A Comparison of Two Verification Methods for Speculative Instruction Execution*

Tamarah Arons and Amir Pnueli

The John von Neumann Minerva Center for Verification of Reactive Systems,
Weizmann Institute of Science, Rehovot, Israel
{tamarah,amir}@wisdom.weizmann.ac.il

Abstract. In this paper we describe and compare two methodologies for verifying the correctness of a speculative out-of-order execution system with interrupts. Both methods are deductive (we use PVS) and are based on refinement. The first proof is by direct refinement to a sequential system; the second proof combines refinement with induction over the number of retirement buffer slots.

1 Introduction

Modern out-of-order super-scalar microprocessors use dynamic scheduling to increase the number of instructions executed per cycle. These processors maintain a fixed-size window into the instruction stream, analyzing the instructions in the window to determine which can be executed out of order to improve performance. Branch prediction and register renaming are employed in order to keep the window full, while result-buffering techniques maintain the in-order-execution model required by the architecture.

In this paper we discuss two *refinement*-based proofs of the correctness of such processors. Our model is based on the Tomasulo algorithm in [13,4] and [6], with modifications for in-order-retirement and speculative instruction prediction adapted from [5]. This paper is a continuation of the work on out-of-order execution presented in [4,2] and [10]. We extend the methodology of these papers to deal with exceptions and speculative instruction execution, while also presenting a new, inductive, methodology. Both proofs have been verified using the PVS [9] theorem prover[1].

In the first proof, which we refer to as the *direct* proof, we use a top-down methodology to generate and prove the system invariants needed to prove that our speculative system refines a sequential system. Starting with the final invariant to be proved, this methodology allows the user to systematically generate and prove all other necessary invariants.

In the second proof we combine *refinement* and *induction*. Under the premise that the more similar two systems are the easier it should be to prove refinement between them, we use induction to generate two refinement proofs between

* Research supported in part by a grant from the German-Israel bi-national GIF foundation and a gift from Intel.

[1] The PVS files are available at `http://www.wisdom.weizmann.ac.il/~tamarah`

S. Graf and M. Schwartzbach (Eds.): TACAS/ETAPS 2000, LNCS 1785, pp. 487–502, 2000.

similar systems. Noting that the number of instructions which are in progress in an out-of-order system is limited to the number of retirement buffer slots, we first show that a speculative system which has only one buffer slot (and thus functions sequentially) refines a sequential system, and then, that a system with $B+1$ buffer slots refines one with B slots. The base case thus deals only with the differences in data structures, while in the induction step we focus on the effect of the greater measure of 'out-of-order'ness allowed by one additional buffer slot.

Due to the enormous differences between sequential and speculative systems it is not immediately obvious how the first may refine the second. However, it is easy to anticipate that a system may refine another with one more retirement buffer. Thus, our intuition was that the inductive proof would prove to be simpler than the direct one. However, this proved not to be the case. While the run-time of the direct-proof is somewhat longer than that of the inductive proof, the inductive proof required far more intricate human interaction, taking far more person-time. We believe that not only was the inductive proof more complex in this case, but that using induction will frequently complicate refinement proofs. This evaluation is discussed in the final section of the paper.

While there is a lot of work in the field of out-of-order executions, not much has been published on speculative execution. It is unclear whether, or how, techniques used for out-of-order execution can be applied to speculative instruction execution. Candidate techniques which have been used to verify out-or-order execution include the completion function approach [7], incremental flushing [12], compositional model checking [8], and techniques combining model checking with uninterpreted functions [3].

A speculative system is verified in [11]. This system is more complex than ours, including memory operations, but the proof is specific to one configuration. An intermediate model comprising a table of history variables is used to verify the system in ACL2. Our proofs have the advantage of being independent of the system configuration and of not requiring an intermediate abstraction.

2 Refinement between Systems

Refinement is the comparison of an *abstract system* $S_A = \langle V_A, \Theta_A, \rho_A \rangle$ and a *concrete system* $S_C = \langle V_C, \Theta_C, \rho_C \rangle$ where V is the set of system variables, Θ defines the initial conditions of the systems, and ρ, the transition relation, defines how the system progresses from one state to another. The abstract system serves as a *specification* capturing all the acceptable correct computations of the concrete system. Correctness of the concrete system is established by proving that every computation of S_C corresponds to some computation of S_A.

The correspondence between the two systems is with respect to *observation functions* $\mathcal{O}_A$ and $\mathcal{O}_C$. Intuitively, these are the features of the two systems which are considered significant for the comparison. For example, in instruction execution systems one would expect the register file to be included in the observation functions while internal data structures might not be.

