A Methodology for Deductive Verification of Out-Of-Order Execution Systems Based on Predicted Values*

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 propose a methodology for the deductive verification of out-of-order scheduling algorithms. A 'top-down' scheme for the systematic definition of system invariants is defined. The complementary use of predicted values, auxiliary fields storing a dispatch time prediction of an instruction's value, is proposed as a means of further simplifying the verification of systems in this class. We illustrate the use of the 'top-down' methodology and predicted values in the verification of three out-of-order scheduling algorithms, including a detailed discussion of the verification of a model based on the Mips R10000.

1 Introduction

The two prevalent methods for the formal verification of hardware designs are *model checking* (e.g. [ABHS99,BCRZ99,BBL98]) and *deductive verification* (e.g. [SH98,GHS99,CGZ96]). There are obvious advantages to the model-checking techniques, the most important one being the fact that it is fully automatic and requires no strong familiarity with the internal details of the design. This is why model checking is the technique almost exclusively used and developed internally in most of the hardware industries such as Intel, IBM, and Motorola.

The extensive application of this technique to industrial designs led to a clear identification of the limitations of the model-checking technique, the most serious of which is the limited size of designs which can be fully automatically verified. Today, the size of designs which can be standardly verified is bounded by having no more than 1,000 boolean variables (bits) which is a far cry from modern micro-processor circuits which reach size of 20,000,000 boolean elements.

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. This is a very frequent feature of hardware designs which consider words made up of 8 - 64 bits, and register files as well as various memory elements which are organized as large arrays of identical basic units (words, registers, etc,).

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 require user ingenuity and supervision on two levels: The definition of the auxiliary constructs – the system invariants and the well-founded rule in liveness proofs – requires a large measure of ingenuity. Thereafter, some method must be found to prove these verification conditions.

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

This paper is based on a thesis that if we concentrate on a particular class of circuits then we can develop a uniform methodology for the type and character of auxiliary constructs needed. This methodology can be used for all circuits, past, present, and future belonging to this family.

This thesis has already been adopted by some of the hardware industries for the verification of some arithmetical floating point circuits [CGZ96,KLM98,ML96]. In this paper we wish to report the success of the approach to another family of hardware circuits, out-of-order designs. In the past several years we have considered several instances of such circuits, developing a methodology for formulating their correctness problems and systematically deriving the necessary invariants for their proof [DP97,AP99,PA98,AP00]. In this paper we discuss one of the more useful principles of our methodology, predicted values. We also report here the successful verification of the most elaborate circuit which we have attempted in this family.

The rationale behind this thesis is that the initial investment in the detailed study of the common characteristics of this family and formulation of the appropriate deduction methodology will pay later in an extensive re-use of this methodology for future circuits. The paper demonstrates that this has been true for the incremental sequence of designs we have considered so far.

2 Out-Of-Order Execution and Refinement

Out-of-order execution is used in modern microprocessors to increase the number of instructions executed per cycle. By choosing from a window of instructions one which is ready to be executed the processors minimize waiting due to dependencies. Each instruction is first dispatched, then executed, written back, and, depending on the algorithm, may still have to be retired. The instruction value will be available in some data structures after it has been written back, in others only after retirement.

Verification of such 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. The correspondence between the systems is with respect to *observation functions*, which, intuitively, define the features of the two systems which are significant for the comparison. Intuitively, the two systems are run in parallel, ensuring that after each step the observation functions agree. Since the abstract system (in our case generally a sequential execution system) is assumed to behave correctly, we know that the observables in the concrete system are correct. By choosing the observation functions appropriately we can show that the concrete system functions correctly.

Intuitively, the observation functions should include the values in the register file as it is the correctness of these values which implies the correctness of the algorithm. The problem with doing so is that in out-of-order execution systems the register file is updated many cycles after the instructions are issued and, in certain algorithms, is updated in the order in which instructions were executed and not in program order. The register files of the two systems thus never agree and no straightforward equivalence can be made.

An example The following code fragment illustrates the above explanation:

$$
\begin{aligned}
I1 &: R1 \leftarrow R2 + R3 \\
I2 &: R2 \leftarrow R1 + R2 \\
I3 &: R3 \leftarrow R3 + R3
\end{aligned}
$$

Assume that values for all three registers $R1$, $R2$ and $R3$ are available before dispatching instruction $I1$. The following is a a possible execution sequence:

Instruction $I1$ is dispatched, after which $I2$ and then $I3$ are also dispatched. Both instructions $I1$ and $I3$ are ready for execution, however $I2$, being dependent on $I1$, is not. Instruction $I3$ may be executed first, followed by $I1$. Once the value for $I1$ has been written back $I2$, too, can be executed.

