Mothers of Pipelines

Sava Krstić, Robert B. Jones, and John O'Leary [1,2]

Strategic CAD Labs, Intel Corporation, Hillsboro, Oregon, USA

Abstract

We present a method for pipeline verification using SMT solvers. It is based on a non-deterministic "mother pipeline" machine (*MOP*) that abstracts the instruction set architecture (*ISA*). The *MOP* vs. *ISA* correctness theorem splits naturally into a large number of simple subgoals. This theorem reduces proving the correctness of a given pipelined implementation of the *ISA* to verifying that each of its transitions can be modeled as a sequence of *MOP* state transitions.

Keywords: Pipeline verification, stuttering simulation, confluence, SMT solvers

1 Introduction

Proving correctness of microarchitectural processor pipelines (*MA*) with respect to their instruction set architecture (*ISA*) amounts to establishing a simulation relation between the behaviors of *MA* and *ISA*. There are different ways in the literature to formulate the correctness theorem that relates the steps of the two machines [1], but the complexity of the *MA*'s step function remains the major impediment to practical verification. The challenge is to find a systematic way to break the verification effort into manageable pieces.

We propose a solution based on the obvious fact that the execution of any instruction can be seen as a sequence of smaller actions (let us call them *mini-steps* in this informal overview), and the observation that the mini-steps can be understood at an abstract level, without mentioning any concrete *MA*. Examples of mini-steps are fetching an instruction, getting an operand from the register file, having an operand forwarded by a previous instruction in the pipeline, writing a result to the register file, and retiring. We introduce an intermediate specification *MOP* between *ISA* and *MA* that describes the execution of each instruction as a sequence of mini-steps. By design, our highly non-deterministic intermediate specification admits a broad range of implementations. For example, *MOP* admits implementations that

[1] Thanks to Arvind and Jesse Bingham for their comments.

[2] Email: {sava.krstic,robert.b.jones,john.w.oleary}@intel.com

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

are out-of-order or not, speculative or not, superscalar or not, etc. This approach allows us to separate the implementation-independent proof obligations that relate *ISA* to *MOP* from those that rely upon the details of the *MA*. This can potentially amortize some of the proof effort over several different designs.

The concept of *parcels*, formalizing partially-executed instructions, will be needed for a thorough treatment of mini-steps. We will follow the intuition that from any given state of any *MA* one can always extract the current state of its *ISA* components and infer a queue of parcels currently present in the *MA* pipeline. In Section 2, we give a precise definition of a transition system *MOP* whose states are pairs of the form ⟨*ISA* state, queue of parcels⟩, and whose transitions are mini-steps as described above. Intuitively, it is clear that with a sufficiently complete set of mini-steps we will be able to model any *MA* step in this transition system as a sequence of mini-steps. Similarly, it should be possible to express any *ISA* step as a sequence of mini-steps of *MOP*.

Figure 1 indicates that correctness of a microarchitecture *MA* with respect to *ISA* is implied by correctness results that relate these machines with *MOP*. In Section 3, we will prove the crucial *MOP* vs. *ISA* correctness property: despite its nondeterminism, all *MOP* executions correspond to *ISA* executions. The proof rests on the local confluence of *MOP*—a technique pioneered by Shen and Arvind [15]. We first prove the correspondence in the Burch-Dill style, and then extend it to stuttering bisimulation between *bounded MOP* and *ISA*.

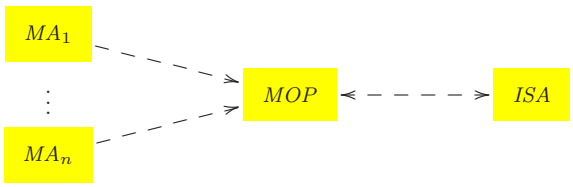


Fig. 1. With transitions that express atomic steps in instruction execution, a *mother of pipelines MOP* simulates the *ISA* and its multiple microarchitectural implementations. Simulation *à la* Burch-Dill flushing justifies the arrow from *MOP* to *ISA*.

The *MA* vs. *MOP* relationship is discussed in Section 4. We will see that all one needs to prove here is a precise form of the simulation mentioned above: there exists an abstraction function that maps *MA* states to *MOP* states such that for any two states joined by a *MA* transition, the corresponding *MOP* states are joined by a sequence of mini-steps.

MA vs. *MOP* vs. *ISA* correctness theorems systematically reduce to numerous subgoals, suitable for automated SMT solvers ("satisfiability modulo theories"). We used CVC Lite [4] and our initial experience is discussed in Section 5.

2 *MOP* Definition

The *MOP* definition depends on the *ISA* and the class of pipelined implementations that we are interested in. The particular *MOP* described in this section has a simple load-store *ISA* and can model complex superscalar implementations with out-of-order execution and speculation.

2.1 The Instruction Set Architecture

instruction *imem.pc*	actions
opc1 dest src1 src2	$pc := pc + 4 \quad rf.dest := \mathsf{alu}\ opc1\ (rf.src1)\ (rf.src2)$
opc2 dest src1 imm	$pc := pc + 4 \quad rf.dest := \mathsf{alu}\ opc2\ (rf.src1)\ imm$
ld *dest src1 offset*	$pc := pc + 4 \quad rf.dest := mem.(rf.src1 + offset)$
st *src1 dest offset*	$pc := pc + 4 \quad mem.(rf.dest + offset) := rf.src1$
opc3 reg offset	$pc := \begin{cases} target & \text{if } taken \\ pc+4 & \text{otherwise} \end{cases}$, where $target = get\_target\ pc\ offset$ $taken = get\_taken\ (get\_test\ opc3)\ (rf.reg)$

Fig. 2. *ISA* instruction classes (left column) and corresponding transitions. The variables *dest*, *src1*, *src2*, *reg* have type Reg, and *imm*, *offset* have type Word. For the three opcodes, we have $opc1 \in \{\mathsf{add}, \mathsf{sub}, \mathsf{mult}\}$, $opc2 \in \{\mathsf{addi}, \mathsf{subi}, \mathsf{multi}\}$, $opc3 \in \{\mathsf{beqz}, \mathsf{bnez}, \mathsf{j}\}$.

ISA is a deterministic transition system with system variables $pc : \mathsf{IAddr}$, $rf : \mathsf{RF}$, $mem : \mathsf{MEM}$, $imem : \mathsf{IMEM}$. We assume the types Reg and Word of registers and machine words, so that *rf* can be viewed as a Reg-indexed array with Word values. Similarly, *mem* can be viewed as a Word-indexed array with values in Word, while *imem* is an IAddr-indexed array with values in the type Instr of instructions. Instructions fall into five classes that are identified by the predicates *alu_reg*, *alu_imm*, *ld*, *st*, *branch*. The form of an instruction of each class is given in Figure 2. The figure also shows the *ISA* transitions—the change-of-state equations defined separately for each instruction class.