Given a concrete system $S_C = \langle V_C, \Theta_C, \rho_C \rangle$ with observation function $\mathcal{O}_C$, and an abstract system $S_A = \langle V_A, \Theta_A, \rho_A \rangle$ with observation function $\mathcal{O}_A$, such that $V_C \cap V_A = \emptyset$, we define an *superposition system*

$$S_S = \langle V_C \cup V_A, \Theta_C \wedge \Theta_A, \rho_C \wedge \rho_A^* \rangle$$

where $\rho_A^*(V_C, V_C', V_A, V_A')$ may refer to all variables in $V_C \cup V_A$ in their primed and unprimed versions.

The intention of the superposition system S_S is that it emulates the joint behavior of S_C and S_A in a way that allows any previously admissible step of S_C and matches it with an S_A-step. Thus, $\rho_C \wedge \rho_A^*$ should not exclude any possible S_C-step, but may select among the possible S_A-steps one that matches the S_C-step. Intuitively, ρ_A^* is a modification of ρ_A taking as parameters V_C and V_C' in order to choose a ρ_A-successor matching the S_C-step. We further require that the projection of an S_S-computation onto V_A is a legal computation of S_A.

In any superposition system S_S satisfying the above requirements the problem of showing that $S_C \sqsubseteq S_A$ is reduced to the problem of showing that $\mathcal{O}_C = \mathcal{O}_A$ is an invariant of S_S. However, to do so it may be useful, or necessary, to prove a stronger invariant, $\alpha(V_C, V_A)$ of the superposition system.

We formalize this as refinement rule REF:

$$\frac{\begin{array}{l}\textbf{R1. } \alpha \wedge \rho_C \longrightarrow \exists V_A' : \rho_A^* \\ \textbf{R2. } \rho_A^* \longrightarrow \rho_A \\ \textbf{R3. } S_S \models \Box\, \alpha \\ \textbf{R4. } \alpha \longrightarrow \mathcal{O}_C = \mathcal{O}_A\end{array}}{S_C \sqsubseteq S_A}$$

That is, S_A refines S_C if using ρ_A^* a legal (R2) computation of S_S can be generated (R1) such that $\mathcal{O}_C$ always equals $\mathcal{O}_A$ (R3, R4).

3 The Reference Model: System SEQ

In this section we present system SEQ which is to serve as a reference model. System SEQ executes in a strictly sequential manner an input program which may contain branches and instructions generating interrupts. It accepts one parameter, R, the number of registers.

An uninterpreted function, *prog*, from *PC_RANGE* to instructions defines the program to be executed. Each instruction has an *operation*, a *target* and two *source* operands. In addition, a *branch target* field stores the target address of branches. A program counter, *pc*, points to the next instruction in *prog*. A register file *reg* records the current values of each register.

At each step, system SEQ either delays, in which case no change is made in the system, or executes the instruction pointed to by *pc*. If the instruction execution generates an interrupt, the program counter is updated to point to the relevant interrupt handler address. In the case of branches, the branch is evaluated and the program counter updated to the branch target if the branch

is taken. The *do_op* and *do_branch* functions are used to compute the value of the instruction (*do_branch* returns "1" if a branch is to be taken, "0" otherwise). This value is stored in the target register (if any), and the program counter is updated to point to the next instruction.

4 The Out-Of-Order Design: System DES

In this section we briefly describe our algorithm for speculative out-of-order data-driven instruction execution with in-order-retirement. Our definitions are based on the descriptions in [6,4] and [5].

Instructions flow from the instruction queue to the retirement buffer, where they assume their places in the queue for retirement, and the dispatch buffer, where they await availability of their source operands and a free execution unit. Branch instructions are *predicted* at dispatch time and the program counter updated accordingly. Once both operands are available execution of the instruction can be initiated by the appropriate functional unit. As in system SEQ, the instruction value is calculated by the *do_op* and *do_branch* functions. During execution an *internal interrupt* can be generated, in which case a flag is set in the retirement buffer slot. Results are written back to the retirement and dispatch buffers. Once an instruction reaches the head of the retirement queue it is checked for an internal interrupt or branch misprediction before being retired. If an interrupt was generated the program-counter is updated to the appropriate interrupt handler address and the dispatch and retirement buffers are flushed. If no interrupt was generated the system checks branches for mispredictions. Mispredictions result in the program counter being updated to the instruction which should follow the branch, while dispatch and retirement buffers are flushed. Instructions which generated neither interrupts nor incorrect predictions can be retired, updating the register file with the instruction result.

The data structures of system DES are illustrated in Fig. 1. The shaded fields are auxiliary variables which have been added to our model in order to simplify the proofs. Auxiliary variables are only updated and copied from one record to another and thus do not affect the flow of control. The two proofs use different auxiliary variables, the unified set of which are shown in the diagrams for completeness. The *numinst* variable counts the number of instructions retired so far and is used in synchronizing the two sequential and speculative systems.

The functionality of system DES can be divided into three subsystems:

- DISPATCH: This module dispatches instructions in program order.
- EXECUTE: This module executes and writes back instructions.
- RETIRE: This module retires the slot at the head of the retirement buffer.