Of course, $I1$ must not use the value for $R3$ calculated by $I3$, but rather $R3$'s value when $I1$ was issued. Each out-of-order algorithm uses some mechanism to ensure this, and the verification of this mechanism is one of the biggest difficulties in verifying out-of-order scheduling algorithms.

3 Predicted values and the 'top-down' methodology

Predicted values are auxiliary fields which are 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 Abadi and Lamport [AL91] these are *history* variables, not prophecy variables. The predicted value is not non-deterministically prophesied, but rather calculated using other values in the system.

Predicted values form a 'shadow' system mirroring the 'real' computation except that all computations occur at dispatch time, using predicted values for operands. We start in a flushed state setting all predicted values equal to the real values (since the state is flushed, all values are available). When the first instruction is dispatched its predicted value is calculated by applying the instruction operation to the predicted values of its operands. The predicted value of this instruction is now available and any subsequent instructions can use it as a predicted operand value. Thus, although the operands of a dispatched instruction may not yet have values, they will always have predicted values, which allow the predicted value to be calculated at dispatch time.

Claim. Non-pending values match their predicted values.

Proof sketch We sketch an inductive proof which was formally verified in the PVS theorem prover.

As the base case we take a flushed system in which all predicted values are set to equal their matching values. The claim holds trivially.

Assume that all available values in the system match their predicted values.

Consider an instruction which is dispatched. Its predicted value is calculated using the predicted values of its operands. Since its value is pending the hypothesis holds.

When the instruction is executed its value is calculated using the values of its operands, values which are no longer pending. By the inductive hypothesis, these are the values predicted for the operands. Since the predicted value for the instruction was calculated using the same operation and these correctly predicted values, the value calculated equals that which was predicted. (This assumes that the correct operand values are used when the instruction is executed, an assumption which must be verified.) Thus, this value was correctly predicted and on being written back and / or retired it will match any predicted value fields for it in the system. □

In each proof using predicted values the correctness of prediction should be formalized by a set of properties which are then proved to be invariants of the system. In our experience this is a relatively simple task.

We illustrate the use of predicted values using the example of the previous section: When $I1$ is issued we calculate its predicted value $pv(R1)$ as $pv(R2) + pv(R3)$. Although no value for $R1$ is available when $I2$ is issued, its predicted value can be calculated using the predicted values of $R2$ and $R1$. The predicted values are thus calculated using the three sequential equations:

$$\begin{aligned} P1 &: pv(R1) \leftarrow pv(R2) + pv(R3) \\ P2 &: pv(R2) \leftarrow pv(R1) + pv(R2) \\ P3 &: pv(R3) \leftarrow pv(R3) + pv(R3) \end{aligned}$$

That is, the predicted value calculation represents an *in-order execution* of the code where (predicted) values are calculated at dispatch time and are thus available for the next instruction. Assuming that for every register x $pv(Rx) = Rx$ just before $I1$ was issued, it is easy to see that the predicted values *correctly predict* the values that will be calculated later for these registers.

3.1 The 'top-down' proof methodology

Our proofs include a number of inter-dependent invariants. These invariants can be systematically defined and proved using a simple two stage procedure:

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. Appropriate properties include simple properties of data-structures or relationships between two data structures.
2. Attempt to prove the final correctness invariant using the invariants proved in step one. Typically, this invariant would state the equivalence of the observation functions in the concrete and abstract machines. When using predicted values one would also prove the correctness of prediction.
   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.

We note that only the second step is 'top-down', however the dominance of this step leads us to term the whole methodology 'top-down'.

The purpose of the first step is threefold: Firstly, it is like to expose any simple errors in the model. In particular, errors in modeling the system in the prover description language may be found. Secondly, the construction of incorrect properties reflects user misunderstanding. Discovering such mistakes helps the user comprehend the model more fully. Thirdly, even in these invariants were not formulated with the final invariant in mind, they will almost certainly be useful in its proof.

While this first step is not strictly necessary, it has the advantage of allowing system errors and user misunderstandings to be revealed in relatively simple proofs, which are far easier to analyze and understand.

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.

4 Using predicted values: three case studies

In this section we discuss three applications in which we used predicted values. All of our proofs were verified in the PVS [ORSS94] theorem prover.

4.1 Tomasulo's Algorithm

This is the first, and the simplest, of the three applications, and is detailed in [AP99]. Tomasulo's algorithm is a simple out-of-order execution, out-of-order retirement, scheduling algorithm. The version that we verified dealt only with register to register operations, ignoring exceptions and branches.

