Verification of an Advanced MIPS-Type Out-of-Order Execution Algorithm*

Tamarah Arons

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

Abstract. In this paper we propose a method for the deductive verification of out-of-order scheduling algorithms. We use TLPVS, our PVS model of *linear temporal logic* (LTL), to deductively verify the correctness of a model based on the Mips R10000 design. Our proofs use the *predicted values* method to verify a system including arithmetic and memory operations and speculation. In addition to the abstraction refinement traditionally used to verify safety properties, we also use fairness constraints to prove *progress*, allowing us to detect errors which may otherwise be overlooked.

1 Introduction

Modern out-of-order 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 and executing them *out of order* so as to improve performance. However, it is typically required that the results of this out of order execution be the same as that of a sequential execution of the program. Proving this correlation is non-trivial: The out of order scheduling algorithm is often complex, and may use a variety of data-structures not used in the sequential algorithm.

The two prevalent methods for the formal verification of hardware designs are *model checking* (e.g. [ABHS99,BCRZ99,JM01]) and *deductive verification* (e.g. [SH98,HGS00,CGZ96]).

There are obvious advantages to the model-checking techniques, the most important being that it is fully automatic and requires no strong familiarity with the internal details of the design. A very serious limitation of model-checking techniques is the limited size of designs which can be fully automatically verified.

The alternative approach based on deductive verification suffers from no such limitations and, in principle, can be used to verify very big designs provided their structure is based on regular patterns. The main drawback of the deductive approach to reactive system verification (as outlined, for example, in [MP95]) is that it is not fully automatic and requires much user ingenuity and supervision.

* Research supported in part by the John von Neumann Minerva Center for Verification of Reactive Systems, and the European Community IST project "Omega".

R. Alur and D.A. Peled (Eds.): CAV 2004, LNCS 3114, pp. 414–426, 2004.
© Springer-Verlag Berlin Heidelberg 2004

In the past several years we have considered several out-of-order execution designs, developing the *predicted values* method for proving correctness using *refinement*. This method was applied to Tomasulo / Pentium II-like models [AP99,AP00] and to a much simplified Mips model [AP01]. None of these models included memory operations, and progress was not proved. The significant amount of human interaction required was a limiting factor in the complexity of the model, and the types of properties, we could verify.

For this reason, we developed TLPVS [PA03], a system for the formal verification of linear temporal logic (LTL) properties built on the PVS [OSRSC01] verification system. Using TLPVS we were able to apply the predicted values method to a significantly more complex model, based on the Mips R10000. In verifying what is arguably the most elaborate out-of-order execution mechanism verified within the academic community to date, we demonstrate the generality of our methods with respect to different execution models, and the feasibility of their use on more complex models.

Furthermore, TLPVS allows us to use fairness conditions to prove progress properties, checking for a class of errors likely to be missed otherwise. Most abstraction based verification methods for verifying out-of-order execution algorithms prove that there is an abstract state matching every concrete one, but do not ensure that the fairness requirements in the abstract system are met. They are therefore unlikely to detect a livelock or deadlock situation. We demonstrate progress in the concrete system by refinement to a weakly fair abstract system.

This paper makes a number of contributions: It is the first report of the successful verification of a detailed model based on the Mips architecture. In it we demonstrate that the predicted values method is general enough to be used on different architectures and extend its use to systems with memory. Furthermore, we extend the verification method to include progress, thus verifying that deadlock does not occur. We also demonstrate the use and advantages of TLPVS in microprocessor verification.

In the next section we discuss related works. Section 3 describes our model, MIPS, Section 4 explains the use of predicted values and Section 5 overviews TLPVS. In Section 6 we describe our proof of correctness. Space constraints limit the detail in this paper; annotated PVS files are available at [Tlpvs].

2 Related Works

Out-of-order execution mechanisms have been a very popular area of research in the last few years. Quite a number of techniques have been developed and applied to a variety of models. However, the models used and simplifications made are not standardized, making comparisons difficult.

The method of *completion functions* proposed by Hosabettu et al [HGS00] is, like ours, purely deductive, and uses PVS. Completion functions are used to complete every unfinished instruction in the implementation system which can then be compared with the specification system. Whereas the completion functions recursively compute the future value of the instruction, we use predicted values

to obtain the same value without flushing, and without constructing completion functions. We believe that in the examples we have examined predicted values are easier to calculate and support than completion functions.