While only one instruction is dispatched or retired per cycle, module EXECUTE is parameterized by the number of functional units: when this module is invoked, each functional unit in the system may execute and write-back a result. Multiple instructions may be executed and written back in each cycle.

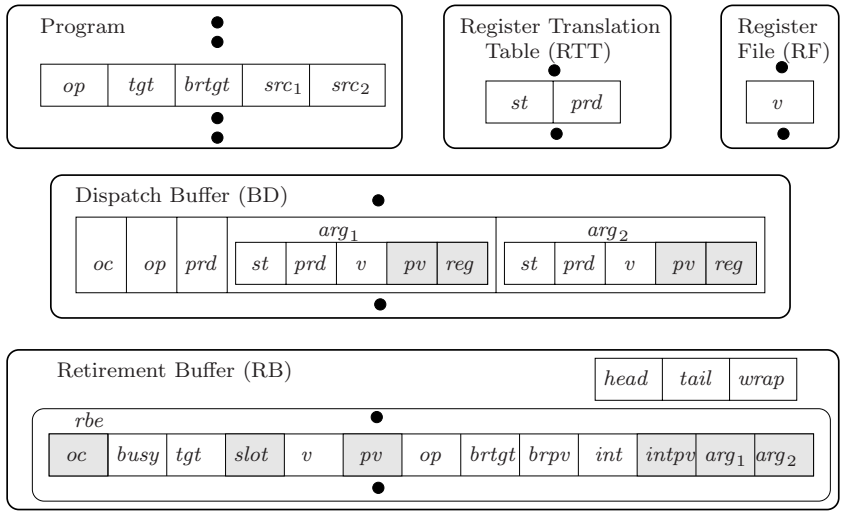

Fig. 1. Data structures for DES

In practice these three subsystems operate concurrently. That is, in the same cycle all three can be invoked simultaneously. Any concurrent execution of the three subsystems is equivalent to a three-step sequential execution of the subsystems in which each subsystem is executed once. We therefore consider each of the three systems separately, ignoring the possible interaction between them.

A note on the retirement buffer The retirement buffer, RB, is the central data structure in the system. It stores instructions in dispatch order until their retirement, ensuring that retirement is in-order. The buffer contains a circular array *rbe* of retirement buffer entries. This array is treated as a queue, with the oldest entry being "popped" off during retirement, while dispatched instructions are "pushed" onto the end of the queue. The pointers *head* and *tail* point to the head of the queue and the next free slot, respectively.

The use of predicted values The inductive proof utilizes the auxiliary predicted value fields. Every value field v in the system is paired with an auxiliary predicted value pv field, while the interrupt field *int* in the retirement buffer slots is matched with an interrupt predict field, *intpv*.

When an instruction is dispatched its predicted values are calculated. The predicted value of arithmetic operations are calculated by applying the instruction operation to the predicted values of its operands.

The generation of interrupts and taking of branches are decided by the uninterpreted functions *interrupt* and *do_branch*, respectively, whose parameters are available at dispatch time. The same functions and parameters are used to predict whether an interrupt will be taken (*intpv*) and the predicted value (pv) of a branch instruction. Both predictions are trivially correct.

Note: The predicted instruction value, stored in the *pv* field of the retirement buffer should not be confused with the system's branch prediction stored in the *brpv* field. The latter is calculated using a different function and is not guaranteed to be correct.

5 Our Direct Proof that DES Refines SEQ

In this section we discuss our direct proof that system DES refines system SEQ. The bulk of this proof is the proving of invariants used to show that the observable functions of the two systems match. We first discuss our 'top-down' methodology and then explain how it was applied to this problem.

5.1 A Two Stage Top-Down Approach to Invariant Generation

Deductive proofs typically include a number of inter-dependent invariants. The human prover, faced with the necessity of proving a fairly complex property may be uncertain how to begin. We define a simple two stage procedure which we believe provides a framework for proving such invariants in a systematic manner. We note that while only the second step of this procedure is 'top-down' the dominance of this step leads us to call the whole procedure 'top-down'.

1. Formulate, and prove, a set of simple invariants of the data structures and the model. These invariants can be chosen with little or no consideration of the invariant to be ultimately proved. Good candidate properties for this step are simple properties of data-structures or relationships between two data structures. Properties chosen in this step are typically sufficiently simple that they are not dependent on any other properties.
2. Attempt to prove the final correctness invariant using the invariants proved in step one. Should the proof reach a step from which one cannot progress, analyze the situation, define one or more properties which would allow the proof to progress and attempt to prove these properties.

The purpose of the first step is threefold. Firstly, it is likely to expose simple errors in the model, should they exist. It is frequently the case that in writing up the model in the PVS description language an error was made, often a very simple one such as using an incorrect index for an array. Such errors may cause the proof of even simple invariants to fail, and the simpler the proof which fails the easier it is to locate the problem in the model. Secondly, the construction of incorrect properties reflects user misunderstanding. Discovering why such properties are incorrect helps the user comprehend the model more completely. Thirdly, even if these properties were not formulated with the final invariant in mind, they will almost certainly be useful in its proof.

