Reducing Manual Abstraction in Formal Verification of Out-of-Order Execution

Robert B. Jones[1,2], Jens U. Skakkebæk[1], 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. Several methods have recently been proposed for verifying processors with out-of-order execution. These methods use intermediate abstractions to decompose the verification process into smaller steps. Unfortunately, the process of manually creating intermediate abstractions is very laborious. We present an approach that dramatically reduces the need for an intermediate abstraction, so that only the scheduling logic of the implementation is abstracted. After the abstraction, we apply an enhanced *incremental-flushing* approach to verify the remaining circuitry by comparing the processor description against itself in a slightly simpler configuration. By induction, we demonstrate that any reachable configuration is equivalent to the simplest possible configuration. Finally, we prove correctness on the simplest configuration. The approach is illustrated with a simple example of an out-of-order execution core.

1 Introduction

Several techniques for formally verifying out-of-order microprocessor designs using theorem proving have recently been suggested [4, 10–12]. These techniques all use some form of intermediate abstraction to bridge the gap in abstraction level between the implementation and the specification, as defined by an *instruction-set architecture* (ISA).

Creating such intermediate abstractions manually and then showing the correspondence between the implementation and the intermediate abstraction is laborious, even for high-level models. Omitting the intermediate abstraction and manually developing the abstraction relation between the implementation and the ISA is even harder. 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 have devised an approach for pipelined microarchitectures that automatically generates the abstraction function by *flushing* the implementation state [3]. The technique has been extended to dual-issue and super-scalar architectures [2, 8, 13]. However, these techniques do 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 have previously proposed ***incremental flushing***, an extension to the Burch and Dill flushing approach that inductively empties the buffer in smaller proof steps [12]. We have applied it to automate part of the verification process of out-of-order designs. The approach requires, however, that the out-of-order core is abstracted into an in-order version. In this paper, we extend the incremental-flushing approach to directly reason about the out-of-order core also. This avoids the need for the in-order abstraction of our earlier approach. The implementation abstraction that is still required is comparatively minimal, and the automated incremental flushing approach can cover a much larger portion of the original design. This automates the generation of the abstraction function and significantly reduces the manual effort required.

The extended technique only requires that the internal scheduling logic of the processor be manually abstracted. An instruction is processed through a number of internal ***steps***, which each may take several cycles. The scheduling logic affected determines which buffer entries, datapath resources, and busses different instructions and steps are assigned to. We apply induction to show that the implementation executing any number of instructions (up to the maximum allowed) is functionally equivalent with the same implementation executing only one instruction at a time. We finally complete the verification by checking the implementation with one instruction against the ISA. This proof is much simpler, since the bypass and buffering logic can be simplified away in the proofs. Note that to make the induction work, it must be possible to stall each stage of the out-of-order pipeline independently.

We use the same simple model of an out-of-order execution core to illustrate our approach that we used previously [12]. Although this 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. We have discharged the proof obligations for the simple example using the Stanford Validity Checker (SVC).

2 Related Work

Sawada and Hunt's theorem-proving approach uses a table of history variables, called a ***micro-architectural execution trace table*** (MAETT) [10, 11]. 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, we do not need an intermediate abstraction of the circuit, only the scheduling logic is abstracted. We then use an incremental flushing technique to automatically generate the abstraction function, 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]. They verify 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 and the corresponding large buffers that are required. Damm and Pnueli's abstraction non-deterministically represents all possible instruction sequences which observe dataflow dependencies. Our non-deterministic scheduler abstraction also observes dataflow dependencies, but is additionally constrained by allowable resource allocations (e.g., buffer entries) in the implementation. Applying their method to architectures with in-order retirement would require manual proof by induction that the intermediate abstraction satisfies the ISA. We automate the proof obligations with incremental flushing.

Hosabettu et al. use a technique for decomposing the abstraction function and have applied it to the example of Sawada and Hunt with out-of-order retirement [7]. Although this aids in finding an appropriate abstraction function, manual intervention is needed in its construction.

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

McMillan model checks the Tomasulo algorithm by manually decomposing the proof into smaller correctness proofs of the internal steps [9]. He also uses a reduction technique based on symmetry to extend the proof to a large number of execution units. Berezin et al. abstract the data path by introducing a data structure called a reference table. Each entry in the reference table corresponds to an uninterpreted term representing computation results of instructions [1]. They have applied their technique to Tomasulo's algorithm. However, the size of the state space grows exponentially with the number of concurrent instructions. Designs with in-order retirement contain a large reorder buffer and can contain many instructions executing simultaneously. In contrast to both automated model-checking approaches, our theorem-proving based method generalizes to arbitrary buffer sizes.