2.2 State

Parcels are records with the following fields:

$instr : \mathsf{Instr}_\perp$	$my\_pc : \mathsf{IAddr}_\perp$	$dest, src1, src2 : \mathsf{Reg}_\perp$
$imm : \mathsf{Word}_\perp$	$opc : \mathsf{Opcode}_\perp$	$data1, data2, res, mem\_addr : \mathsf{Word}_\perp$
$tkn : \mathsf{bool}_\perp$	$next\_pc : \mathsf{IAddr}_\perp$	$wb : \{\perp, \top\}$ $pc\_upd : \{\perp, \mathtt{s}, \mathtt{m}, \top\}$

The subscript $\perp$ to a type indicates the addition of the element $\perp$ ("undefined") to the type. The *empty_parcel* has all fields equal to $\perp$. The field *wb* indicates whether the parcel has written back to the register file (for arithmetic parcels and loads) or to the memory (for stores). Similiarly, *pc_upd* indicates whether the parcel has caused the update of *pc*. The additional values $\mathtt{s}$ and $\mathtt{m}$ are to record that the parcel has updated *pc* $\mathtt{s}$peculatively and that it $\mathtt{m}$ispredicted.

In addition to the architected state components *pc*, *rf*, *mem*, *imem*, the state of *MOP* contains integers *head* and *tail*, and a queue of parcels *q*. The queue is represented by an integer-indexed array with *head* and *tail* defining its front and back ends. We write *idx j* as an abbreviation for the predicate $head \le j \le tail$, saying that *j* is a valid index in *q*. The j^{th} parcel in *q* is denoted *q.j*.

2.3 Transitions

The transitions of *MOP* are defined by the rules given in Figure 3. Each rule is a guarded parallel assignment, where DEFN contains local definitions, GUARD is the set of predicates defining the rule's domain, and ASSIGN are the assignments made when the rule fires. Some rules contain additional predicates and functions, defined next.

The rule **decode** requires the predicate $\mathit{decoded}\ p \equiv p.\mathit{opc} \neq \bot$ and the function *decode* that updates the parcel field *opc* and some of the fields *dest*, *src1*, *src2*, *imm*. This update depends on the instruction class of $p.\mathit{instr}$, as in the following table.

instruction	*opc*	*dest*	*src1*	*src2*	*imm*
ADD R1 R2 R3	add	R1	R2	R3	$\bot$
ADDI R1 R2 17	addi	R1	R2	$\bot$	17
LD R1 R2 17	ld	R1	R2	$\bot$	17
ST R1 R2 17	st	$\bot$	R1	R2	17
BEQZ R1 17	beqz	$\bot$	R1	$\bot$	17
J 17	j	$\bot$	$\bot$	$\bot$	17

To specify how a given parcel should receive its *data1* and *data2*—from the register file or by forwarding—we use the predicates $\mathit{no\_mrw}\ r\ j \equiv (S = \emptyset)$ and $\mathit{mrw}\ r\ j\ k \equiv (S \neq \emptyset \wedge \max S = k)$, where $S = \{k \mid k < j \wedge \mathit{idx}\ k \wedge q.k.\mathit{dest} = r\}$. The former checks whether the parcel $q.j$ needs forwarding for a given register r and the latter gives the position k of the forwarding parcel (*mrw* = "*m*ost *r*ecent *w*rite").

The rule **write_back** allows parcels to write back to the register file out-of-order. The parcel $q.j$ can write back assuming (1) it is not mispredicted, and (2) there are no parcels in front of it that write to the same register or that have not fetched an operand from that register. These conditions are expressed by predicates $\mathit{fit}\ j \equiv \bigwedge_{\mathit{head}<j'\leq j} \mathit{fit\_at}\ j'$ and $\mathit{valid\_data\_upto}\ j \equiv \bigwedge_{\mathit{head}\leq j'\leq j} \mathit{valid\_data}\ j'$, where

$$\mathit{fit\_at}\ j \equiv q.j.\mathit{my\_pc} = q.(j-1).\mathit{next\_pc} \neq \bot$$
$$\mathit{valid\_data}\ j \equiv q.j.\mathit{data1} \neq \bot \wedge (\mathit{alu\_reg}\ q.j \Rightarrow q.j.\mathit{data2} \neq \bot)$$

Memory access rules (**load** and **store**) enforce in-order execution of loads and stores. The existence and the location of the most recent memory access parcel are described by predicates *mrma* ("*m*ost *r*ecent *m*emory *a*ccess") and *no_mrma*, analogous to *mrw* and *no_mrw* above: one has $\mathit{mrma}\ j\ k$ when k is the largest valid index such that $k < j$ and $q.k$ is a load or store; and one has $\mathit{no\_mrma}\ j$ when no such number k exists. The completion of a parcel's memory access is formulated by

$$\mathit{ma\_complete}\ p \equiv (\mathit{load}\ p \wedge p.\mathit{res} \neq \bot) \vee (\mathit{store}\ p \wedge p.\mathit{wb} = \top)$$

The next four rules in Figure 3 (with **branch** in their name) cover the computation of the next pc value of a parcel, and the related test of whether the branch is taken and (if so) the computation of the target address. The functions *get_taken* and *get_target* are the same ones used by the *ISA*.

The rules **pc_update** and **speculate** govern the program counter updating. The first is based on the *next_pc* value of the last parcel and implements the regular *ISA* flow. The second implements practically unconstrained speculative updating of the program counter, specified by an arbitrary branch_predict function.