In the second step constant progress is made towards the conclusion of the proof. When the proof fails, and it is expected to, it is generally due to the necessity of proving another invariant first. The second step thus incrementally

reveals the "hidden" properties on which the desired invariant is dependent, generating a string of properties to be proved invariant. The recursive proving of invariants should conclude after a few iterations, allowing the model to be proved correct.

The balance between the two steps is variable. The greater the number of invariants proved in the first step the less frequently the proofs of the second step will fail due to missing properties or simple errors in the model. However, there is no need to worry about too few, or "missing" invariants in the first step. All invariants needed in the proof will be revealed in the second step and can be proved at this point. While invariants proved in the first step will typically be useful, they are not strictly necessary.

The framework described is very flexible, but, we believe, firm enough to provide structure and direction to the proof.

5.2 System DES refines system SEQ

Auxiliary variables used: The auxiliary variables used in this proof are the *reg* field in the operand structures, and the *oc* and *arg* fields of the retirement buffers.

Both SEQ and DES have program counters called *pc* and counters *numinst* counting the number of instructions which have been completed. We will term the variables in SEQ pc_a and $numinst_a$ and those in DES pc_c and $numinst_c$.

For ρ_A^*, we restrict ρ_A by modifying the *delay* variable such that SEQ delays when the two systems have completed the same number of instructions:

$$delay \;:=\; (numinst_a = numinst_c)$$

The system invariant, α, is simply the conjunction of the single system invariants and the equality $\mathcal{O}_C = \mathcal{O}_A$, with the observation functions defined as:

$$\begin{aligned}
&\mathcal{O}_C : (RF, numinst_c, \textbf{if} \quad RB.head = RB.tail \wedge \neg RB.wrap \\
&\qquad\qquad\qquad\qquad \textbf{then } pc_c \textbf{ else } RB.rbe[RB.head].pc\ ) \\
&\mathcal{O}_A : (reg, numinst_a, pc_a)
\end{aligned}$$

Thus, the register files of the two systems always agree. When the retirement buffer is empty the program counters also agree. Otherwise, the program counter of the next instruction to be retired matches the program counter of SEQ.

Proving premises R1, R2 and R4 of the refinement rule is easy, the difficult part is in proving that $\mathcal{O}_C = \mathcal{O}_A$ is an invariant of the system. To do this we must prove that both machines compute the same value for each instruction, and modify the program counter identically. Since both the value and the program counter are influenced by taking an interrupt, we must also show that an instruction generates an interrupt in system DES if and only if it does so in SEQ.

5.3 Invariants Used in Proving the Refinement

In the first stage we prove simple properties of the system, for example, lemmas relating to the structure of the retirement buffer (e.g. if *tail* and *head* point to the same slot then the retirement buffer is full if *wrap* is true, empty otherwise).

We consider now the second stage of the proof. We start by trying to prove that the value in the head retirement buffer slot is the value calculated by SEQ for the given instruction. This property is quickly formalized and divided into four properties. The first states that the value in the head slot is the value that would be obtained by applying the *do_op* or *do_branch* functions to the values in the operand registers:

$$\begin{array}{l}\phi_1 : RB.rbe[RB.head].oc \wedge \neg RB.rbe[RB.head].busy \longrightarrow \\ \quad RB.rbe[RB.head].v = \\ \qquad \textbf{if} \quad type\_op(RB.rbe[RB.head].op) = \textbf{branch} \\ \qquad \textbf{then } do\_branch(RB.rbe[RB.head].pc, \\ \qquad\qquad\qquad iss\_before(numinst, RB, RB.head)) \\ \qquad \textbf{else } \ do\_op(RB.rbe[RB.head].op, \\ \qquad\qquad\qquad RF[RB.rbe[RB.head].arg[1].reg], \\ \qquad\qquad\qquad RF[RB.rbe[RB.head].arg[2].reg])\end{array}$$

However, this invariant is insufficient: the values stored in fields of *rbe* must be matched to counterparts in SEQ to allow us to prove that the computed values are correct. This relationship is asserted by showing that the operation, register, target and branch target fields in the retirement buffer match those in the program used by both systems. We must also prove that the two systems use the same criteria to generate interrupts, and will thus generate interrupts at the same time. Lastly, it is necessary that the program counters in the two systems match, if not they will execute different instructions. Whereas in a purely sequential program the updating of the program counter is trivial, once branches are considered the relation between the instruction indices of two instructions that complete one after the next may vary and the correspondence between the program counters is more complicated.

Of these properties, the relationship between the retirement buffer and the program, and the matching interrupt generation are simple to prove while property ϕ_1 is the most difficult. We concentrate on the proof of this property.