Whereas completion functions can be seen as implicit flushing mechanisms, explicit flushing mechanisms have also been used in out-of-order verification. In Burch and Dill's seminal paper [BD94] a pipelined systems is verified using refinement and flushing. After flushing the implementation state it is compared with the specification model. However, this approach does not work for out-of-order architectures as flushing the large buffers of partially completed instructions is too complex [SJD98a]. In order to verify out-of-order scheduling an *incremental flushing* mechanism [SJD98a] was proposed, as well as induction [SJD98b].

More recently, Lahiri and Bryant [LB03] used *shadow variables*, as well as refinement maps, to verify various processor models using UCLID, a deductive system based on the logic of CLU. Their shadow variables are taken directly from the abstract system – it is verified that the values computed in the concrete state match these auxiliary values. These auxiliary variables are similar to our predicted value fields, however conceptually our proof of the correctness of prediction is independent of the abstract system, while this proof is dependent on both. They also verify models including superscalar dispatch and retirement (though not, apparently, in conjunction with speculation / memory operations.)

Jhala and McMillan [JM01] use refinement maps to modelcheck out-of-order execution systems. Like [LB03] they use auxiliary variables calculated by the abstract system. This proof has a fundamentally deductive flavor although invariants are proved using model checking. This has the advantage of increased automation. However, the amount of user understanding needed to construct the correct refinement maps, and use the co-induction and abstraction mechanisms is, in our experience, far from negligible. Furthermore, this method relies heavily on symmetry, and it is unclear how it would work in asymmetric systems.

In [SH98] a model including speculation, memory operations, external interrupts and precise exceptions is deductively verified. An intermediate model comprising a table of history variables (MAETT) is used to verify the system in ACL2. The entire system state is stored in this table at dispatch time, and removed if a flush occurs. Like them, we also use auxiliary variable to allow for roll-backs in the case of a mispredicted branch, however we save only memory values, and only when branches are executed. Far more, and more complex, auxiliary variables are used than in our proofs, but none are used to 'predict' future values. The model is impressively detailed, but the proof has the disadvantage of being specific to one configuration and limited to bounded resources.

Our fully parameterized model, MIPS, is not restricted to bounded resources, and we do not explicitly use symmetry in our proofs. However, we exclude exceptions, a feature included in most of the models mentioned above. In the past we have used predicted values to verify systems with exceptions [AP00], and believe that there is no difficulty, in principle, in doing so in this case also.

While other researchers chose to extend their models and thus demonstrate how their methods scale up in the face of increasing complexity, we chose to test

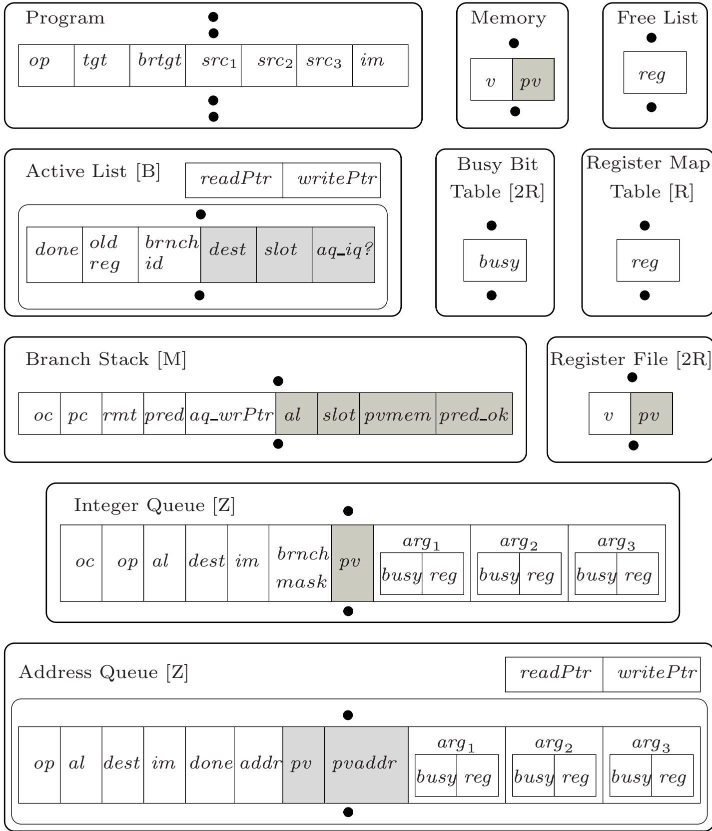

Fig. 1. Data structures for MIPS. Shading indicates auxiliary variables.

the flexibility of our methods by trying them on a totally different out-of-order execution mechanism – that of the MIPS model. To the best of our knowledge no other method has yet been used to verify this out-of-order execution mechanism. Its varied data structures and complex control algorithm make it more challenging than our Tomasulo-based models; we believe that it is the most complex model verified in the academic community to date. Furthermore, this is the first out-of-order processor verification effort which includes a proof of progress – a necessary feature which appears to have been universally overlooked.