DEFN	$i = imem.pc$	**fetch**
GUARD	$length = 0 \vee q.tail.pc\_upd \in \{\mathbb{s}, \top\}$	
ASSIGN	$q.(tail + 1) := empty\_parcel[instr \mapsto i, my\_pc \mapsto pc] \quad tail := tail + 1$	
DEFN	$p = q.j$	**decode** j
GUARD	$idx\ j \quad \neg(decoded\ p)$	
ASSIGN	$p := decode\ p$	
DEFN	$p = q.j$	**data1_rf** j
GUARD	$idx\ j \quad decoded\ p \quad p.src1 \neq \bot \quad p.data1 = \bot \quad no\_mrw\ (p.src1)\ j$	
ASSIGN	$p.data1 := rf.(p.src1)$	
DEFN	$p = q.j \quad \bar{p} = q.k,$ where $mrw\ (p.src1)\ j\ k$	**data1_forward** j
GUARD	$idx\ j \quad decoded\ p \quad p.src1 \neq \bot \quad \bar{p}.res \neq \bot \quad p.data1 = \bot$	
ASSIGN	$p.data1 := \bar{p}.res$	
DEFN	$p = q.j \quad d = p.data1 \quad d' = \begin{cases} p.data2 & \text{if } alu\_reg\ p \\ p.imm & \text{if } alu\_imm\ p \end{cases}$	**result** j
GUARD	$\begin{bmatrix} idx\ j \quad p.data1 \neq \bot \quad p.res = \bot \\ (alu\_reg\ p \wedge p.data2 \neq \bot) \vee alu\_imm\ p \end{bmatrix}$	
ASSIGN	$p.res := alu\ p.opc\ \ d\ d'$	
DEFN	$p = q.j \quad d = p.data1 \quad d' = p.data2$	**mem_addr** j
GUARD	$idx\ j \quad p.mem\_addr = \bot \quad (ld\ p \wedge d \neq \bot) \vee (st\ p \wedge d' \neq \bot)$	
ASSIGN	$p.mem\_addr := \begin{cases} d + p.imm & \text{if } ld\ p \\ d' + p.imm & \text{if } st\ p \end{cases}$	
DEFN	$p = q.j$	**write_back** j
GUARD	$\begin{bmatrix} idx\ j \quad alu\_reg\ p \vee alu\_imm\ p \vee ld\ p \quad fit\ j \quad valid\_data\_upto\ j \\ no\_mrw\ (p.dest)\ j \quad p.res \neq \bot \quad p.wb = \bot \end{bmatrix}$	
ASSIGN	$rf.(p.dest) := p.res \quad p.wb := \top$	
DEFN	$p = q.j$	**load** j
GUARD	$\begin{bmatrix} idx\ j \quad ld\ p \quad p.mem\_addr \neq \bot \quad p.res = \bot \\ no\_mrma\ j \vee (mrma\ j\ k \wedge ma\_complete\ q.k) \end{bmatrix}$	
ASSIGN	$p.res := mem.(p.mem\_addr)$	
DEFN	$p = q.j$	**store** j
GUARD	$\begin{bmatrix} idx\ j \quad st\ p \quad p.mem\_addr \neq \bot \quad p.data1 \neq \bot \quad p.wb = \bot \quad fit\ j \\ no\_mrma\ j \vee (mrma\ j\ k \wedge ma\_complete\ q.k) \end{bmatrix}$	
ASSIGN	$mem.(p.mem\_addr) := p.data1 \quad p.wb := \top$	
DEFN	$p = q.j$	**branch_target** j
GUARD	$idx\ j \quad branch\ p \quad decoded\ p \quad p.res = \bot$	
ASSIGN	$p.res := get\_target\ (p.my\_pc)\ (p.imm)$	
DEFN	$p = q.j \quad t = get\_test\ (p.opc)$	**branch_taken** j
GUARD	$idx\ j \quad branch\ p \quad decoded\ p \quad p.data1 \neq \bot \quad p.tkn = \bot$	
ASSIGN	$p.tkn := get\_taken\ t\ (p.data1)$	
DEFN	$p = q.j$	**next_pc_branch** j
GUARD	$idx\ j \quad branch\ p \quad p.tkn \neq \bot \quad p.res \neq \bot \quad p.next\_pc = \bot$	
ASSIGN	$p.next\_pc := \begin{cases} p.res & \text{if } p.tkn \\ (p.my\_pc) + 4 & \text{otherwise} \end{cases}$	
DEFN	$p = q.j$	**next_pc_nonbranch** j
GUARD	$idx\ j \quad \neg(branch\ p) \quad decoded\ p \quad p.next\_pc = \bot$	
ASSIGN	$p.next\_pc := (p.my\_pc) + 4$	
DEFN	$p = q.tail$	**pc_update**
GUARD	$length > 0 \quad decoded\ p \quad p.next\_pc \neq \bot \quad p.pc\_upd \neq \top$	
ASSIGN	$pc := p.next\_pc \quad p.pc\_upd := \top$	
DEFN	$p = q.tail$	**speculate**
GUARD	$length > 0 \quad decoded\ p \quad branch\ p \quad p.pc\_upd = \bot \quad p.next\_pc = \bot$	
ASSIGN	$pc :=$ branch_predict $p.my\_pc \quad p.pc\_upd := \mathbb{s}$	
DEFN	$p = q.j$	**prediction_ok** j
GUARD	$idx\ j \quad idx\ (j + 1) \quad p.pc\_upd = \mathbb{s} \quad fit\_at\ (j + 1)$	
ASSIGN	$p.pc\_upd := \top$	
DEFN	$p = q.j$	**squash** j
GUARD	$idx\ j \quad idx\ (j + 1) \quad p.pc\_upd = \mathbb{s} \quad \neg(fit\_at\ (j + 1)) \quad p.next\_pc \neq \bot$	
ASSIGN	$tail := j \quad p.pc\_upd := \mathbb{m}$	
DEFN		**retire**
GUARD	$length > 0 \quad complete\ (q.head)$	
ASSIGN	$head := head + 1$	

Fig. 3. *MOP* transitions. The rules **data2_rf** and **data2_forward** are analogous to **data1_rf** and **data1_forward**, and are not shown.

Note that the status of a speculating branch changes when its *next_pc* value is computed; if the prediction is correct (matches *my_pc* of the next parcel), the change is modeled by rule **prediction_ok**. And if the *next_pc* value turns out wrong, rule **squash** becomes enabled, effecting removal of all parcels in the shadow of the mispredicted branch.

Rule **retire** fires only for parcels that have completed their expected modification of the architected state. The predicate *complete* p is defined by $(p.wb = \top) \wedge (p.pc\_upd = \top)$ for non-branches, and by $p.pc\_upd = \top$ for branches.

3 *MOP* Correctness

We call *MOP* states with empty queues *flushed* and consider them the initial states of the *MOP* transition system. The map $\gamma\colon s \longmapsto \langle s, empty\_queue\rangle$ establishes a bijection from *ISA* states to flushed *MOP* states.

Note that *MOP* simulates *ISA*: if s and s' are two consecutive *ISA* states, then there exists a sequence of *MOP* transitions that leads from $\gamma(s)$ to $\gamma(s')$. The sequence begins with **fetch** and proceeds depending on the class of the instruction that was fetched, keeping the queue size equal to one until the last transition **retire**. One can prove with little effort that a requisite sequence from $\gamma(s)$ to $\gamma(s')$ can always be found within the set described by the strategy

$$\begin{aligned}&\mathbf{fetch}\,;\,\mathbf{decode}\,;\,(\mathbf{data1\_rf} \,[\!]\, (\mathbf{data1\_rf}\,;\,\mathbf{data2\_rf}))\,;\\ &(\mathbf{result} \,[\!]\, \mathbf{mem\_addr} \,[\!]\, (\mathbf{branch\_taken}\,;\,\mathbf{branch\_target}))\,;\,[\mathbf{load} \,[\!]\, \mathbf{store}]\,;\\ &(\mathbf{next\_pc\_branch} \,[\!]\, \mathbf{next\_pc\_nonbranch})\,;\,\mathbf{pc\_update}\,;\,\mathbf{retire}\end{aligned}$$

For the proof that *ISA* simulates *MOP* (Theorem 3.4 below), we use the approach introduced by Shen and Arvind [15].

A *MOP invariant* is a property that holds for all states reachable from initial (flushed) states. Local confluence is *MOP*'s fundamental invariant.