Property ϕ_1 is, intuitively, stating two phenomena – firstly, that the result of the instruction is that obtained from the operands used, and secondly, that the values of these operands can now be found in the register file. This second property, operand correctness, depends primarily on operands with "retired" status having values matching those in the register file:

$$\begin{array}{l}\phi_2 : RB.rbe[rb].oc \wedge RB.rbe[rb].arg[j].st = \textbf{retire} \longrightarrow \\ \qquad RB.rbe[rb].arg[j].v = RF[RB.rbe[rb].arg[j].reg] \wedge \\ \qquad \forall rb' \ RB.rbe[rb'].oc \wedge RB.rbe[rb'].tgt = RB.rbe[rb].arg[j].reg \longrightarrow \\ \qquad\qquad rb = rb' \ \vee \ precede(rb, rb', RB)\end{array}$$

where $preceed(rb, rb', RB)$ is true iff both retirement buffer slots are occupied and slot rb precedes slot rb' in the queue of slots waiting for retirement.

The need to prove that there is no preceding retirement buffer slot targeting the operand registers is crucial: should such a slot exist it would, on retiring, over-write the values in the register file, invalidating any correspondence between the retirement buffer slot operand fields and the register file.

Property ϕ_2, in turn, depends on the value in operand fields matching the closest preceding slot targeting the operand register, when such a slot exists. Property ϕ_3 asserts that while the operand status is **write_b** (the operand value has been written back but the instruction has not yet been retired) such a slot does exist, and its value matches that in the operand fields. In order to prove that there is no slot targeting the registers matching retired operands, as required in ϕ_2, it is necessary now to prove a parallel property: there is no slot targeting the operand register between the instruction slot and the slot pointed to by the operand fields.

In order to prove the invariance of ϕ_3 it is necessary to define an invariant, ϕ_4, defining similar properties for busy operands.

Proving the invariance of ϕ_2, ϕ_3, and ϕ_4 is the most difficult part of the direct proof. Intuitively, these properties assert the correctness of the relationship between instructions and their operands, that instructions always use the value calculated for the operand by the last preceding instruction writing to the operand register. These dependency relations are one of the difficulties of out-of-order executions, and it is unsurprising that proving that they hold is the crux of our correctness proof.

We proved a total of 23 invariants in our proof, many of which were simple technical results, such as proving that if the *head* and *tail* pointers of the retirement buffer are equal then the buffer is either full or empty. We omit further details of these invariants.

6 An Inductive Proof of Refinement

There is an enormous difference between an out-of-order system in which many instructions progress simultaneously and a simple sequential system. Whereas in the direct approach we prove a correspondence between these diverse systems, the inductive approach is based on the premise that it will be easier to prove a number of smaller refinements between systems which are more similar. This approach requires more user effort in defining the multiple refinement relations, an investment which simplifies the invariants which need to be proved.

We have performed induction on the number of slots in the retirement buffer. In the base case, where there is only one slot, the out-of-order machine will operate sequentially as only one instruction can be in progress. The inductive step involves proving that machine DES(B+1) with $B+1$ slots refines one with B slots (denoted DES(B)). The difference between these two machines is intuitively far less than that between an out-of-order system and a sequential one.

The invariants needed to prove the refinement relations were proved using the top-down approach detailed previously. In fact, many of the properties needed were proved as part of the direct proof.

Auxiliary variables used: The predicted value fields in the dispatch and retirement buffers are used, as are the *oc* and *slot* fields of the retirement buffer.

6.1 Base Case: DES(1) Refines SEQ(R)

We consider DES(1), an implementation of DES with only one retirement buffer slot. As was the case of the direct proof, we synchronize the two systems at retirement time by setting the delay variable exactly when the *numinst* variables of the two systems agree. Details of this straightforward proof are omitted.

6.2 The Inductive Step: DES(B+1) Refines DES(B)

We show that a system with $B + 1$ retirement buffer slots refines one with B slots. We have chosen to synchronize at instruction dispatch time.

There are two difficulties here: Firstly, DES(B+1) can store $B + 1$ issued but incomplete instructions whereas DES(B) cannot; secondly, even when the two systems contain the same number of occupied retirement buffer slots, their positions will be different since as soon as the *head* pointer wraps the *head* pointers of the two systems will differ. This technical problem complicates the proof which we therefore divided into two stages. We first prove that DES(B+1) refines DES_f(B+1), a system with $B + 1$ slots in which there is always at least one free slot. We then show that DES_f(B+1) refines DES(B). That is, the first proof proves that a system functioning with one fewer slot refines DES(B), without considering mismatched slot positions, a problem delayed to the second proof.

DES(B+1) refines DES_f(B+1): We run the two systems in parallel, synchronizing at instruction issue. As long as there is at least one free slot in DES(B+1), all the data structures in the two systems are identical. We consider the case of an instruction being issued into the last free retirement buffer slot of DES(B+1).