The register file contains a busy bit indicating, for each entry, whether the value is valid or pending. When instructions are issued they set the busy bit in the register file. A register translation table notes for each register which dispatch buffer entry is calculating its updated value. The instruction is copied into the dispatch buffer along with its operand information, including the predicted value for the operand (copied from the register file), its value (if any) or the index of the dispatch buffer which will compute the operand value (copied from the register translation table.)

When both the operands of an instruction in the dispatch buffer are available the instruction can be executed: it vacates the dispatch buffer and its value is written back to both the register file and to any operand fields in the dispatch buffer awaiting this result.

We matched every value field in the system with a predicted value field, and then, in manner similar to the proof of Section 3, showed that values are predicted correctly.

In our refinement proof we used as the abstract system a sequential model, synchronizing the two systems at dispatch time. As observables we took the program counters of the two systems (allowing us to synchronize the two systems), the register file of the sequential system and the *predicted values* of the registers of Tomasulo. Thus, the predicted value of every register in Tomasulo equals the register's value in the sequential system. When all instructions have completed, no results are pending and so the values in Tomasulo are available, equal their predicted values, and hence also equal the values in the sequential system.

Comparison with alternative proofs for this system The predicted value proof was designed to simplify a more complex, multi-stage proof involving an intermediate abstraction previously developed by Damm and Pnueli [DP97]. Before trying predicted values we successfully verified the algorithm using recursive functions to calculate the value that would eventually be calculated for each instruction (recursive calls calculate the values of operands which are not yet available.) The predicted value proof requires a small fraction of the time and effort needed in the previous attempts. Its advantage is its *efficiency*.

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 [SJD98]. In order to verify out-of-order scheduling an *incremental flushing* mechanism [SJD98] was proposed. Using the Stanford Validity Checker a two stage approach was developed: First it is shown that the out-of-order implementation refines an intermediate in-order-execution abstraction in which instructions are buffered until they are retired in the implementation system. Functional equivalence between this abstraction and the specification system is then shown by incrementally flushing the pipeline and buffers in the intermediate abstraction. In a subsequent paper [SJD98b] this methodology was modified so as to avoid the need for an in-order abstraction. Instead, induction is used over the number of instructions that can be in progress simultaneously. We compare this approach with our inductive approach to verifying a different model in Section 4.2.

In [BBCZ98] symbolic model checking and deductive methods are combined to verify Tomasulo's algorithm. A reference file containing symbolic values is used to implement uninterpreted functions, and abstraction techniques are developed to reduce the size of the model. Flushing is used when comparing the implementation and specification systems. The flushing mechanism, and the equivalence of the states, are proved using a combination of theorem proving and model checking. This approach is applicable only to designs of limited size, although this problem may be alleviated somewhat by combining it with other approaches.

A purely deductive proof of Tomasulo's algorithm using *completion functions* is presented in [GHS99]. Completion functions are used to complete every unfinished instruction in the implementation system which can then be compared with the specification system. This is essentially a flushing mechanism. Whereas the completion functions recursively compute the future value of the instruction, we use predicted values to obtain this without flushing, and without constructing completion functions. That is, the completion functions and predicted values, intuitively, compute the same values, but we believe that in this example predicted values are very easy to calculate, whereas completion functions are complicated to construct.

The advantage of our method over those mentioned above is that predicted values allow us to obviate the need for a flushing mechanism or an intermediate abstraction. Using predicted values we can prove the correctness of the values in a flushed state without explicitly flushing the system.

Ken McMillan [McM98] also verified Tomasulo's algorithm without using a flushing mechanism. Refinement maps are defined between the specification state and the implementation state, and then verified using the SMV model checker. The specification state values are used as implementation state auxiliary variables – it is verified that the values computed in the implementation 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 specification system, while this proof is dependent on both. This proof has a fundamentally deductive flavor although invariants are proved using model checking. This has the advantage of increased automation, but the disadvantage that the system configuration must be fixed and the arithmetic functions modeled. Furthermore, it is not clear how the refinement maps, which required 'substantial human insight' were derived. We believe that the 'top-down' methodology of Section 3 gives a framework for defining the invariants needed.