Theorem 3.1 *Restricted to reachable states, MOP is locally confluent.*

We omit the proof of Theorem 3.1. Note, however, that proof of local confluence breaks down into lemmas—one for each pair of rules. For *MOP*, most of the cases are resolved by rule commutation: if $m_1 \xleftarrow{\rho_1} m \xrightarrow{\rho_2} m_2$ (i.e., ρ_i applies to the state m and leads from it to m_i), then $m_1 \xrightarrow{\rho_2} m' \xleftarrow{\rho_1} m_2$, for some m'. For the sake of illustration, we show in Figure 4 three examples when local confluence requires non-trivial resolution. Diagrams 1 and 2 show two ways of resolving the

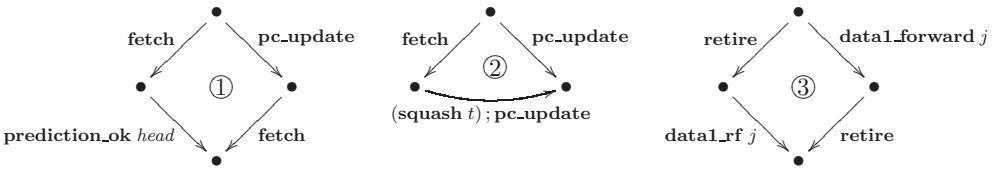


Fig. 4. Example non-trivial cases of local confluence

confluence of the rule pair (**fetch**, **pc_update**). Note that both rules are enabled

only when $q.tail.pc\_upd = \mathtt{s}$. Thus, the parcel $q.tail$ must be a branch, and the fetch is speculative. Diagram 1 applies when the speculation goes wrong, Diagram 2 when the fetched parcel is correct. (In Diagram 2, t is the index of the branch at the tail of the original queue.) Diagram 3 shows local confluence for the pair $(\mathbf{retire}, \mathbf{data1\_forward}\, j)$ when $mrw\, j\, (q.j.src1)\, head$ holds.

The second fundamental property of *MOP* is related to termination. Even though *MOP* is not terminating (of course), every infinite run must have an infinite number of **fetch**es:

Lemma 3.2 *Without the rule* **fetch**, *MOP (on reachable states) is terminating and locally confluent.*

Proof. Every *MOP* rule except **fetch** either reduces the size of the queue, or makes measurable progress in at least one of the fields of one parcel, while keeping all other fields the same. Measurable progress means going from $\bot$ to a non-$\bot$ value, or, in the case of the *pc_upd* field, going up in the ordering $\bot \prec \mathtt{s} \prec \mathtt{m} \prec \top$. This finishes the proof of termination. Local confluence of *MOP* without **fetch** follows from a simple analysis of the (omitted) proof of Theorem 3.1. □

Let us say that a *MOP* state is *irreducible* if none of the rules, except possibly **fetch**, applies to it. It follows from Lemma 3.2, together with Newman's Lemma [3], that for every reachable state m there exists a unique irreducible state which can be reached from m using non-fetch rules. This state will be denoted $|m|$.

Lemma 3.3 *For every reachable state m, the irreducible state $|m|$ is flushed.*

Proof. Suppose the queue of $|m|$ is not empty and let p be its head parcel. We need to consider separately the cases defined by the instruction class of p. All cases being similar, we will give a proof only for one: when p is a conditional branch. Since **decode** does not apply to it, p must be fully decoded. Since **data1_rf** does not apply to p, we must have $p.data \neq \bot$ (other conditions in the guard of **data1_rf** are true). Now, since **branch_taken** and **branch_target** do not apply, we can conclude that $p.res \neq \bot$ and $p.tkn \neq \bot$. This, together with the fact that **next_pc_branch** does not apply, implies $p.next\_pc \neq \bot$. Now, if $p.pc\_upd = \top$, then **retire** would apply. Thus, we must have $p.pc\_upd \neq \top$. Since **pc_update** does not apply, we must have $head \neq tail$, so the queue has length at least 2. If $p.pc\_upd = \mathtt{s}$, then either **squash** or **prediction_ok** would apply to the parcel p. Thus, $p.pc\_upd$ is equal to $\bot$ or $\mathtt{m}$, and this contradicts the (easily checked) invariant saying that a parcel with $p.pc\_upd$ equal to $\bot$ or $\mathtt{m}$ must be at the tail of the queue. □

Define $\alpha(m)$ to be the *ISA* component of the flushed state $|m|$. Recall now the function γ defined at the beginning of this section. The functions γ and α map *ISA* states to *MOP* states and the other way around. Clearly, $\alpha(\gamma(s)) = s$.

The function α is analogous to the pipeline flushing functions of Burch-Dill [5]. Indeed, we can prove that *MOP* satisfies the fundamental Burch-Dill correctness property with respect to this flushing function.

Theorem 3.4 *Suppose a MOP transition leads from m to m', and m is reachable. Then $\alpha(m') = isa\_step\ (\alpha(m))$ or $\alpha(m') = \alpha(m)$.*

Proof. We can assume the transition $m \longrightarrow m'$ is a **fetch**; otherwise, we clearly have $|m| = |m'|$, and so $\alpha(m) = \alpha(m')$. The proof is by induction on the minimum-length k of a chain of (non-fetch) transitions from m to $|m|$. If $k = 0$, then m is flushed, so $m = \gamma(s)$ for some *ISA* state s. By the discussion at the beginning of Section 3, the fetch transition $m \longrightarrow m'$ is the first in a sequence that, without using any further fetches, leads from $\gamma(s)$ to $\gamma(s')$, where $s' = \mathit{isa\_step}\ s$. It follows that $|m'| = |\gamma(s')|$, so $\alpha(m') = \alpha(\gamma(s')) = s'$, as required.

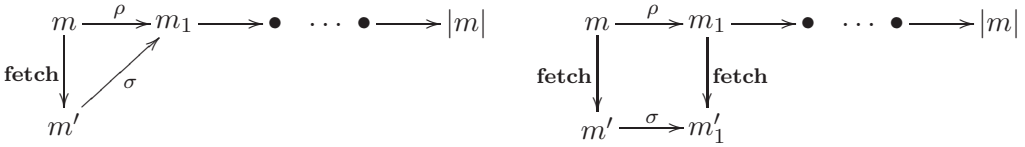


Fig. 5. Two cases for the inductive step in the proof of Theorem 3.4

Assume now $k > 0$ and let $m \xrightarrow{\rho} m_1$ be the first transition in a minimum-length chain from m to $|m|$. Analyzing the proof of Theorem 3.1, one can see that local confluence in the case of the rule pair $(\mathbf{fetch}, \rho)$ can be resolved in one of the two ways shown in Figure 5, where σ has no occurrences of **fetch**. In the first case, we have $\alpha(m') = \alpha(m_1)$, and in the second case we have $\alpha(m') = \alpha(m'_1)$, where m'_1 is as in Figure 5. In the first case, we have $\alpha(m') = \alpha(m_1) = \alpha(m)$. In the second case, the proof follows from $\alpha(m) = \alpha(m_1)$, $\alpha(m') = \alpha(m'_1)$, and the induction hypothesis: $\alpha(m'_1) = \alpha(m_1)$ or $\alpha(m'_1) = \mathit{isa\_step}(\alpha(m'_1))$. □

3.1 Stuttering Equivalence of Bounded MOP and ISA

We begin with a set of definitions.