3 A Model Based on the Mips R10000

In this section we detail our model, MIPS, based on [Yea96,Gwe94]. We have tried to make our model as accurate as possible, but made some simplifications, and took some assumptions in cases where the exact implementation was unclear to us. The more significant of these are explicitly noted. Our model includes

arithmetic and memory operation, load forwarding, branch prediction and speculation, but not exceptions. The two most prominent differences between the Mips R10000 algorithm and the Tomasulo-like algorithms on which models are generally based are the register renaming and branch verification strategies.

Register renaming is effected by clearly distinguishing the logical register numbers, referenced within instruction fields, from the physical registers, which are locations in the hardware *register file* (*RF*). A *register map table* (*RMT*) maps each logical register to a physical register in *RF*. Other registers in *RF* may be used to hold prior values of logical registers. A *busy-bit table* (*BB*) indicates for each physical register whether it contains a valid value. The *free list* (*FL*) records the list of physical registers not currently in use.

The processor **verifies branch prediction** as soon as the branch condition is determined, even if earlier branches are still pending. If the prediction was incorrect the processor immediately aborts all instructions fetched along the mispredicted path, and restores its state to that before the misprediction. Every dispatched branch is stored in a *branch stack* (*BS*). In addition to the alternate branch address *pc* (program counter), the *BS* stores a copy of the current *RMT*, a *pred* bit indicating whether the branch was predicted taken, and the address queue write pointer position, *aq_wrPtr*[1]. This is significantly more complex than flushing the entire re-order buffer when retiring a mispredicted branch – the mechanism typically used in verification models.

The *active list (AL)* is a circular buffer which maintains program order between all dispatched but not yet retired instructions. (It is functionally comparable to the re-order buffer in Tomasulo-type models.) Arithmetic and branching instructions awaiting execution occupy entries in the *integer queue (IQ)*. Similarly, memory instructions occupy entries in the *address queue* (*AQ*)[2]. We model *memory* as a mapping from address to value, both of which are undefined.

The active list and address queue are both ordered circular buffers, with *write pointers* pointing to the next free entry, and *read pointers* noting the oldest queue entry. The integer queue and branch stack are unordered.

During **dispatch** an instruction is allocated the next entry in the active list. In addition, instructions are allocated entries in the *IQ* or *AQ* depending on their type. The current physical registers for operands are looked up in the *RMT*, their availability checked in the *BB*, and the information stored in the *IQ* (*AQ*). The *IQ* (*AQ*) entry also contains a *dest* field indicating the physical register in which the result will be stored, a tag *al* pointing to the active list entry associated with this instruction and any immediate value (*im*) encoded in the instruction. Address queue entries also have a *done* field recording whether the address has been calculated, and a field storing the address. A branch mask *brnch_mask* is calculated for new entries in the *IQ* by noting which entries of the *BS* are currently occupied. Intuitively, this mask will later be used to determine

[1] It appears that the write pointer and a prediction bit are stored, but this is not explicitly stated in the literature.

[2] The Mips R1000 also has a floating-point queue, very similar to the integer queue, which we have not modeled.

whether this instruction is along the path of a mispredicted branch and should be flushed.

If the instruction has a target register, t, it is also allocated a free physical register from the FL, and the RMT is updated accordingly. The identity of the previous physical register to which t was mapped is stored as *old_reg* in the AL.

Branch instructions are predicted, and the relevant information is stored in a new entry of the branch stack. The index of the BS entry is stored in the AL as the *bid* (branch id)[3].

Address calculation can be performed when all operand values are available. Using operand values obtained from the register file, the address is calculated and stored in the AQ. A *done* bit in the AQ is set after address calculation.

Load execution occurs after the load address has been calculated and the address of all preceding stores in the AQ have also been calculated. If any of these stores is to the same address as the load, then the value of the most recent preceding store is forwarded to the load. Otherwise, the value is obtained from the memory. The new value is written to the target register in the RF, the BB is updated, and the done bit in the AL entry is set.