We cannot issue the instruction in system DES_f(B+1) as this system will not allow all $B + 1$ slots to be occupied simultaneously. We free the slot at the head of the retirement buffer (that pointed to by *head*) and then issue the instruction.

We consider first the case of the *head* slot containing an executed instruction (the *busy* flag is *false*) which is not a mispredicted branch, nor generates an interrupt. This instruction is retired, after which system DES_f(B+1) issues the new instruction. The register files of the two systems are equal except that the value of the target register of the head slot is updated in DES(B+1) with the value found in the head slot in DES_f(B+1).

However, it may be the case that no value is yet available in the *head* slot as the instruction has not yet been executed. In this case the instruction is stored in the dispatch buffer pointed to by the auxiliary *slot* field of the retirement buffer entry. Any operands of the instructions depended on values of previous

instructions, all of which have been retired, and so the instruction will have available operands and can be executed. After execution, the instruction can be retired and the new instruction issued.

The fact that DES(B+1) does not have any value for the instruction makes matching the two systems more difficult. The new value in the register file (assuming that the retired instruction had a target register) of DES_f(B+1) is not found anywhere in the DES(B+1) system. This problem has been overcome by using *predicted values*. The value which has been calculated and retired should be the same value that will be calculated and retired for the instruction at the head of *RB*. We formalize this by predicting the value of all instructions at dispatch time, and later prove that these predictions are correct. We can then assert that

> The predicted value of the head retirement buffer slot in DES(B+1) equals that found in the the r'th register of the register file of DES_f(B+1), where r is the target index stored in the head slot of DES(B+1).

Similarly, dispatch buffer operand values which are now written back in system DES_f(B+1) match the predicted values for these operands in system DES(B+1).

The final case is that of instructions which either generate interrupts or are mispredicted branches. We use predicted values to assert that when the slot at the head of the retirement buffer in DES(B+1) is retired, an interrupt will be generated or a branch misprediction discovered.

Once system DES(B+1) retires the head slot all data structures of the two systems will again match. Until this retirement occurs, DES(B+1) cannot issue another instruction (it has no free slots) but can execute and write-back instructions stored in the dispatch buffer.

Values are predicted correctly In this subsection we sketch our proof that values are predicted correctly.

We would like to prove that value finally obtained for a field matches its predicted value:

$$\begin{aligned}\psi_1 : \forall s : [1..Z], j : [1..2].\ \ & DB[s].oc \wedge DB[s].arg[j].st \neq \textbf{busy} \ \longrightarrow \\ & DB[s].arg[j].v = DB[s].arg[j].pv \\ \wedge\ \forall b : [1..B].\ \ & RB.rbe.[b].oc \wedge \neg RB.rbe[b].busy \ \longrightarrow \\ & RB.rbe[b].v = RB.rbe[b].pv \ \wedge \ RB.rbe[b].int = RB.rbe[b].intpv\end{aligned}$$

The proof is inductive. The base case is the state before the start of execution. Since all dispatch and retirement slots are unoccupied property ψ_1 holds trivially.

Assume that ψ_1 holds at the current state. The next state is obtained by either issuing, executing, or retiring an instruction. These three cases are considered separately.

Consider a data instruction issued into dispatch buffer s and retirement buffer slot *tail*. The busy flag of retirement buffer slot *tail* is set to true, and thus there is no constraint on its predicted values. Each of the two operands s_i of s are looked up in the *RTT*. If the *RTT* entry for s_i is not busy, the value in $RF[s_i]$ is

copied to both the value and predicted value fields of the dispatch buffer. Else, the status, value and predicted value fields are copied from the retirement buffer slot pointed to by the RTT. If the status of the retirement buffer slot is not **busy** then, by the induction hypothesis, its value and predicted values agree. Otherwise, the operand status is set to **busy** and there is no requirement that its value and predicted values agree. Thus, in all cases, if the operand status is not **busy**, its value and predicted value will agree.

We next assume that instruction I is executed and written back. We consider first a data instruction. Both of its operands are available and are not busy and thus, by the induction hypothesis, their value and predicted value fields agree. The value of the instruction is calculated by applying the instruction operation to the value of the operands. As the predicted value was obtained by applying the operation to the predicted value of the operands, the value and predicted values for the instruction will agree. Thus, when the instruction value is written back to any operand fields waiting for it, and to the instruction retirement slot, it will match the predicted value field in these data structures.

Interrupt generation and the predicted values of branches are both decided by the same functions, with the same parameters, as were used to predict the interrupt or the instruction value when the instruction was dispatched. This prediction is trivially correct.

Lastly, we consider instruction retirement. The only value or predicted value fields modified are the value fields in the register file (which have no predicted values). It is easy to prove that ψ_1 continues to hold.