4.2 A speculative model based on the Pentium II

In [AP00] we compare two proofs of a speculative, in-order-retirement design based on the Pentium II [Gwe95]. Predicted values were used in an inductive proof in the following manner:

We wanted to show that an out-of-order execution design DES(B) with a retirement buffer of B entries refines DES(B+1) which has a retirement buffer of size $B+1$. Every partially completed instruction in the system resides in an entry in the retirement buffer and thus the retirement buffer size determines the number of instructions that can be in progress simultaneously. We synchronized the two systems so that DES(B) dispatched, executed, or retired an instruction exactly when DES(B+1) did. However, when DES(B+1) dispatches an instruction into its $B+1$'st slot DES(B) is unable to follow suit. We forced DES(B) to free a buffer entry by executing (if not already executed) and then retiring the oldest instruction in its retirement buffer. The value of this executed instruction is written back to operand fields in the dispatch buffer (as in the Tomasulo algorithm) and is later retired to the register file. Thus, both the dispatch buffer operand fields and the register file in DES(B) contain values still pending in DES(B+1). This asymmetry continues until DES(B+1) executes and retires the instruction, which may occur a number of steps in the future, with both systems executing other instructions in the interim.

Here we used predicted values to ensure that the value that would be calculated and retired by DES(B+1) in the future is the same value that has be calculated and retired by DES(B). We matched the *predicted values* of the two systems, while also proving that the prediction is correct. This allowed us to prove that as soon as DES(B+1) retires its oldest buffer (which it must do unless it idles indefinitely) all values in the two systems will again match.

If the retired buffer generates a flush (it was a mispredicted branch or generated an exception) we used predicted values to prove that DES(B+1) would also have to flush before it could retire any other instruction. (In addition to the predicted value field we had a "predict exception" field which predicted which instructions would generate exceptions, decided in our system by an uninterpreted function.)

We note that induction was used as part of the proof in [SJD98b] of the correctness of an out-of-order execution, in-order-retirement design. In this case the induction is over the number of instructions which are in progress simultaneously. Since the size of the retirement buffer is a limit on the maximum number of instructions in progress, there is a similarity between these two approaches. Their approach focuses on re-ordering *sub-instructions* (issue, retirement, etc) in an execution. By limiting the size of the retirement buffer we in some cases force earlier retirement of the oldest instruction, thus generating some limited re-ordering. However, our re-ordering is more restrictive, and, we believe, less complex than that in [SJD98b].

Comparison with alternative proofs for this system We did not attempt any other techniques of the inductive step of this proof. A different proof was attempted for this system which totally omitted this step in which predicted values were used.

In [SH98] a model including speculation and memory instructions is verified. An intermediate model comprising a table of history variables is used to verify the system in ACL2. While the model verified is impressively detailed, the proof has the disadvantage of being specific to one configuration.

4.3 A model based on the Mips R10000

The Mips R10000 [Yea96] out-of-order execution mechanism is very different from that of the Pentium II and Tomasulo's algorithm. Our simplified implementation model, IMPL, includes speculation but not exceptions.

Intuitively, the system implements register renaming by doubling the set of physical registers. A *register translation table* maps each logical register to a physical register. A *busy-bit table* indicates for each physical register whether it contains a valid value.

Each dispatched instruction is allocated an entry in the dispatch buffer, a physical register, and an entry in the active list. The physical register for each source operand is looked up in the register translation table, and stored in the operand field of the dispatch buffer, along with the busy bit from the busy bit table. The dispatch buffer 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.