Arithmetic instruction execution is enabled when all operands in the IQ are ready. The calculated value is written back to the RF, and the BB. The AL entry is updated, as are all matching operand fields in the AQ and IQ. The IQ entry is freed. If the instruction is a *correctly predicted* branch, then the BS entry is freed, and the corresponding branch mask bit is reset in all IQ entries. If the instruction is a *mispredicted* branch, then the program counter (*pc*) is set to the alternative value and all instructions succeeding are flushed: All IQ entries whose branch mask matches this branch are freed; the write pointer of the AL is set to the entry succeeding the branch (effectively freeing all entries after it); the RMT is restored to the values stored in the BS, i.e. all mappings due to instructions succeeding the branch are undone; physical registers allocated to instructions along the mispredicted path are returned to the free list; the write pointer of the AQ is restored to its position when the branch was dispatched.

The instruction at the head of the AL can be **retired** if its *done* bit is set. The physical register stored in its *old_reg* field is freed and added to the free list. Thus, a physical register is freed only when the next instruction targeting the same logical register is retired. The value in the *old_reg* register may be needed as an operand value for instructions dispatched after it but before the next instruction targeting the same logical register. The retirement policy ensures that register values are available as long as there is a possibility that they will be needed. If the instruction is a load or store, the entry at the head of the address queue (which must match this instruction) is also freed. In the case of a store memory is updated with the store value.

To this system we add a number of auxiliary variables (shaded in Fig. 1). Some auxiliary fields duplicate information in a more accessible form (the *aq_iq?* field of AL records whether it is an arithmetic or a memory instruction) or

[3] It is likely that the AL does not really have a *bid* field, and another mechanism, which we could not determine, relates BS entries to those in the AL and IQ.

provide pointers to related data structures (the *slot* fields in the active list points to the instruction's IQ or AQ entry). The auxiliary $BS.pvmem$ field allows us to roll-back memory in the case of a mispredicted branch (Section 4.1). The system also contains many auxiliary *predicted value* fields, the use of which is explained in the next section.

4 Out-of-Order Execution and Predicted Values

Verification of out-of-order (OOO) systems often makes use of *refinement*. That is, the *concrete*, out-of-order *implementation* design is compared to an an *abstract* system *specifying* all the acceptable correct computations. Typically, the abstract system is taken to be a sequential system in which each instruction is completed (issued, executed, retired) in one step.

A difficulty with this comparison is that in the OOO execution systems the register file and memory are updated many cycles after the instructions are issued. Thus the program counter in the two systems may match if we synchronize at dispatch time, but the register file only if we synchronize at retirement time.

We developed the predicted values approach to deal with this disparity. *Predicted values* are auxiliary fields mapped to some or all of the value fields in the system, predicting the eventual value. This prediction is fully deterministic and depends only on other values in the system at the time of its calculation. Using the terminology of [AL91] these are *history*, not prophecy, variables. Predicted values form a "shadow" system mirroring the "real" computation except that all computations occur at dispatch time, using predicted values for operands.

Our method consists of two verification steps:

1. *Correctness of prediction*: We determine conditions under which values and predicted values in the OOO system agree. These are ordinary, single system invariants.
2. *Refinement*: We use refinement to prove a relationship between the predicted values in the OOO system, and values in the sequential system. The systems are synchronized at dispatch time.

This method has the advantage of allowing us to relatively easily synchronize the out-of-order and sequential systems at dispatch time, proving refinement without costly flushing or complicated roll-back mechanisms.

4.1 Using Predicted Values in MIPS

In the initial state all predicted value fields are equal to the real values. When the first instruction is dispatched its predicted value is calculated by applying the instruction operation to the predicted values of its operands. We consider, first, an arithmetic operation. When it is dispatched its predicted value is calculated from the predicted value of its operands in the RF, and stored in the IQ entry and as the predicted value of its target register in the RF.

Similarly, loads and stores calculate their predicted addresses ($pvaddr$) by using predicted values in the RF. The predicted value of a load is the predicted

value at the *pvaddr* location in memory. Stores update the predicted value of the *pvaddr* location in the memory.