This completes the inductive step. This proof, like all others, has been rigorously proved in the PVS theorem prover. □

Completing the proof of refinement We would like to use the refinement rule of section 2. However, this rule requires that the abstract machine, progress one step with each step of the concrete machine, while we need the abstract system, $\text{DES}_f(\text{B}+1)$, to progress up to three steps with each step of $\text{DES}(\text{B}+1)$.

To overcome this problem we follow Abadi and Lamport [1] in using auxiliary variables to introduce stuttering into the system. We add an auxiliary variable *stutter* to $\text{DES}(\text{B}+1)$ to derive system $\text{DES}_s(\text{B}+1)$. Intuitively, *stutter* is the minimum number of idling steps that the system must take before taking a non-idling step. When an instruction is dispatched into the $B+1$'st slot of $\text{DES}(\text{B}+1)$ *stutter* is set so as to force $\text{DES}(\text{B}+1)$ to idle while $\text{DES}_f(\text{B}+1)$ performs all the necessary actions to retire the *head* slot before dispatching the new instruction. The transition relation is modified so as to idle, decrementing *stutter*, if it is non-zero.

The proof sketched above allows us to show that the stuttering system $\text{DES}_s(\text{B}+1)$ refines $\text{DES}_f(\text{B}+1)$. To complete the proof that $\text{DES}(\text{B}+1)$ refines $\text{DES}_f(\text{B}+1)$ we must show that $\text{DES}(\text{B}+1)$ refines $\text{DES}_s(\text{B}+1)$.

Abadi and Lamport describe formally under which conditions a stuttering system refines a non-stuttering one. Our system fulfills these requirements and so $\text{DES}(\text{B}+1)$ refines $\text{DES}_s(\text{B}+1)$ and therefore $\text{DES}(\text{B}+1)$ refines $\text{DES}_f(\text{B}+1)$.

DES$_f$(B+1) **refines** DES(B)**:** System DES$_f$(B+1) has one more slot than DES(B), but as it can never fill all its slots simultaneously, the two systems function as if they have the same number (B) of slots. The difference in the size of the buffer does, however, affect the values of the *head* and *tail* pointers – after the retirement buffer has wrapped these values no longer agree in the two systems. Similarly, any producer fields, whether in the dispatch buffer or register translation table, do not agree for the two systems, and while each retirement buffer entry in system DES$_f$(B+1) has a matching entry in DES(B) its slot index differs.

A mapping, *map*, is defined from slot indices in DES$_f$(B+1) to those in DES(B). The two systems are run in parallel, both issuing, executing and retiring instructions simultaneously. All data structures in the two systems are identical, modulo the *map* function.

Refinement is thus intuitively simple: ρ_A^* is ρ_A with the non-deterministic choices made as they were in system DES$_f$(B+1). As our observation functions we take the register files of the two systems. Since the register files do not mention retirement slot indices, these are identical at all stages.

7 Liveness Properties

Our system is highly non-deterministic and each of the three sub-instructions (dispatch, execute or retire) can cause the system to idle instead of progressing. There is thus no guarantee that any instruction will ever complete.

However, he have proved that it is always *possible* for the system to progress. That is, there is always at least one instruction in the system which can either be dispatched, executed or retired.

8 Conclusion: Comparing the Two Proofs

In this paper we have shown that both the direct, top-down approach, and an inductive methodology are applicable to proving the correctness of our speculative instruction execution model. We note that we used the top-down approach in proving invariants in the inductive proof, too. The two approaches are not mutually exclusive, however using induction modifies the structure of the proof enormously. In this section we compare the two approaches.

We found that, perhaps counter-intuitively, the inductive proof was far more difficult to construct than the direct proof: it was far easier to prove refinement between a speculative and a sequential system than between two speculative systems where one has one more retirement buffer slot.

Most of the complexity of the inductive proof was in proving that DES$_f$(B+1) refines DES(B+1). The data structures in the two systems are 'almost' the same, but we found it necessary to define *precisely* how they differ, in all circumstances. For example, the dispatch buffers are the same unless DES$_f$(B+1) has retired one instruction more than DES(B). In this case the dispatch buffer of DES$_f$(B+1) will be empty if the retired instruction generated a flush. Otherwise, the value of the

retired instruction may be available in operand fields in $\text{DES}_f(\text{B}{+}1)$ but not in the corresponding fields in $\text{DES}(\text{B}{+}1)$. All the cases sketched in subsection 6.2 had to be rigorously examined and formalized. The invariant α of the superposition system details the differences between *each* data structure of the two systems.