3 Preliminaries

The desired behavior of a processor is defined by an *instruction-set architecture* (ISA). The ISA represents the programmer-level view of a machine that executes instructions sequentially. The ISA for our example is shown in Figure 1a. The ISA state consists of a register file (RF), while the next-state function is computed with a generic execution unit (EU) that can execute any instruction. The ISA

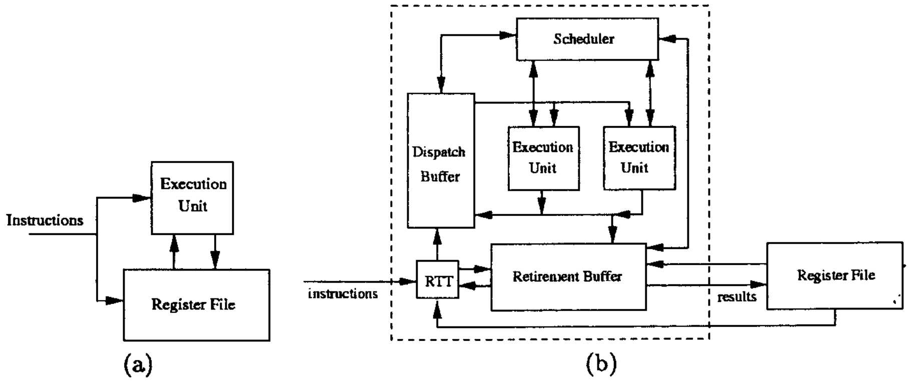

Fig. 1. (a) The simple ISA model. (b) Instruction flow in our out-of-order execution core IMPL.

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 implement the ISA more aggressively. 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. An instruction is processed in a number of *steps*, which may each last a number of cycles: 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.

Figure 1b also shows the *scheduling logic*, which handles the allocation of hardware resources and instruction flow. Scheduling must determine (1) which slot in the DB to allocate at issue, (2) when to dispatch a ready instruction and which EU to dispatch it to, (3) when an EU writes back a completed execution result, and (4) when to retire a completed instruction. We call this collection of resource allocation and dataflow decisions from the scheduling logic the *choice* for a given cycle.

There are obviously many sound scheduling algorithms, and many allowable scheduling choices exist for a given configuration. Which choices are allowable is determined by the state of other instructions and available hardware resources. For example, a sound but inefficient scheduling algorithm would only allow one instruction to execute at a time—greatly simplifying the interaction between instructions. An optimal scheduling algorithm would execute instructions in whatever dataflow order makes the best use of execution resources. An implementable scheduling algorithm falls somewhere in the middle and must balance execution performance against implementation considerations.

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. 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 verification discussed in this paper uses a model with unbounded buffers.

4 The Approach

As in [12], the goal of our 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 an initial state q_i and an input instruction i and returns a new state q_i', e.g., $q_i' = \delta_i(q_i, i)$. 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 $\pi_{\mathrm{RF}}(q_i)$ to denote the register file contents in state q_i - which we define as the specification state. For clarity in presentation, we define $q_{i1} \overset{\mathrm{RF}}{=} q_{i2}$ to be $\pi_{\mathrm{RF}}(q_{i1}) = \pi_{\mathrm{RF}}(q_{i2})$, and we will sometimes use $\overset{\mathrm{RF}}{=}$ 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 formally as:

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

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

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.

Our approach has three steps. First, we locate and abstract the IMPL scheduling logic and prove the abstraction correct. We refer to the abstracted implementation as SAI (scheduler-abstracted implementation). In the second step, we use incremental flushing to show that SAI with an abstracted scheduler calculates the same results as if the instructions were executed one at a time. Note that while the functional results should be identical, the timing of the results will of course be different. This proves the correctness of the reordering control logic. Finally, we show that SAI with an abstracted scheduler executing one instruction at a time satisfies the ISA.

5 First Step: Abstracting the Scheduling Logic

We first identify the scheduling logic in the design and its interface to the rest of the circuit. We wish to replace the original scheduling logic with the most general scheduling algorithm that still provides legal choices to the rest of the circuit. For example, the abstracted scheduling logic for our simple example will (1) issue an instruction to any empty slot in the DB, (2) dispatch an instruction to any available execution unit, (3) write back results from any execution unit that has finished executing, and (4) retire any instruction with result data. In a given state, the abstracted scheduling logic in SAI non-deterministically chooses an allocation based on the current state of the SAI. The non-determinism is implemented as an extra, unconstrained input.