When branches are dispatched, the system speculates whether or not to take the branch using *value* fields in the system, a decision noted in the $BS.pred$ field. (Note that this is not a predicted value – it is a system based branch prediction that may be incorrect. It records whether the branch is speculatively taken.) Using *predicted value* fields from the RF for the source operands, the branch condition is evaluated, determining whether it *should* be taken. This value is stored in the *pv* field if the IQ. The *pred_ok* field records whether the branch is mispredicted (by comparing $BS.pred$ with $IQ.pv$). In addition, a complete copy of the memory predicted values is stored in the $BS.pvmem$ field.

When a mispredicted branch is verified and flushed, it is necessary to restore the RMT and the *mem* to the state they were in before the misprediction. Instructions along the mispredicted path may have updated the RMT and these updates are undone by copying the $BS.rmt$ field to the RMT. The real values in memory will not have been affected by these instructions as stores update memory only when they are retired. However, we do update the predicted values in memory when stores are dispatched. Analogously to the treatment of the RMT, we save the predicted memory values as $BS.pvmem$ when a branch is dispatched, and restore them if it is verified as being mispredicted.

Predicted values are neither read nor written during address calculation, instruction execution (excluding misprediction verification) and retirement.

5 A Brief Overview of TLPVS

In order to reduce the enormous manual effort in conducting deductive proofs, we developed TLPVS which includes a formal PVS specification of LTL based on [MP95] and a framework for defining systems. A number of rules for proving safety and response properties are included in the system, each one accompanied by a strategy supporting its use. These rules and strategies greatly reduce the routine theorem proving interaction.

All proof rules used are defined and proved correct within TLPVS. In doing so we eliminate the pen-and-paper application of "known" rules typical in many proofs, and the validity of our final proof rests solely on the correctness of PVS.

5.1 Parameterized Fair Systems

The computational model of *parameterized fair systems* [PA03] is used for defining systems in TLPVS. This is a variation of the fair discrete systems of [KP00] which, in turn is derived from the model of *fair transition systems* [MP95].

A *parameterized fair system* (PFS) $S = \langle V, \Theta, \rho, \mathcal{F}, \mathcal{J}, \mathcal{C} \rangle$ consists of

- V : A finite set of typed *system variables*. We define a *state* s to be a type-consistent interpretation of V. A *(state) predicate* is a function which maps states into truth values. A *bi-predicate* defines a binary relation over states.
- Θ : The *initial condition*. A predicate characterizing the initial states.
- ρ: The *transition relation*. A bi-predicate relating a state to its successor.

- $\mathcal{F}$: A non-empty *fairness domain*. This is a domain which is used to parameterize the fairness requirements of justice and compassion.
- $\mathcal{J}$: The *justice (weak fairness) requirement*. This is a mapping from $\mathcal{F}$ to predicates ($\mathcal{J} : [\mathcal{F} \mapsto \text{predicate}]$). For every $t \in \mathcal{F}$, a computation must contain infinitely many $\mathcal{J}[t]$-states.
- $\mathcal{C}$: The *compassion (strong fairness) requirement*. These were not needed in this proof. See [PA03] for details.

A *run* of a PFS is an infinite sequence of states satisfying the requirements of initiality and consecution. A *computation* is a run satisfying the justice and compassion requirements.

Our definitions of LTL are taken from [MP95], and for brevity are omitted.

6 A Proof of the Correctness of MIPS

In this section we use refinement to prove that every execution of MIPS has a matching sequential execution, thus proving the safety property that MIPS computes values correctly. Thereafter we prove progress by proving that a matching *computation* of the sequential system can be found, in which infinitely many non-idling steps are taken.

More precisely, we prove that our concrete system, MIPS: $S_C = \langle V_C, \Theta_C, \rho_C, \mathcal{F}_C, \mathcal{J}_C, \mathcal{C}_C \rangle$, *refines* an abstract system SEQ: $S_A = \langle V_A, \Theta_A, \rho_A, \mathcal{F}_A, \mathcal{J}_A, \mathcal{C}_A \rangle$ in which each instruction is completed in a single step. Since both MIPS and SEQ include a program counter (*pc*), memory (*mem*), and register file (*RF*), we subscript MIPS instances with "*C*", and SEQ instances with "*A*".

