*-1 execution can be derived from an arbitrary *Max*-n execution, this step completes the proof of ABS-ISA equivalence.

7 Results

We have mechanically checked Proof Obligations 1-5 for our models using the Stanford Validity Checker (SVC). The three models (IMPL, ABS, and ISA) and the proof obligations were written in a Lisp-like HDL. The proof formulas were constructed by symbolically simulating the models in Lisp. SVC was invoked through a foreign-function interface to decide the validity of the formulas. SVC's built-in support for linear arithmetic was used to model buffer pointers for the IMPL, RB, and ABS annotated RB. We also extended SVC with special read and write updates to support the writeback to the associative memory in the dispatch buffer.

The total CPU run times and number of case splits required are enumerated in Figure 5. The number of case splits is a rough indicator of the relative complexity of the simplified formula.

IMPL-ABS Verification	IMPL Reach. Inv. CPU (sec)	IMPL Reach. Inv. Case Splits	IMPL-ABS CPU (sec)	IMPL-ABS Case Splits
Base Case	1.9	10	0.7	4
Issue	454.8	26,214	130.9	18,686
Dispatch	49.1	12,036	163.3	45,828
Writeback	35.0	842	42.1	4,426
Retire	29.5	8,392	307.0	59,474

(a)

ABS-ISA Verification	CPU (sec)	Case Splits
ABS Inv.	222.2	48,440
Obl. 2	37.6	530
Obl. 3	26.2	2
Obl. 4	7.0	2
Obl. 5	17.8	14

(b)

Fig. 5. (a) SVC run-times and number of case splits required for Proof Obligation 1, specified for each IMPL transaction. (b) SVC run-times and case splits for the verification of ABS. All runs performed on a 200-MHz Intel Pentium Pro system running Redhat Linux.

8 Discussion

Our work addresses two of the major problems in symbolic verification of out-of-order processor designs: the complexity of the out-of-order scheduling logic and the deep effective length of the pipeline. While our IMPL example is far simpler than an actual out-of-order implementation, it is representative of the architectural features which make out-of-order verification difficult for existing techniques.

There is still much work to be accomplished in addressing the complexity limitations encountered by formal methods on practical industrial designs. As these problems are solved, we expect that our approach will be directly applicable. We also anticipate that the incremental flushing approach will find use in a wide variety of verification problems involving very deep pipelines, such as digital-signal processing.

We are currently formalizing the incremental flushing theory in the PVS theorem prover [12]. For each new design, PVS will automatically instantiate the proof obligations and pass them to SVC for automatic verification.

Acknowledgments

We thank Mark Aagaard, Tom Melham, and Carl Seger for reading drafts of this paper. They each provided detailed and helpful feedback.

The second author is supported at Stanford by an NDSEG graduate fellowship. The other authors are partially supported by DARPA under contract number E276. Insight about the difficulties associated with verifying pipelined processors was developed while the third author was a visiting professor at Intel's Strategic CAD Labs in the summer of 1995.

References

1. C. Barrett, D. L. Dill, and J. Levitt. Validity checking for combinations of theories with equality. In *FMCAD '96*, volume 1166 of *LNCS*, pages 187–201, Stanford, CA, USA, November 1996. Springer-Verlag.
2. J. R. Burch. Techniques for verifying superscalar microprocessors. In *DAC*, pages 552–557, Las Vegas, Nevada, USA, June 1996. ACM Press.

3. J. R. Burch and D. L. Dill. Automatic verification of microprocessor control. In David L. Dill, editor, *CAV*, volume 818 of *LNCS*, pages 68–80, Stanford, California, USA, June 1994. Springer-Verlag.
4. W. Damm and A. Pnueli. Verifying out-of-order executions. In H.F. Li and D.K. Probst, editors, *CHARME '97*, pages 23–47, Montreal, Canada, October 1997. Chapman & Hall.
5. J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, 1990.
6. T. A. Henzinger, S. Qadeer, and S. K. Rajamani. You assume, we guarantee: Methodology and case studies. Technical report, Electronics Research Lab, Univ. of Californaia, Berkeley, CA 94720, 1998.
7. R. B. Jones, D. L. Dill, and J. R. Burch. Efficient validity checking for processor verification. In *ICCAD'95*, November 1995.
8. R. B. Jones, C.-J. H. Seger, and D. L. Dill. Self-consistency checking. In *FMCAD '96*, volume 1166 of *LNCS*, pages 159–171, Stanford, CA, USA, November 1996. Springer-Verlag.
9. B. Jonsson. On decomposing and refining specifications of distributed systems. In J. W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems. Models, Formalisms, Correctness.*, volume 430 of *LNCS*, pages 361–385, Mook, The Netherlands, May-June 1989. Springer-Verlag.
10. S. Katz. Refinement with global equivalence proofs in temporal logic. In D. A. Peled, V. R. Pratt, and G. J. Holzmann, editors, *Partial Order Methods in Verification*, volume 29 of *DIMACS, Series in Discrete Mathematics and Theoretical Computer Science*, pages 59–78, Princeton, NJ, USA, 1996. Amer. Math. Society.
11. K. McMillan. Verification of an implementation of Tomasulo's algorithm by compositional model checking. Appears in this volume.
12. S. Owre, S. Rajan, J. M. Rushby, N. Shankar, and M. K. Srivas. PVS: Combining specification, proof checking, and model checking. In R. Alur and T.A. Henzinger, editors, *CAV '96*, volume 1102 of *LNCS*, pages 411–414, New Brunswick, NJ, July/Aug 1996. Springer-Verlag.
13. J. Sawada and W. A. Hunt. Processor verification with precise exceptions and speculative execution. Appears in this volume.
14. J. Sawada and W. A. Hunt. Trace table based approach for pipelined microprocessor verification. In Orna Grumberg, editor, *CAV '97*, volume 1254 of *LNCS*, pages 364–375, Haifa, Israel, June 1997. Springer-Verlag.
15. P. J. Windley and J. R. Burch. Mechanically checking a lemma used in an automatic verification tool. In *FMCAD'96*, pages 362–376, November 1996.