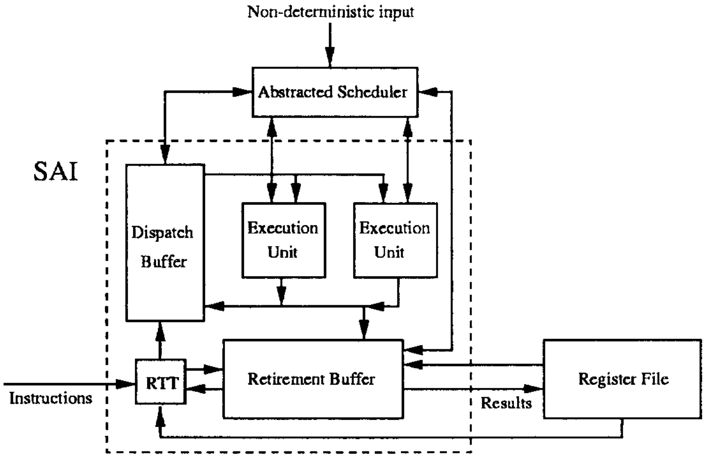

Fig. 2. Instruction flow in SAI with the abstracted resource allocator.

The SAI with an abstracted scheduler is illustrated in Figure 2. The abstract scheduler monitors the state of SAI and provides SAI with a scheduling choice for every instruction input. Naturally, we want the abstracted scheduler to make *legal* choices that only allocate free resources and advance only ready instructions from one stage to the next. For example, only instructions that have completed executing may be written back and retired. Identifying and abstracting the scheduling logic in a realistic design requires a detailed understanding of the circuit and may be error-prone. Fortunately, soundness of our approach is not compromised by a bad selection of abstracted scheduler. The later proof steps will fail if the abstracted scheduler in SAI is either incorrect or too general to verify its behavior against ISA. Note, however, that we do not require the scheduler to be centralized. The technique is equally applicable to a distributed scheduler, where each part of the scheduler is appropriately abstracted.

We first show that the abstract scheduler is sufficiently general to capture all the possible choice outputs that the implementation scheduler makes. We then extend this result with a composition argument to show that SAI with the abstracted scheduler is an appropriate abstraction of IMPL. Let $\mathcal{S}_i$ be the transition function of the implementation scheduler and let $\mathcal{S}_a$ be the transition function of the abstract scheduler. $\mathcal{S}_a$ takes an extra, non-deterministic input i_{nd}. We must show that for each step that $\mathcal{S}_i$ makes, there exists an $\mathcal{S}_a$ step such that the choice outputs are identical:

Proof Obligation 1 (Scheduler Abstraction Correctness) *For every reachable state q_i of IMPL and for every input i, there exists an input i_{nd} such that*

$$out(\mathcal{S}_i(q_i, i)) = out(\mathcal{S}_a(q_a, i, i_{nd})).$$

One way of instantiating the abstract scheduler for this proof is to use an oracle which observes the original scheduler's behavior and knows how the non-deterministic input affects the abstract scheduler.

Next, we must establish that SAI with the abstracted scheduler is an appropriate abstraction of IMPL. We define δ_a to be the SAI next-state function, which takes an initial state q_a and a pair consisting of an input instruction i and scheduler choice ch and returns a new state q'_a, e.g., $q'_a = \delta_a(q_a, \langle i, ch \rangle)$. We extend the definition of δ_a to sequences of instruction inputs w and choice sequence $w_{ch} = ch_0 \ldots ch_n$ such that $q'_a = \delta_a(q_a, \langle w, w_{ch} \rangle)$[1]. We say that a choice sequence w_{ch} is $\mathcal{S}_a(q_a, w)$-generated, if it is obtained by stimulating the abstract scheduler to provide a sequence of choices corresponding to the instruction sequence w from the state q_a. We define states q_i of IMPL and q_a of SAI to be *consistent* when $q_i \overset{\mathrm{RF}}{=} q_a$, i.e., they have identical architecturally visible states. Using Proof Obligation 1 and a composition argument, we can prove that:

IMPL-SAI Refinement *For every instruction sequence w and every pair of consistent initial states q_i°, q_a°, there exists a $\mathcal{S}_i(q_a^\circ, w)$-generated choice sequence*

[1] The pair of sequences $\langle w, w_{ch} \rangle$ is easily derived from the corresponding sequence of pairs $\langle i_0, ch_0 \rangle, \ldots, \langle i_n, ch_n \rangle$.

w_{ch} such that

$$\delta_i(q_i^\circ, w) \overset{RF}{=} \delta_a(q_a^\circ, \langle w, w_{ch}\rangle).$$

We prove this by providing the following witness. By induction, we extend Proof Obligation 1 to work on sequences of inputs and obtain a $\mathcal{S}_a(q_a^\circ, w)$-generated sequence w_{ch} that is equal to the sequence that is output from the implementation scheduler. Since SAI was obtained from IMPL by abstracting only the resource allocation logic, the property follows trivially.