Let Σ_C and Σ_A denote the sets of concrete and abstract states respectively. Let Ω, referred to as the *domain of observations*, denote a set of elements. Let $\mathcal{O}_A : \Sigma_A \mapsto \Omega$ and $\mathcal{O}_C : \Sigma_C \mapsto \Omega$ be two functions termed the *abstract* and *concrete observation functions*, respectively. They indicate the parts of the systems compared in the refinement relations. We define an *interpolating system*

$$S_I = \langle V_I = V_C \cup V_A, \Theta_C \wedge \Theta_A^*, \rho_C \wedge \rho_A^*, \mathcal{F}_C, \mathcal{J}_C, \mathcal{C}_C \rangle$$

where $\rho_A^*(V_I, V_C', V_A')$ and $\Theta_A^*(V_C, V_A)$ may refer to all variables in $V_C \cup V_A$. Functions Θ_A^* and ρ_A^* allow us to "choose" from the possible transitions of SEQ, one that correctly matches the MIPS transition. We denote the V_C (V_A) component of V_I by $V_I\Downarrow_C$ ($V_I\Downarrow_A$, respectively).

Within TLPVS we define, and prove the validity of, rule REF (Fig. 2). Intuitively, R1 and R2 together ensure that the S_A component of a run of S_I is a run of S_A. Premise R1 requires that an initial S_A-state matching the initial S_C-state can be found. Premise R2 requires that an S_A-state satisfying ρ_A^* can always be found, ensuring that an S_I-successor state can be built. Premise R3 asserts that throughout the computation the observation functions of the two systems agree. The conclusion, that S_C *refines* S_A, denoted $S_C \sqsubseteq S_A$, is formalized as:

(1) For every computation seq_C of S_C, there is a run seq_A of S_A, such that at every time t, $\mathcal{O}_C(seq_C(t)) = \mathcal{O}_A(seq_A(t))$.

Rule REF

R1. $\Theta_C(V_C) \longrightarrow \exists V_A : \Theta^*_A(V_C, V_A) \wedge \Theta_A(V_A)$
R2. $\rho_C(V_I \Downarrow_C, V'_C) \longrightarrow \exists V'_A : \rho^*_A(V_I, V'_C, V'_A) \wedge \rho_A(V_I \Downarrow_A, V'_A)$
R3. $S_I \models \Box(\mathcal{O}_C(V_I \Downarrow_C) = \mathcal{O}_A(V_I \Downarrow_A))$

$S_C \sqsubseteq S_A$

Fig. 2. Rule REF: Proving refinement.

Formula Θ^*_A initializes the *pc*, *mem*, and *RF* in S_A with the same values as in Θ_C. Transition relation ρ^*_A defines an instruction execution whenever ρ_C defines a dispatch which is not along a mispredicted path. All other ρ_C-transitions cause ρ^*_A to idle. Premises R1 and R2 are trivial to verify. We discuss the proof of R3:

We define $\mathcal{O}_A$ as the tuple $(pc_A,\ mem_A,\ RF_A)$.

If the current state of MIPS does not include a mispredicted branch (detected by checking the auxiliary *pred_ok* variable in *BS*) then $\mathcal{O}_C$ is defined as

$$(pc_C,\ \lambda a : mem_C.pv(a),\ \lambda r : RF_C(RMT(r)).pv)$$

That is, we take the current program counter and the predicted values for memory. We use the predicted value of each logical register r, obtained from the *RF* by using the *RMT* to identify the physical register to which r is mapped.

Otherwise, $\mathcal{O}_C$ is derived from the branch stack. Letting *firstMis* be the index of the first mispredicted branch, we define $\mathcal{O}_C$ as

$$(BS(firstMis).pc,\ BS(firstMis).pvmem,\ \lambda r : RF_C(BS(firstMis).rmt(r)).pv)$$

That is, we take the alternative branch address store in the *BS*, the copy of memory predicted values stored when the branch was taken, and the predicted values for logical registers obtained using the *BS.rmt* mapping. Intuitively, *BS.pvmem* and *BS.rmt* record "snapshots" of the system before the misprediction occurred, and unlike *RMT* and *mem*, do not include changes made by instructions along the mispredicted path.

Result (1) refers to auxiliary variables in MIPS. From it we derive

(2) For every computation seq_C of S_C, there is a run seq_A of S_A, such that at every time t at which $seq_C(t).AL$ is empty:

$$\begin{aligned} & pc_A &&= pc_C \\ \wedge\ & RF_A &&= \lambda r : RF_C(RMT(r)).v \\ \wedge\ & mem_A &&= \lambda a : mem_C(a).v \end{aligned}$$

by proving that predicted values and real values match under certain conditions.

Verifying that R3 of REF holds required that a number of system properties be proved invariant. Similarly in deducing (2) from (1).