Two transition systems $\rightarrow_1$ and $\rightarrow_2$ defined on sets S_1, S_2 respectively are said to be *stuttering equivalent* when there exists a relation $R \subset S_1 \times S_2$ that is a *stuttering bisimulation*. This is to say that both R and its inverse R^{-1} are *stuttering simulations*, where the last concept is defined as follows.

Definition 3.5 Let R and the two transition systems be as in the previous paragraph. We say that execution sequences

$$\sigma_1 :\ a_1 \rightarrow_1 a_2 \rightarrow_1 a_3 \rightarrow_1 \cdots \qquad \sigma_2 :\ b_1 \rightarrow_2 b_2 \rightarrow_2 b_3 \rightarrow_2 \cdots$$

are *R-matching* if there exist strictly increasing functions $f, g : \{1, 2, 3, \ldots\} \rightarrow \{1, 2, 3, \ldots\}$ such that for every k, i, j one has

$$f(k) \leq i < f(k+1)\ \wedge\ g(k) \leq j < g(k+1)\ \Rightarrow\ R(a_i, b_j).$$

We say that R is a *stuttering simulation from S_1 to S_2* if for every pair $(a_1, b_1) \in R$ and every execution sequence σ_1 that begins with a_1, there exists an R-matching sequence σ_2 that begins with b_1.

Systems that are stuttering equivalent satisfy the same properties in the temporal logic without the next-state operator; see [13] and references therein.

Let MOP^k $(k \geq 1)$ be the restriction of MOP on the subset of its states for which the queue has length at most k.

Theorem 3.6 *The relation* $R = \{(s, m) \mid s = \alpha(m)\}$ *is a stuttering bisimulation between* MOP^k *and ISA.*

We need three lemmas about occurrences of **squash** in chains of *MOP* transitions. The first is a refinement of an argument used in the proof of Theorem 3.4.

Lemma 3.7 *Let* m *and* m' *be reachable MOP states such that* $m \xrightarrow{\textbf{fetch}} m'$*. Let* σ *be a sequence of rules that lead from* m *to* $|m|$*. If* σ *contains no occurrence of* **squash***, then* $\alpha(m') = \mathit{isa\_step}(\alpha(m))$*.*

Proof. It is easy to see that σ applies to m' as well. Moreover, one can prove that **fetch** commutes with all transitions of σ, i.e. $\textbf{fetch}\,;\,\sigma =_m \sigma\,;\,\textbf{fetch}$. (See Figure 5, diagram on the right.) Thus,

$$\begin{aligned}\alpha(m') &= \alpha(m'\,\sigma) = \alpha((m\,\textbf{fetch})\,\sigma) = \alpha(m\,(\textbf{fetch}\,;\,\sigma)) = \alpha(m\,(\sigma\,;\,\textbf{fetch}))\\ &= \alpha((m\,\sigma)\,\textbf{fetch}) = \alpha(|m|\,\textbf{fetch}) = \mathit{isa\_step}(\alpha(|m|)) = \mathit{isa\_step}(\alpha(m))\end{aligned}$$

□

Lemma 3.8 *Suppose* σ *is a chain of MOP rules that applies to some MOP state. If* **squash** j *occurs twice in* σ*, then between these two occurrences there must be an occurrence of* squash j' *for some* $j' < j$*.*

Proof. Notice a general fact that holds for an arbitrary sequence σ of *MOP* rules, an arbitrary *MOP* state m and an arbitrary index j: if σ applies to m and **squash** j' does not occur in σ for any $j' < j$, then on the way from m to $m\,\sigma$ via σ every field of the parcel $m.q.j$ either makes progress or stays the same.

Now, write

$$\sigma = \sigma_1\,;\,\textbf{squash}\,j\,;\,\sigma_2\,;\,\textbf{squash}\,j\,;\,\sigma_3,$$

where σ and j are as given in the lemma. Let $m' = m\,(\sigma_1\,;\,\textbf{squash}\,j)$ and $m'' = m\,\sigma_2$. Then $m'.q.j.\mathit{pc\_upd} = \mathtt{m}$ and $m''.q.j.\mathit{pc\_upd} = \mathtt{s}$. Since $\mathtt{s} \prec \mathtt{m}$, the general fact above implies that for some $j' < j$, **squash** j' must occur in σ_2. □

Let us say that j is *safe for* m if there is no chain of rules that applies to m and contains **squash** j' for some $j' \le j$.

Lemma 3.9 *Suppose* σ *is a chain of rules that applies to* m *and does not involve* **squash** j' *for any* $j' \le j$*. Then: if* j *is safe for* $m\,\sigma$*, it is safe for* m *too.*

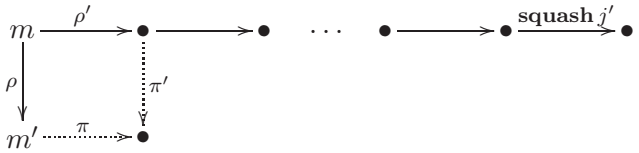


Fig. 6. Illustration for proof of Lemma 3.9.

Proof (Sketch) It is no loss of generality to assume that σ is a single rule ρ distinct from **squash** j' for any $j' \le j$. Assuming that there is a chain μ of rules that starts at m and includes **squash** j' for some $j' \le j$, we can prove that there exists a chain μ' that starts at $m' = m\,\rho$ and also includes **squash** j' for some $j' \le j$. See Figure 6, where μ is the chain on the top. The proof is by induction on the number of **squash**

rules in μ and the length of μ. It depends on the case analysis over pairs (ρ, ρ') in the confluence proof (Theorem 3.1) and the form of π, π'. □

Proof of Theorem 3.6. Part 1: From ISA to MOP^k. Given an *ISA* execution sequence

$$\sigma\colon s_1 \longrightarrow s_2 \longrightarrow s_3 \longrightarrow \cdots$$

and m_1 such that $s_1 = \alpha(m_1)$, we have a chain of *MOP* transitions $m_1 \longrightarrow \cdots \longrightarrow |m_1| = \gamma(s_1)$. Furthermore, for every i there is a path in *MOP*^{1} from $\gamma(s_i)$ to $\gamma(s_{i+1})$, as explained at the beginning of Section 3. Splicing these paths together gives a sequence

$$\mu\colon m_1 \longrightarrow^* \gamma(s_1) \longrightarrow^* \gamma(s_2) \longrightarrow^* \cdots$$

of *MOP*^{1} transitions. In this sequence, $\alpha(m) = s_1$ holds for all states m on the path $m_1 \longrightarrow^* \gamma(s_1)$. Also, for every i, $\alpha(m) = s_i$ holds for all states m on the path $\gamma(s_{i-1}) \longrightarrow^* \gamma(s_i)$, except the first. Thus, μ R-matches σ.

Part 2: From MOP^k to ISA. Let

$$\mu\colon m_1 \longrightarrow m_2 \longrightarrow m_3 \longrightarrow \cdots$$

be an infinite chain of *MOP*^{k}-transitions. By Theorem 3.4, in the chain

$$\sigma\colon \alpha(m_1) \longrightarrow \alpha(m_2) \longrightarrow \alpha(m_3) \longrightarrow \cdots$$

we have for every i that either $\alpha(m_i) \longrightarrow \alpha(m_{i+1})$ is an *ISA* transition, or $\alpha(m_{i+1}) = \alpha(m_i)$ holds. We need to show that the first case occurs infinitely often.