In contrast, in the direct proof the comparison between the abstract and concrete systems involves only the observables, and the internal data structures (dispatch buffer, etc) of the speculative machine are not matched with any in the sequential system. The speculative system is designed so that externally its speculative, out-of-order character is hidden and the register file presents an in-order view of instruction execution. Since we synchronize at retirement time we can compare the register files and not the internal data structures, utilizing the external 'in-order' behavior of the speculative machine so that neither speculation nor out-of-order execution is overtly verified in the refinement proof. Instead, a number of extra invariants of the speculative system were needed to show that it, indeed, behaves 'correctly' – that instruction values are calculated correctly and that the correct instructions are flushed when mispredictions occur. In particular, the instruction-operand relationship expressed by ϕ_2, ϕ_3 and ϕ_4 is used for the purpose of showing that instruction values are correctly calculated. These invariants have trivial counterparts in the base case of the inductive proof, and no counterparts in the inductive step. When we are performing a comparison between two speculative systems these properties hold in both systems and need not be expressed explicitly.

Thus, the different structures of the two proofs resulted in different types of difficulty. In the direct proof the emphasis was on proving single system invariants, in the inductive proof on proving properties of the superposition system. While proving system invariants can be tedious and time consuming, it required less user effort than the complicated, if faster running, refinement analyses in the inductive proof. That single system invariants were easier to formulate than those of the superposition system is reasonable since the the relationship between two systems is potentially more complex than the complexity of each system individually. Since human effort, rather than run-time, is the more limiting factor in deductive proofs of this type, we consider the slower, yet simpler, direct proof to be the more efficient and evaluate the top-down methodology as the one more appropriate for this problem.

Our conclusion is that more important than the similarity of the systems between which we prove refinement is the *complexity* of the two systems and the *granularity* of the comparison between them.

In the inductive proof both the abstract and concrete systems are of similar complexity; in the direct proof the abstract system is far simpler. The complexity of the abstract system contributes directly to the complexity of the refinement proof. Both the definition of the refinement relation and its proof are dependent on the complexity of both systems. For example, in proving premise R1 of the refinement rule we generate for each concrete step a matching transition in the

abstract system. In the direct proof the simplicity of SEQ makes this trivial, in the inductive proof it is more difficult.

The granularity of the comparison is crucial: When the comparison is fine grain it is reasonable that defining it correctly, and then proving it invariant, will be a process requiring a similarly detailed understanding of the systems. When the comparison is coarser much of the complexity is shifted from properties of the superposition system to properties of the individual systems, which, we believe, tend to be simpler to formalize.

When using induction one compares two relatively similar systems. Intuitively, this suggests that a fine grain comparison will often be necessary, as it is only in a detailed examination of the systems that a meaningful comparison can be made. The similarity of the systems seems to be, in this case, detrimental rather than beneficent, implying both a complex abstract system and a fine grain comparison.

The balance between the complexity of the additional single system invariants needed in a direct proof and the complexity of the inductive comparison will, of course, differ from problem to problem. However, it is our contention that not only was the inductive methodology inappropriate for our refinement, but that the difficulties we encountered will often occur when combining induction and refinement: Induction inherently suggests that the abstract system will be of complexity similar to that of the concrete system, with the differences between them small and thus apparent only in a fine grain comparison.

References

1. M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science* 82(2):253–284, May 1991. 498
2. T. Arons and A. Pnueli. Verifying Tomasulo's algorithm by refinement. *Proceedings of the 12'th VLSI design conference*, 1999. 487
3. S. Berezin, A. Biere, E. Clarke and Y. Zhu. Combining symbolic model checking with uninterpreted functions for out-or-order processor verification. *FMCAD'98*:369–386, Palo Alto, 1998. 488
4. W. Damm and A. Pnueli. Verifying out-of-order executions. *CHARME'97*:23–47, Montreal, 1997. Chapmann & Hall. 487, 490
5. Gwennap L. Intel's p6 uses decoupled superscalar design. *Microprocessor Report*, 9(2):9–15, 1995. 487, 490
6. J.L. Hennessy and D.A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., 1996. 487, 490
7. R. Hosabett, G. Gopalakrishnan and M. Srivas. A proof of correctness of a processor implementing Tomasulo's algorithm without a reorder buffer. *CHARME' 99*. 488
8. K.L. McMillan. Verification of an implementation of Tomasulo's algorithm by compositional model checking. *CAV'98*:110–121, 1998. 488
9. S. Owre, J.M. Rushby, N. Shankar, and M.K. Srivas. A tutorial on using PVS for hardware verification. *Proceedings of the Second Conference on Theorem Provers in Circuit Design*:167–188. FZI Publication, Universität Karlsruhe, 1994. 487
10. A. Pnueli, T. Arons. Verification of Data-Insensitive Circuits: An In-Order-Retirement Case Study. *FMCAD'98*:351–368, Palo Alto, 1998. 487

11. J. Sawada and Jr. W.A. Hunt. Processor verification with precise exceptions and speculative execution flushing. *CAV'98*:135–146, Vancouver, 1998. 488
12. J.U. Skakkebaek, R.B. Jones, and D.L. Dill. Formal verification of out-of-order execution using incremental flushing. *CAV'98*:pp 98–110, Vancouver, 1998. 488
13. R.M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM J. of Research and Development*, 11(1):25–33, 1967. 487