We proved the invariance of 19 general properties if MIPS, such as that if slot S of the *IQ* is occupied, then its *al* field points to an occupied entry in the

AL, in which the *done* bit is false. We also prove the invariance of 7 properties relating directly to predicted value. Properties in this group include prediction correctness (under which conditions value and predicted values are guaranteed to agree) and relationships between predicted values in different structures.

Most of the 26 properties whose invariance we proved, the bulk of the verification effort, fell into 1 of 4 groups of *mutually inductive* properties. We defined, and proved the correctness of, a simple compositional mutual induction rule which allowed us to verify each property in the group separately.

6.1 Proving Progress

The refinement proof described above is not very dissimilar from that of other researchers e.g. [LB03,JM01]. However, we claim that it is insufficient. There are errors that may not be found by the above method – errors which prevent the system from progressing. Consider, for example, a version of MIPS in which we do not return the *old_reg* register to the FL on retirement. Within a finite number of steps the system will flush itself out (AL empties) but be unable to dispatch the next instruction with a target as there are no "free" physical registers. The matching abstract run will be one with an infinite suffix of idling transitions.

To prove progress we define the justice condition for SEQ as $\lambda n : pc_A \neq n$. I.e., for every $n \in \mathbb{N}^+$, we require that infinitely often the program counter is not n. Intuitively, the sequential system can always progress, and doing so causes the program counter to change. However, it is possible for a program to violate this condition by containing a branch, at location m, with m as its branch target. We obviate this undesirable scenario by assuming that $\forall n : prog(n).brtgt \neq n$.

We now try to prove that

(3) For any computation seq_C of S_C, there is a *computation* seq_A of S_A, in which infinitely many non-idling steps are taken, such that at every time t $\mathcal{O}_C(seq_C(t)) = \mathcal{O}_A(seq_A(t))$ and if $seq_C(t).AL$ is empty then:

$$\begin{aligned} & pc_A &&= pc_C \\ \wedge\ & RF_A &&= \lambda r : RF_C(RMT(r)).v \\ \wedge\ & mem_A &&= \lambda a : mem_C(a).v \end{aligned}$$

That is, we try to show that every computation of MIPS matches a *fair* run of SEQ in which progress is made (infinitely many instructions are completed). It is easy to derive (3) from (1) and (2) once we prove the response property

$$(4)\quad \text{MIPS} \models \forall n : \Box((getPC = n) \longrightarrow \Diamond(getPC \neq n))$$

where $getPC$ is defined as pc_C if the system contains no mispredicted branches, $BS(firstMis).pc$ otherwise.

We use a derivation of the WELL rule of [SPBA00] (Fig. 3) to verify (4).

The justice requirements we define for MIPS are that every sub-transition (dispatch, address calculation, execution, retirement) which is enabled infinitely often is eventually taken.

We define a ranking function which decreases every time a partially executed instruction which is not along a mispredicted path progresses (e.g. its address is

Rule WELL
For PFS $S = \langle V, \Theta, \rho, \mathcal{F}, \mathcal{J}, \mathcal{C} \rangle$,
Given initial and goal predicates p, q, helpful predicates $\{h_t : t \in \mathcal{F}\}$,
a well founded relation $\succ$ over $\mathcal{A}$, and ranking functions $\delta : \Sigma \mapsto \mathcal{A}$

$$\text{W1.} \quad p \rightarrow q \vee \bigvee_{t \in \mathcal{F}} h_t$$

$$\text{W2.} \; \forall t \in \mathcal{F} : \; h_t \wedge \rho \rightarrow q' \vee \bigvee_{u \in \mathcal{F}} (\delta \succ \delta' \wedge h'_u) \vee (h'_t \wedge \delta = \delta' \wedge \neg \mathcal{J}'[t])$$

$$\square(p \longrightarrow \diamond q)$$

Fig. 3. Rule WELL. Primed variables refer to values in the next state.

calculated, or the instruction is retired). Instructions along mispredicted paths do not effect the rank. When an instruction which is not along a mispredicted path is dispatched (a *goal dispatch*), the *pc* changes and the goal state is reached.

The helpful predicate requires that if AL is empty then an instruction be dispatched, otherwise that the instruction at the head of AL progress. To prove that h'_u always holds for some u, we show that the instruction at the head of the AL (if any) can always progress, and that a dispatch is always enabled if the AL is empty. The latter required us to prove new safety properties regarding resource recovery, properties unnecessary for the proof of (2).