Since *MOP* without **fetch** is terminating, the rule **fetch** must be used in infinitely many transitions $m_i \longrightarrow m_{i+1}$ in μ. We claim that **retire** must also be used in infinitely many transitions in μ. Suppose the claim is not true; then there exists h and i_0 such that $m_i.head = h$ and $m_i.tail \leq h + k$, for all $i \geq i_0$. Since **retire** and **squash** are the only rules that decrease the queue size, to compensate for the infinitely many **fetch**'s, there must be infinitely many **squash** transitions in μ. More precisely, it follows that for some j such that $h \leq j \leq h + k$, there are infinitely many transitions **squash** j in μ. However, Lemma 3.8 easily implies that any chain of *MOP* rules applicable to a *MOP* state may contain only finitely many occurrences of **squash** j for any particular j. The contradiction finishes the proof that μ contains infinitely many occurrences of **retire**.

Now we know that the (non-decreasing) sequence $m_1.head, m_2.head, m_3.head, \ldots$ is unbounded. Consequently, for any given l, there are only finitely many i such that $m_i.tail \leq l$. (Note that the sequence $m_1.tail, m_2.tail, m_3.tail, \ldots$ is unbounded, but not necessarily monotonic because of uses of **squash**.) Let then $\hat{l}$ denote the largest i such that $m_i.tail = l$. Clearly, the transition $m_{\hat{l}} \longrightarrow m_{\hat{l}+1}$ must be a **fetch** for every l.

Fix l. Let l' be any index such that $m_{l'}.head > \hat{l}$ and let σ denote the subchain of μ leading from $m_{\hat{l}}$ to $m_{l'}$. Note that **squash** j for $j \leq l$ cannot occur in σ because that would violate the maximality condition on $\hat{l}$. Note also that $m_{l'}$ is safe for any j such that $j \leq l$, as a consequence of $l < m_{l'}.head$. By Lemma 3.9, $m_{\hat{l}}$ is safe for

any $j \leq l = m_{\hat{l}}.tail$. This implies that no **squash** rule can occur in a (**fetch**-free) reduction sequence $m_{\hat{l}} \longrightarrow^* |m_{\hat{l}}|$. By Lemma 3.7, we have $\alpha(m_{\hat{l}+1}) = \alpha(m_{\hat{l}}\,\mathbf{fetch}) = \mathit{isa\_step}(\alpha(m_{\hat{l}}))$.

4 Simulating Microarchitectures in *MOP*

Suppose *MA* is a microarchitecture purportedly implementing the *ISA*. We will say that a state-to-state map β from *MA* to *MOP* is a *MOP-simulation* if for every *MA* transition $s \longrightarrow_{MA} s'$, the state $\beta(s')$ is reachable in *MOP* from $\beta(s)$. Existence of a *MOP*-simulation proves (the safety part of) the correctness of *MA*. Indeed, for every execution sequence $s_1 \longrightarrow_{MA} s_2 \longrightarrow_{MA} \ldots$, we have $\beta(s_1) \longrightarrow^+_{MOP} \beta(s_2) \longrightarrow^+_{MOP} \ldots$, and then by Theorem 3.4, $\alpha(\beta(s_1)) \longrightarrow^*_{ISA} \alpha(\beta(s_2)) \longrightarrow^*_{ISA} \ldots$, demonstrating the crucial simulation relation between *MA* and *ISA*.

For a given *MA*, the *MOP*-simulation function β should be easy to guess. The difficult part is to verify that it satisfies the required property: the existence of a chain of *MOP* transitions $\beta(s) \longrightarrow^+_{MOP} \beta(s')$ for each transition $s \longrightarrow_{MA} s'$. Somewhat simplistically, this verification task can be partitioned as follows.

Suppose *MA*'s state variables are $v_1, \ldots, v_n$. (Among them are the *ISA* state variables, of course.) The *MA* transition function

$$s = \langle v_1, \ldots, v_n \rangle \longmapsto s' = \langle v'_1, \ldots, v'_n \rangle$$

is given by n functions $\mathit{next\_}v_i$ such that $v'_i = \mathit{next\_}v_i\,\langle v_1, \ldots, v_n \rangle$. The n-step sequence $s = s_0 \rightsquigarrow s_1 \rightsquigarrow \ldots \rightsquigarrow s_{n-1} \rightsquigarrow s_n = s'$ where $s_i = \langle v'_1, \ldots, v'_i, v_{i+1}, \ldots, v_n \rangle$ conveniently serializes the parallel computation that *MA* does when it makes a transition from s to s'. These n steps are not *MA* transitions themselves since the intermediate s_i need not be legitimate *MA* states at all. However, it is reasonable to expect that the progress described by this sequence is reflected in *MOP* by actual transitions:

$$\beta(s) = m_0 \longrightarrow^*_{MOP} m_1 \longrightarrow^*_{MOP} \ldots \longrightarrow^*_{MOP} m_n = \beta(s'). \tag{1}$$

Defining the intermediate *MOP* states m_i will usually be straightforward. Once they have been identified, the task of proving that $\beta(s')$ is reachable from $\beta(s)$ is broken down into n tasks of proving that m_{i+1} is reachable from m_i. Effectively, the correctness of the *MA* next-state function is reduced to proving a correctness property for each state component update function $\mathit{next\_}v_i$.

5 Mechanization

Our method is intended to be used with a combination of interactive (or manual) and automated theorem proving. The correctness of the *MOP* system (Theorem 3.4) rests largely on its local confluence (Theorem 3.1), which is naturally and easily split into a large number of cases that can be individually verified by an automated SMT solver. The solver needs decision procedures for uninterpreted functions, a fragment of arithmetic, and common datatypes auch as arrays, records and enumeration types. Once the *MA*-simulation function β of Section 4 has been defined and the

intermediate *MOP* states m_i in the chain (1) identified, it should also be possible to generate the proof of reachability of m_{i+1} from m_i with the aid of the same solver.

We have used manual proof decomposition and CVC Lite to implement the proof procedure just described. Our models for *ISA*, *MOP*, and *MA* are all written in the reflective general-purpose functional language *reFLect* [7]. In this convenient framework we can execute specifications and—through a HOL-like theorem prover on top of *reFLect* or an integrated CVC Lite—formally reason about them at the same time. Local confluence of *MOP* is (to some extent automatically) reduced to about 400 goals, which are individually proved with CVC Lite. For *MA* we used the textbook *DLX* model [9] and proved it is simulated in *MOP* by constructing the chains (1) and verifying them with CVC Lite. This proof is sketched in some detail below.

Mechanization of our method is still in progress. Clean and efficient use of SMT solvers to prove properties of executable high-level models written in another language comes with challenges, some of which were discussed in [8]. For us, particularly exigent is the demand for heuristics for deciding when to expand occurrences of function symbols in goals passed to the SMT solver with the functions' definitions, and when to treat them as uninterpreted.