Note that this proof requires reachability invariants for IMPL and SAI. Finding the reachability invariant for IMPL is necessary for any inductive method, and is not unique to our approach. Finding the reachability invariant for SAI is straightforward, because of the minimal changes from IMPL.

6 Second Step: Functional Equivalence of SAI and ISA

The second step in the verification is to prove that SAI with the abstract scheduler satisfies ISA. Formally:

SAI-ISA Equivalence *For every completed instruction sequence w, initial SAI state q_a°, and $\mathcal{S}_a(q_a^\circ, w)$-generated sequence of choices w_{ch}:*

$$\delta_a(q_a^\circ, \langle w, w_{ch}\rangle) \overset{RF}{=} \delta_s(\pi_{RF}(q_a^\circ), w).$$

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 RB with the ISA model. The Burch-Dill flushing technique would unroll SAI to the depth of the RB, resulting in a logical expression too large for the decision procedure to check.

We extend the *incremental-flushing* approach presented in [12] to overcome this problem. Rather than flushing the entire pipeline directly, a set of smaller, inductive flushing steps is performed. Taken together, these proof obligations imply the full, monolithic flushing operation. To illustrate the approach, consider the graphical presentation of two different *executions* (state sequences) of SAI in Figure 3. We define the *execution* of a system as the sequence of states that the system passes through when executing a given input sequence. For instance, the execution indicated in Figure 3a is a result of executing the instruction sequence:

$$i_1, i_2, \texttt{bubble}, \texttt{bubble}, i_3, \texttt{bubble}, i_4, i_5, \texttt{bubble}, \texttt{bubble}, i_6, \texttt{bubble}, \texttt{bubble}.$$

with some choice sequence that appropriately allocates the resources so that all instructions have retired in the final state state. Apart from self-loops indicating internal execution, edges are only traversed when instructions are issued or retired.

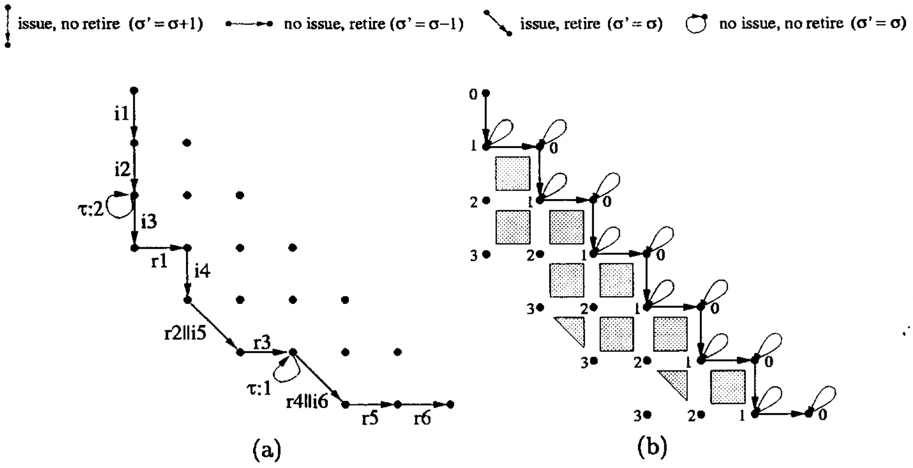
issue, no retire (σ' = σ+1)
no issue, retire (σ' = σ−1)
issue, retire (σ' = σ)
no issue, no retire (σ' = σ)
i1
i2
τ:2
i3
r1
i4
r2||i5
r3
τ:1
r4||i6
r5
r6
(a)
0
1
2
3
(b)

Fig. 3. (a) A *Max-n* execution ε_n. 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. (b) An equivalent *Max*-1 execution ε_1. The squares indicate the distance between ε_n and ε_1.