Justice conditions ensure that instruction at the head of the queue progresses, and thus the rank decreases. Well-foundedness ensures that as long as no goal dispatch occurs, the rank continuously decreases until the AL is empty. At this point justice conditions ensure that a goal dispatch occurs.

7 Conclusion

In this paper we present the predicted values method for the verification of out-of-order execution. Using predicted values we are able to prove refinement between a complex out-of-order execution system and a simple sequential system.

Our ability to verify a model of the size and complexity of MIPS is in no small part thanks to the use of TLPVS, which eliminates part of the drudge work. Despite this, our proofs are by no means automatic, and the effort required (two to three person months) is significant. However they are *fully* automated, with every rule being proved within the PVS theorem prover (as part of TLPVS). We have not sufficed at using a "known" refinement rule (or any other rule), but have verified the rule as well. In totally eliminating the pen-and-paper element, we believe that we provide a higher degree of certainty than most previous proofs.

Acknowledgments

I benefited much from the insight of Prof Amir Pnueli, my thesis adviser, particularly in the development of TLPVS. Many thanks to Orna Lichtenstein for valuable suggestions on an earlier draft of this paper.

References

[ABHS99] A. Aziz, J. Baumgartner, T. Heyman, and V. Singhal. Model checking the IBM gigahertz processor. *CAV'99*:72–83, 1999.

[AL91] M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, 1991.

[AP99] T. Arons and A. Pnueli. Verifying Tomasulo's algorithm by refinement. In *VLSI'99*:306–309, 1999.

[AP00] T. Arons and A. Pnueli. A comparison of two verification methods for speculative instruction execution. In *TACAS'00*:487–502, 2000.

[AP01] T. Arons and A. Pnueli. A methodology for deductive verification of out-of-order execution systems based on predicted values. Technical Report MCS01-04, Weizmann Institute, 2001.

[BCRZ99] A. Biere, E. Clarke, R. Riami, and Y. Zhu. Verifying safety properties of a PowerPC microprocessor using symbolic model checking without BDDs. In *CAV'99*:60–71, 1999.

[BD94] J. R. Burch and D. L. Dill. Automatic verification of pipelined microprocessor control. In *CAV'94*:68–80, 1994.

[CGZ96] E.M. Clarke, S.M. German, and X. Zhao. Verifying the SRT division algorithm using theorem proving techniques. In *CAV'96*:111–122, 1996.

[Gwe94] L. Gwennap. MIPS R10000 uses decoupled architecture. *Microprocessor Report*, pages 18–24, October 1994.

[HGS00] R. Hosabettu, G. Gopalakrishnan, and M. Srivas. Verifying microarchitectures that support speculation and exceptions. In *CAV'00*:521–537, 2000.

[JM01] R. Jhala and K. McMillan. Microarchitecture verification by compositional model checking. In *CAV'01*:397–410, 2001.

[KP00] Y. Kesten and A. Pnueli. Control and data abstractions: The cornerstones of practical formal verification. *STTT*, 2(1):328–342, 2000.

[LB03] S.K. Lahiri and R.E. Bryant. Deductive verification of advanced out-of-order microprocessors. In *CAV'03*:341–354, 2003.

[MP95] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag, New York, 1995.

[OSRSC01] S. Owre, N. Shankar, J.M. Rushby, and D.W.J. Stringer-Calvert. *PVS System Guide*. Menlo Park, CA, November 2001.

[PA03] A. Pnueli and T. Arons. TLPVS: A PVS-based LTL verification system. In *Verification: Theory and Practice*:598–625, 2003.

[SH98] J. Sawada and W.A. Hunt. Processor verification with precise excpetions and speculative execution flushing. In *CAV'98*:135–146, 1998.

[SJD98a] J.U. Skakkebaek, R.B. Jones, and D.L. Dill. Formal verification of out-of-order execution using incremental flushing. In *CAV'98*:98–110, 1998.

[SJD98b] J.U. Skakkebaek, R.B. Jones, and D.L. Dill. Reducing manual abstraction in formal verification of out-of-order execution. In *FMCAD'98*:2–17, 1998.

[SPBA00] E. Sedletsky, A. Pnueli, and M. Ben-Ari. Formal verification of the Ricart-Agrawala algorithm. In *FSTTCS'00*: 325–335, 2000.

[Tlpvs] TLPVS Homepage. http://www.wisdom.weizmann.ac.il/~verify/tlpvs.

[Yea96] K.C. Yeager. The Mips R10000 superscalar microprocessor. *IEEE Micro*, pages 28–40, April 1996.