5.1 Simulating DLX in MOP

To illustrate the the ideas given in Section 5, we use a simple five-stage pipelined processor based on *DLX* [9]. Its states are 7-tuples $s = \langle p_1, p_2, p_3, p_4, pc, rf, mem\rangle$, where the p_i are the pipeline registers and the rest is the architected state. The *DLX* next-state function is briefly explained in Figure 7; for more details, see [9].

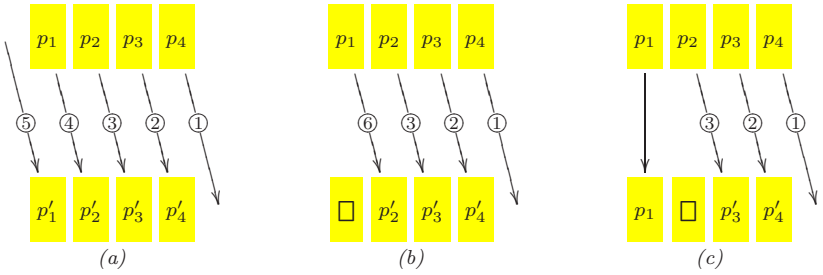
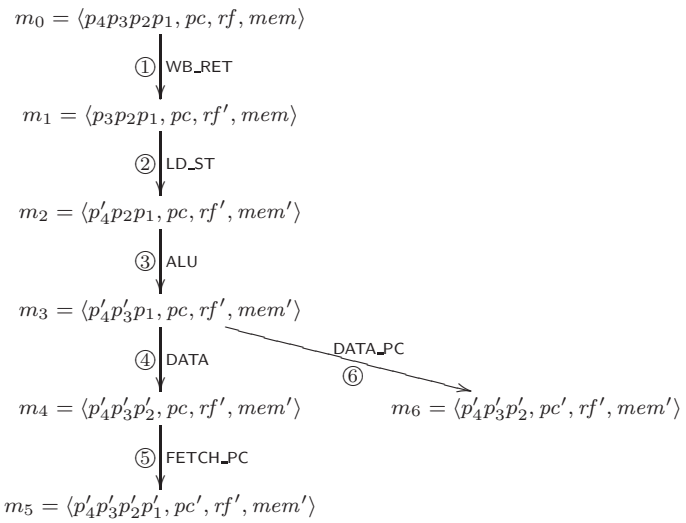


Fig. 7. Dynamics of *DLX*. *(a)* In a regular cycle, the *DLX* step can be seen as a sequence of five actions: (1) parcel p_4 writes back and retires; (2) p_3 performs memory access; (3) alu computes the result of p_2 or the address for its memory access; (4) p_1 receives data from the register file or by forwarding, and if it a branch, its target is computed as well as whether the branch is taken or not; (5) a new parcel p'_1 is fetched and pc is incremented. *(b)* If p_1 is a taken branch, action (4) is modified to include updating the pc with the computed target (becoming action (6) in the picture), and no parcels are fetched. *(c)* The machine stalls one cycle if p_2 is a load and p_1 depends on it.

It is straightforward to associate a *MOP* parcel to each valid (non-bubble) value of a *DLX* pipeline register. Combining the resulting parcels into a *MOP* queue and copying the *ISA* state, we can define the simulation function β that maps *DLX* states to *MOP* states. If s is as above, then $\beta(s)$ can be written as $\langle \tilde{p_4}\tilde{p_3}\tilde{p_2}\tilde{p_1}, pc, rf, mem\rangle$, where $\tilde{p_4}\tilde{p_3}\tilde{p_2}\tilde{p_1}$ denotes the *MOP* queue corresponding to the contents of the pipeline registers (p_1, p_2, p_3, p_4), with the proviso that $\tilde{p_i}$ is absent if p_i is a bubble. Figure 8 sketches the proof that β is a legitimate simulation function. (To avoid clutter, we dropped the tildes from all *MOP* parcels.)



$$\mathsf{WB\_RET} = \begin{cases} \textbf{retire} & \text{if } \mathit{store}\ p_4 \vee \mathit{branch}\ p_4 \\ \textbf{write\_back}_1\,;\,\textbf{retire} & \text{if } \mathit{alu\_reg}\ p_4 \vee \mathit{alu\_immed}\ p_4 \vee \mathit{load}\ p_4 \end{cases}$$

$$\mathsf{LD\_ST} = \begin{cases} \textbf{load}_1 & \text{if } \mathit{load}\ p_3 \\ \textbf{store}_1 & \text{if } \mathit{store}\ p_3 \end{cases}$$

$$\mathsf{ALU} = \begin{cases} \textbf{result}_2 & \text{if } \mathit{alu\_reg}\ p_2 \vee \mathit{alu\_immed}\ p_2 \\ \textbf{mar}_2 & \text{if } \mathit{load}\ p_2 \vee \mathit{store}\ p_2 \end{cases}$$

$$\mathsf{DATA} = \begin{cases} \textbf{data1}_3\,;\,\textbf{data2}_3 & \text{if } \mathit{alu\_reg}\ p_1 \vee \mathit{store}\ p_1 \\ \textbf{data}_3 & \text{if } \mathit{alu\_immed}\ p_1 \vee \mathit{load}\ p_1 \\ \textbf{branch\_data}_3\,;\,\textbf{next\_pc\_branch}_3 & \text{if } \mathit{branch}\ p_1 \end{cases}$$

$$\mathsf{DATA\_PC} = \textbf{branch\_data}_3\,;\,\textbf{next\_pc\_branch}_3\,;\,\textbf{pc\_update}_3 \quad \text{if } \mathit{branch}\ p_1$$

$$\mathsf{FETCH\_PC} = \begin{cases} \textbf{fetch}\,;\,\textbf{decode}_4\,;\,\textbf{speculate} & \text{if } \mathit{branch}\ p'_1 \\ \textbf{fetch}\,;\,\textbf{decode}_4\,;\,\textbf{next\_pc\_nonbranch}_4\,;\,\textbf{pc\_update}_4 & \text{otherwise} \end{cases}$$

Fig. 8. Simulation of *DLX* in *MOP*. The *MOP* state m_0 is $\beta(s)$, where s is an arbitrary *DLX* state. If s' is the next *DLX* state, then $\beta(s')$ is either m_5, m_6, or m_3, depending whether we are in the case *(a)*, *(b)*, or *(c)* in Figure 7. The figure shows the sequence of *MOP* steps that go from $\beta(s)$ to $\beta(s')$ in all cases.

The "rule" $\textbf{data1}_3$ is either $\textbf{data1\_rf}_3$ or $\textbf{data1\_forward}_3$, depending on whether the parcel p_1 depends on the parcels p_2, p_3 or not (similarly for $\textbf{data2}_3$). The "rule" $\textbf{branch\_data}_3$ is an abbreviation for $\textbf{data}_3\,;\,\textbf{branch\_taken}_3\,;\,\textbf{branch\_target}_3$.