We use $\varepsilon(q_a, \langle w, w_{ch} \rangle)$ to denote the execution resulting from the application of δ_a to a state q_a and the input sequence pair $\langle w, w_{ch} \rangle$. We define $last(\varepsilon(q_a, \langle w, w_{ch} \rangle))$ as the last state of the execution. Note that, by definition $last(\varepsilon(q_a, \langle w, w_{ch} \rangle)) = \delta_a(q_a, \langle w, w_{ch} \rangle)$. 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 3b. We call an execution which contains states with most size 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). An example of a *Max*-1 execution corresponding to the execution above could be

$$i_1, \texttt{bubble}^4, i_2, \texttt{bubble}^4, i_3, \texttt{bubble}^4, i_4, \texttt{bubble}^4, i_5, \texttt{bubble}^4, i_6, \texttt{bubble}^4.$$

where $\texttt{bubble}^4 = \texttt{bubble}, \texttt{bubble}, \texttt{bubble}, \texttt{bubble}$. The execution is illustrated in Figure 3b.

The first step of the SAI-ISA verification establishes that:

Incremental-Flushing Induction Step *For every initial state q_a°, and for every* Max-*n* *execution $\varepsilon_n(q_a, \langle w, w_{ch} \rangle)$, there exists $\langle w^1, w_{ch}^1 \rangle$ derived from input pair $\varepsilon_n(q_a, \langle w, w_{ch} \rangle)$ and a corresponding* Max-1 *execution $\varepsilon_1(q_a, \langle w^1, w_{ch}^1 \rangle)$ such that:*

$$\mathrm{last}(\varepsilon_n(q_a^\circ, \langle w, w_{ch} \rangle)) = \mathrm{last}(\varepsilon_1(q_a^\circ, \langle w^1, w_{ch}^1 \rangle)).$$

A *Max*-1 execution is derived from a *Max-n* execution by "stretching" the w and w_{ch} sequences with the appropriate `bubbles` and stalling choices, respectively, to stall the relevant parts of the out-of-order core. The intuition behind this approach is that the final results of *Max-n* and *Max*-1 executions should be identical—because `bubbles` and stalling choices should not affect functional behavior. Clearly, if enough `bubbles` are inserted between subsequent instructions only one instruction will be in the pipeline at a time. In this situation it is computationally manageable to compare SAI with ISA, since the bypass control logic can be discarded in the proof. Section 6.1 details the proof obligations for this step and describes how we proved this property on our example.

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

Incremental Flushing ISA Step *For every initial state q_a°, and every* Max-1 *execution ε_1 corresponding to an instruction sequence w^1 and every $S_i(q_a^\circ, w^1)$-generated choice sequence w_{ch}^1:*

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

Proving this is much simpler than the original problem of directly proving SAI-ISA equivalence, since only one instruction is in the machine at any given time (because of the stretching `bubbles` and stalling choices). The proof is carried out by induction on the length of instruction sequences, as described in Section 6.2.

6.1 Inductive Step

The incremental flushing proof step can be split up into three proof obligations. First, we identify the maximum number of cycles required to symbolically simulate the implementation in order to ensure that at least one instruction is retired. This is used to prove termination of the induction proof. Let δ_a^n denote n cycles of symbolic execution. Formally, we must prove that:

Proof Obligation 2 (Retirement Upper-Bound) *There exists an upper bound u, such that for every reachable state q_a such that $\sigma(q_a) \geq 1$ and input sequence pair $\langle w, w_{ch}\rangle$, at least one active instruction from q_a will be retired between q_a and $\delta_a^u(q_a, \langle w, w_{ch}\rangle)$.*

That is, we make a progress assumption that the implementation retires an instruction within u cycles. We derive u by a *worst-case* analysis and determine the longest path that an issued instruction could potentially follow before being retired.

The upper bound u is assumed in the main induction. As we shall see, the induction case is used to inductively move the last issued instruction to the end of the execution sequence. In each application, independently executing instruction steps are reordered. This reordering is performed by moving the instruction till after the steps of the previously issued instructions.

In each application of the induction case, a subsequence is selected out of the execution such that an instruction i is issued in the first cycle of the subsequence. We denote the length of the subsequence by v, and will choose it to be $\geq u$. The length of the subsequence is doubled in the application of the induction case:

the v choices are split up in a way that the first v steps allow SAI to perform all steps that are not dependent on i. The steps related to i are then ***replayed*** in the remaining cycles. As a consequence, the freshly-issued instruction i and its steps are delayed by v cycles.