6 Related Work

The idea of flushing a pipeline automatically was introduced in a seminal paper by Burch and Dill [5]. In the original approach, all in-flight instructions in the implementation state are flushed out of the pipeline by inserting *bubbles*—NOPs that do not affect the program counter. Pipelines that use a combination of superscalar execution, out-of-order execution, and variable-latency execution units are too complex to flush directly. In response, researchers have invented a variety of ways, many based on flushing, to relate the implementation pipeline to the specification. We cover here only those approaches that are most closely related to our work. The interested reader is refered to [1] for a relatively complete survey of pipeline verification approaches.

Shen and Arvind [15] were first to prove an example of Burch-Dill correctness using the flushing function defined as the normal form in a confluent system. They model an abstract out-of-order processor and a simple specification machine as *term rewriting systems*. Their implementation model is similar to our intermediate

specification *MOP*, and its Burch-Dill correctness against the specification *ISA* is the main result of [15]. We go a step further by proving stuttering bisimulation. Also, *MOP* is for us only an intermediate model that, in turn, allows us to reason about deterministic and more realistic implementations.

Hosabettu *et al.* [10] devised a method to decompose the Burch-Dill correctness statement into lemmas, one per pipeline stage. This inspired the decomposition we describe in Section 4.

Lahiri and Bryant [12], and Manolios and Srinivasan [13] verified complex microprocessor models using the SMT solver UCLID. Some *consistency invariants* in [12] occur naturally in our confluence proofs as well, but the overall approach is not closely related. The *WEB-refinement* method used in [13] produces proofs of stuttering bisimulation between *ISA* and *MA* that implies liveness. This gave motivation for our Theorem 3.6, but our stuttering bisimulation proof is different.

Skakkebæk *et al.* [16,11] introduce *incremental flushing* and use a non-deterministic intermediate model to prove correctness of a simple out-of-order core with in-order retirement. Like us, they rely on arguments about transaction re-ordering. While incremental flushing must deal with transactions as they are defined for the pipeline, we decompose pipeline transactions into much simpler "atomic" transactions. This facilitates a more general abstraction and should require significantly less manual proof effort for a given pipeline than the incremental flushing approach.

Damm and Pnueli [6] use a non-deterministic specification that generates all program traces that satisfy data dependencies. They use an intermediate abstraction with auxiliary variables to relate the specification and an implementation with out-of-order retirement based on Tomasulo's algorithm. In each step of the specification model, an entire instruction is executed atomically and its result written back. In the *MOP* approach, the execution of each instruction is broken into a sequence of mini-steps in order to relate to a pipelined implementation.

Sawada and Hunt [14] use an intermediate model with an unbounded history table called a *micro-architectural execution trace table*. It contains instruction-specific information similar to that found in the *MOP* queue. Arons [2] follows a similar path, augmenting an implementation model with history variables that record the predicted results of instruction execution. In these approaches, auxiliary state is—like the *MOP* queue—employed to derive and prove invariants about the implementation's relation to the specification. While their auxiliary state is derived directly from the *MA*, *MOP* is largely independent of *MA* and has fine-grained transitions.

7 Conclusion

We have presented an approach for verifying a pipelined system $\mathcal{P}$ against its specification $\mathcal{S}$ by using an intermediate "pipeline mother" system $\mathcal{M}$ that explicates atomic computations occurring in steps of $\mathcal{S}$. For definiteness, we assumed that $\mathcal{P}$ is a microprocessor model and $\mathcal{S}$ is its *ISA*, but the method can potentially be applied to verify pipelined hardware components in general, or in protocol verification. This can all be seen as a refinement of the classical Burch-Dill method, but with the difficult flushing-based simulation pushed to the $\mathcal{M}$ vs. $\mathcal{S}$ level, where it amounts to

proving local confluence of $\mathcal{M}$—a conjunction of easily-stated properties of limited size, readily verifiable by SMT solvers.

As an example, we specified a concrete intermediate model *MOP* for a simple load-store architecture and proved its correctness. We also verified the textbook machine *DLX* against it. However, our *MOP* contains more than is needed for verifying *DLX*: it is designed for simulation of microprocessor models with complex out-of-order execution that cannot be handled by currently-available methods. This will be addressed in future work. Also left for future work are improvements to our methodology (manual decomposition of verification goals into subgoals which we prove with CVC Lite [4]) and performance comparison with other published methods.

References

[1] 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.

[2] T. Arons. Verification of an advanced MIPS-type out-of-order execution algorithm. In *Computer Aided Verification (CAV'04)*, volume 3114 of *LNCS*, pages 414–426, 2004.

[3] F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[4] C. Barrett and S. Berezin. CVC Lite: A new implementation of the cooperating validity checker. In R. Alur and D. A. Peled, editors, *Computer Aided Verification (CAV'04)*, volume 3114 of *LNCS*, pages 515–518, 2004.

[5] J. Burch and D. Dill. Automatic verification of pipelined microprocessor control. In D. L. Dill, editor, *Computer Aided Verification (CAV'94)*, volume 818 of *LNCS*, pages 68–80, 1994.

[6] W. Damm and A. Pnueli. Verifying out-of-order executions. In H. F. Li and D. K. Probst, editors, *Correct Hardware Design and Verification Methods (CHARME'97)*, pages 23–47. Chapman and Hall, 1997.

[7] J. Grundy, T. Melham, and J. O'Leary. A reflective functional language for hardware design and theorem proving. *J. Functional Programming*, 16(2):157–196, 2006.

[8] J. Grundy, T. F. Melham, S. Krstić, and S. McLaughlin. Tool building requirements for an API to first-order solvers. *Electr. Notes Theor. Comput. Sci.*, 144(2):15–26, 2006.

[9] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 1995.

[10] R. Hosabettu, G. Gopalakrishnan, and M. Srivas. Verifying advanced microarchitectures that support speculation and exceptions. In E. A. Emerson and A. P. Sistla, editors, *Computer Aided Verification (CAV'00)*, volume 1855 of *LNCS*, pages 521–537, 2000.

[11] R. B. Jones. *Symbolic Simulation Methods for Industrial Formal Verification*. Kluwer, 2002.

[12] S. K. Lahiri and R. E. Bryant. Deductive verification of advanced out-of-order microprocessors. In W. A. Hunt Jr. and F. Somenzi, editors, *Computer Aided Verification (CAV'03)*, volume 2725 of *LNCS*, pages 341–354, 2003.

[13] P. Manolios and S. K. Srinivasan. A complete compositional reasoning framework for the efficient verification of pipelined machines. In *IEEE/ACM International conference on Computer-aided design (ICCAD'05)*, pages 863–870. IEEE Computer Society, 2005.

[14] J. Sawada and W. 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 *LNCS*, pages 135–146, 1998.

[15] X. Shen and Arvind. Design and verification of speculative processors. In *Workshop on Formal Techniques for Hardware*, Maarstrand, Sweden, June 1998.

[16] J. Skakkebæk, R. Jones, and D. 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 *LNCS*, pages 98–109, 1998.