(a)

Cycle #	1	2	3	4	5
issue	i4	—	—	—	—
dispatch	—	d4	d3	—	—
writeback	w2	—	w4	w3	—
retire	r1	r2	—	—	r3

(b)

Cycle #	1	2	3	4	5	6	7	8	9	10
issue	—	—	—	—	—	i4	—	—	—	—
dispatch	—	—	d3	—	—	—	d4	—	—	—
writeback	w2	—	—	w3	—	—	—	w4	—	—
retire	r1	r2	—	—	r3	—	—	—	—	—

Fig. 4. (a) A choice sequence w_{ch}. (b) A stretched version $w_{ch}{}'$ of the original choice sequence w_{ch}.

To illustrate, consider the scheduling sequence w_{ch} shown in Figure 4a. Each vertical box corresponds to a choice and the labels in, dn, wn, and rn respectively denote which dispatch buffer entry to store an issued instruction in, which dispatch buffer entry to dispatch, which completed instruction to write back, and whether or not to allow retirement of an instruction ready for retirement. Each number identifies a particular instruction n. For instance, the first choice retires instruction 1, writes back instruction 2, and issues instruction 4. A choice field which keeps a particular resource allocation unchanged is denoted with "—".

A scheduling sequence $w_{ch}{}'$ is constructed by adding `bubbles` and stalling choices to w_{ch} (Figure 4b). Observe that the ordering of the issue, dispatch, writeback, and retirement choices for a given instruction are maintained. The only difference is the delayed issue of instruction 4 and its subsequent dispatch and writeback. On a per-instruction basis, the resources in $w_{ch}{}'$ and w_{ch} must be the same and occur in the same order. This crucial requirement guarantees that the resulting partially-executed state is the same in both cases and facilitates an inductive proof over SAI state.

In the induction case, the length of the subsequence, v, must be chosen so that it is at least u cycles and long enough to make sure that the instruction can properly be moved passed the steps of other instructions. In our example, v must be at least double the maximum execution time in an execution unit, i.e., which in total is less than $2u$ (from Proof Obligation 2 we know that the time that any instruction spends in the execution unit is less than u). By doing this, we are able to delay the instruction sufficiently far to avoid resource contention when reordering.

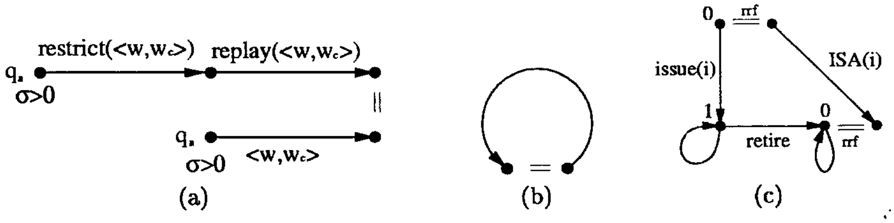

Fig. 5. (a) Illustration of Proof Obligation 3; the nodes are labelled with their sizes. (b) Illustration of Proof Obligation 4. We must prove that self loops return to the same state. (c) Illustration of Proof Obligation 5, the ISA induction step.

Given an input-sequence pair $\langle w, w_{ch}\rangle$, define *restrict*$_i(\langle w, w_{ch}\rangle)$ to be the projection of all elements of $\langle w, w_{ch}\rangle$ not depending on i. Similarly, we define *replay*$_i(\langle w, w_{ch}\rangle)$ to denote the projection of the elements of $\langle w, w_{ch}\rangle$ that depend on i. The proof obligation is then:

Proof Obligation 3 (Incremental Step) *For every reachable state q_a such that $\sigma(q_a) \geq 1$, and for every input-sequence pair $\langle w, w_{ch}\rangle$ such that the first element of w is a non-*`bubble` *instruction and w_{ch} is $S_i(q_a, w)$-generated:*

$$\delta_a^v(q_a, \langle w, w_{ch}\rangle) = \delta_a^{2v}(q_a, \langle w', w_{ch}'\rangle).$$

where $\langle w', w_{ch}'\rangle$ is the concatenation of $\mathrm{restrict}_i(\langle w, w_{ch}\rangle)$ *and* $\mathrm{replay}_i(\langle w, w_{ch}\rangle)$.

In other words, we must show that the stretched sequence results in the same state as the original sequence. The proof obligation is illustrated in Figure 5a.

As we shall see below, in the proof of Proof Obligation 3 it is sufficient to consider the cases where the necessary resource is available so that the instruction being moved can be scheduled appropriately and avoid resource contention. This weakening assumption can be added to the proof obligation.

Note that Proof Obligation 3 requires also that internal registers with auxiliary values to agree on the resulting states. To illustrate, the replayed instructions in our model may get their source operands from the RF rather than the RB. The fields in the dispatch buffer indicating the physical sources of the operands at issue may differ and should be set to some reset value after use.

Also observe that in each application of the induction step, more than one instruction may retire within the v steps. Naturally, the worst-case upper bound u (number of cycles before an instruction is guaranteed to retire) and therefore v may be quite large in some designs due to execution units with long latencies. This could result in symbolic expressions that are too large to check. In these cases, the execution units and associated arbitration logic must be abstracted separately.

The final proof obligation states that `bubble` inputs with stalling choices do not change SAI state (illustrated in Figure 5b):

Proof Obligation 4 (Correctness of Self-Loops) *For every reachable state q_a, instruction i, and stalling choice ch_{st}:*

$$\delta_a(q_a, \langle i, ch_{st} \rangle) = q_a.$$

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 provide 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

$$\varepsilon_n(q_a^\circ, \langle w, w_{ch} \rangle) \overset{\text{last}}{=} \varepsilon_1(q_a^\circ, \langle w^1, w_{ch}^1 \rangle).$$

We prove this by complete induction on the "distance" between the non-diagonal *Max-n* execution ε_n and the *Max*-1 execution ε_1, where distance is the number of "squares" and "triangles" that separate the two executions. For example, eight squares and two triangles separate the executions in Figures 3a and 3b.

First, if all states in ε_n have $\sigma = 0$ in states where instructions are issued, then we have a *Max*-1 sequence and are trivially done—no more than one instruction is ever executed at a time. This is the base case.

Otherwise, we reduce the distance by inductively moving the last instruction issued in a state of $\sigma \geq 1$ back until $\sigma = 0$. We repeat this until all instructions do not overlap in execution and thus obtain the base case.

In the induction, we repeatedly choose the last such instruction i and identify the choice subsequence of length v starting with i. If necessary, we can make the subsequence long enough, by extending ε_n with extra, trailing stalling choices, using Proof Obligation 4. We then apply Proof Obligation 3. If we have added the previously mentioned weakening assumption that resources are available at the end of v, we can satisfy this by locating the last place that the resource was freed and delay the following rescheduling till after the v cycles, using Proof Obligation 4[2].

We know that the number of internal steps between the instruction issue and the end of the execution sequence monotically decreases in each application, since we are moving the instruction passed at least one step of any kind in each application. We also know that we are able to move all the internal steps of the instruction, since the length v is greater than u. Furthermore, since the instruction sequence is completed, we know that we are also moving the instruction past

[2] In implementations where the freeing and scheduling of the resource overlap in time, we can prove a separate lemma that shows the correctness of the slight delay of the rescheduling after the freeing.

instruction retires, each time monotonically decreasing the distance as defined above and eventually reaching the base case. The induction is thus well-founded. **End Proof Sketch**

6.2 ISA Step

The final verification step is to show that all *Max*-1 executions of SAI are functionally equivalent with ISA. Because the instruction sequence w^1 completes all executions (i.e., leaves no outstanding instructions in the pipeline), we can divide it up into issue-retire fragments in the *Max*-1 execution. We can assume that each fragment has length u, since if one does not, we can apply Proof Obligation 4 to add or remove the necessary stalling cycles. The proof is an induction on the number of such fragments, comparing the execution and retirement of an arbitrary instruction from an arbitrary *Max*-1 initial state with the result that is retired by ISA. This is illustrated in Figure 5c. Formally:

Proof Obligation 5 (SAI-ISA Induction) *For every initial IA state q_a°, instruction i, and input sequence pair $\langle w, w_{ch} \rangle$ of length u containing only i as its first instruction:*

$$\delta_a^u(q_a^\circ, \langle w, w_{ch} \rangle) \overset{RF}{=} \delta_s(\pi_{RF}(q_a^\circ), i).$$

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 SAI-ISA equivalence.

7 Mechanical Verification

We have mechanically checked our simple SAI abstraction and Proof Obligations 3-5 for our example using the Stanford Validity Checker (SVC). The proofs finished in minutes. The three models (IMPL, SAI, and ISA) and the proof obligations are written in a Lisp-like HDL. The proof formulas are constructed by symbolically simulating the models in Lisp. SVC is invoked through a foreign-function interface to decide the validity of the formulas.

The mechanical verification of Proof Obligation 3 has exposed several bugs in the way the choice signals were introduced in the SAI. For instance, the original formulation did not stall retirement properly. This was detected in the verification with SVC when a stretched execution retired an instruction that the original execution did not. This illustrates the ability of the incremental-flushing step to detect possible bugs in the exposing of the scheduler interface.

We were able to locate the error using the counter example information that SVC produced when the error was reached. The counter example is a conjunction of predicates satisfied in the interpretation that falsifies the proof obligation. The user can apply this information in the context of the original system model to debug the error.

8 Discussion

This work addresses a recurring difficulty encountered in symbolic verification of out-of-order processor designs: the difficulty of creating an appropriate implementation abstraction. The extension of the incremental flushing technique enables significantly more automation than the basic technique alone and reduces the need for manual abstraction. On the down side, the computational complexity of the resulting proof obligations is higher, since more steps of symbolic simulations are performed in each proof step. This was not an issue in the verification of our very simple example. However, more research is needed to address the application of the approach to more realistic designs. Also, work is needed to establish if our techniques for avoiding resource contentions during reordering are sufficient for more complex architectures.

It has been argued that localizing the (possibly distributed) scheduling logic in a circuit will be difficult. Our assumption of practice is that optimal scheduling algorithms are determined empirically by simulation and that location and interfaces are clearly identifiable when plugging in different scheduler implementations. We expect that this knowledge can be exploited when locating the scheduling logic for verification purposes.

Acknowledgments

We would like to thank the anonymous reviewers for their comments to the paper. The first author is supported at Stanford by an NDSEG graduate fellowship. The other authors are partially supported by DARPA under contract number DABT63-96-C-0097-P00002. 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. S. Berezin, A. Biere, E. Clarke, and Y. Zhu. Combining symbolic model checking with uninterpreted functions for out-of-order processor verification. Appears in this volume.
2. J. R. Burch. Techniques for verifying superscalar microprocessors. In *33rd ACM/IEEE Design Automation Conference*, 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, *Computer Aided Verification. 6th International Conference*, volume 818 of *LNCS*, pages 68–80, Stanford, California, USA, June 1994. Springer-Verlag.
4. Werner Damm and Amir Pnueli. Verifying out-of-order executions. In Hon F. li and David K. Probst, editors, *Advances in Hardware Design and Verification: IFIP WG10.5 Internation al Conference on Correct Hardware Design and Verification Methods (CHARME)*, 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 California, Berkeley, CA 94720, 1998.
7. R. Hosabettu, M. Srivas, and G. Gopalakrishnan. Decomposing the proof of correctness of pipelined microprocessors. In A. J. Hu and M. Y. Vardi, editors, *Computer Aided Verification (CAV'98)*, volume 1427 of *Lecture Notes in Computer Science*, pages 122–134, Vancouver, Canada, June-July 1998. Springer-Verlag.
8. R. B. Jones, D. L. Dill, and J. R. Burch. Efficient validity checking for processor verification. In *Proceedings: IEEE International Conference on Computer-Aided Design (ICCAD)*, November 1995.
9. K. McMillan. Verification of an implementation of Tomasulo's algorithm by compositional model checking. In A. J. Hu and M. Y. Vardi, editors, *Computer Aided Verification (CAV'98)*, volume 1427 of *Lecture Notes in Computer Science*, pages 110–121, Vancouver, Canada, June-July 1998. Springer-Verlag.
10. J. Sawada and W. A. Hunt. Trace table based approach for pipelined microprocessor verification. In Orna Grumberg, editor, *Computer-Aided Verification, CAV '97*, volume 1254 of *Lecture Notes in Computer Science*, pages 364–375, Haifa, Israel, June 1997. Springer-Verlag.
11. J. Sawada and W. A. Hunt. Processor verification with precise exceptions and speculative execution. In A. J. Hu and M. Y. Vardi, editors, *Computer Aided Verification (CAV'98)*, volume 1427 of *Lecture Notes in Computer Science*, pages 135–146, Vancouver, Canada, June-July 1998. Springer-Verlag.
12. J. U. Skakkebæk, R. B. Jones, and D. L. Dill. Formal verification of out-of-order execution using incremental flushing. In A. J. Hu and M. Y. Vardi, editors, *Computer Aided Verification (CAV'98)*, volume 1427 of *Lecture Notes in Computer Science*, pages 98–109, Vancouver, Canada, June-July 1998. Springer-Verlag.
13. P. J. Windley and J. R. Burch. Mechanically checking a lemma used in an automatic verification tool. In *Proceedings: International Conference on Formal Methods in Computer-Aided Design*, pages 362–376, November 1996.