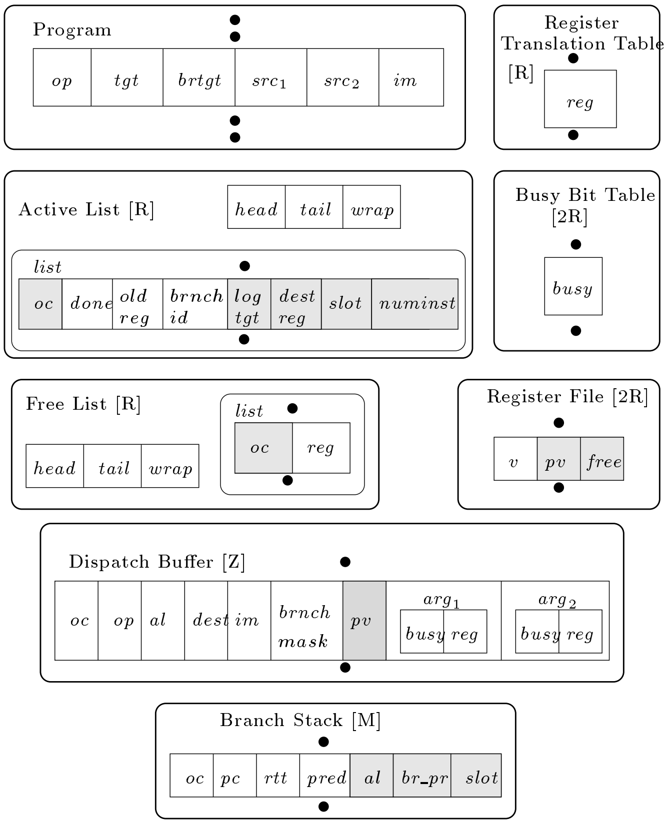

Fig. 1. Data structures for a model, IMPL, based on the Mips design

The *free list* is a list of all physical registers which are not in use. The first free register is allocated to the new instruction, its busy-bit is set to true, and the register translation table is updated. The instruction is also allocated the next free slot in the *active list*. The active list is a queue in which instructions wait for retirement. The *old* physical register for this logical register, the one pushed out of the register translation table, is stored in the active list as is a *done* bit indicating whether the instruction has been executed.

The *branch stack* stores, for every branch, whether the branch was predicted taken, (*pred*) the alternative branch address (*pc*), and a complete copy of the register translation table (*rtt*) when the branch was taken. The dispatch buffer contains, for each instruction, a *branch mask*, indicating which entries of the branch stack were occupied when the instruction was dispatched i.e. which branches lie on the path to this instruction. If the dispatched instruction was a branch it is allocated an entry on the branch stack. We also store the branch stack entry index in the active list (*brnch id*): It is unclear how the information linking an instruction to its branch stack entry is stored in the system; although the assumption that this link is made in this manner may be erroneous, we do not think that it significantly modifies the verification effort.

An instruction can be *executed* when both of its operands are available. The operand values are read from the register file and used to calculate the instruction value. The instruction then vacates the dispatch buffer, writing its value to the register file, while simultaneously setting the active list *done* bit to true and the busy-bit table entry to not busy. All dispatch buffer operands awaiting this value are also set to not-busy.

If the instruction was a correctly predicted branch its branch stack entry is freed and the corresponding bit in the branch mask of all instructions in the dispatch buffer is reset. However, if the instruction is an incorrectly predicted branch all instructions succeeding it are flushed from the system. All active list entries succeeding that for the executed instruction are flushed, and their registers reclaimed on the free list. All dispatch buffer entries whose branch mask includes the branch index of this instruction, that is, all dispatch entries for instructions succeeding this branch, are also flushed. Similarly, any branch stack entries for branches succeeding this one are also freed. Other, preceding, branch instructions may still be outstanding in the system. The branch recovery is completed by restoring the register translation table from the branch stack of the mispredicted branch. That is, the register translation table is now exactly as it was before the branch was taken and no longer contains any registers from instructions following this mispredicted branch.

The *retirement* of instructions is simple: when the instruction at the head of the active list is *done* it can be retired by removing it from the active list. The physical register stored in its *old* 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* 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 insures that register values are available as long as there is a possibility that they will be needed.

To this system we added a number of auxiliary variables (shaded in Fig. 1) including predicted value fields in the register file and dispatch buffer. Some auxiliary fields duplicate information in a more accessible form, e.g. The occupied flag, *oc*, is true exactly when the buffer is entry, and the *free* field in the register file is true when the register is in the free list. This information can be obtained by manipulating other data structures but we found it preferable to duplicate it in this form. Other auxiliary fields provide pointers to related data structures (the *slot* fields in the active list points to the instruction's dispatch buffer entry, the *al* field in the branch stack points to the related active list entry). The auxiliary *br_pr* field in the branch stack predicts whether the system prediction of the branch is correct. That is, this value is true when the system correctly predicts the branch, false otherwise. The decision as to whether a branch is taken is determined in our system by an uninterpreted function, *branch_act* which takes as parameters the instruction operation, its operand values, and an immediate value. While the operand values are not available at dispatch time, their (correctly) predicted values are. Using these predicted values we can use the *branch_act* function to determine whether the system branch prediction is correct.

A note on the active and free lists: Both the active and free lists are circular arrays which are treated as queues; the pointers *head* and *tail* point to the head of the queue, and the next free slot, respectively. Whether a buffer is occupied is determined by the *head*, *tail* and *wrap* fields (when *head* and *tail* point to the same entry *wrap* determines whether the buffer is empty or full).

A proof of the correctness of the model

Before showing that IMPL refines a sequential system SEQ, we use the 'top-down' methodology to prove that values are predicted correctly.

Values are predicted correctly We start by proving some simple properties of system IMPL. For example, we prove that the auxiliary *free* field in the register file is true exactly when the the register appears in the free list

$$\psi_1 : \forall p : RF[p].free \leftrightarrow \exists fl : FL[fl].oc \wedge FL[fl].pr = p$$

and that if a register's busy bit is set then it is the destination of a dispatch buffer:

$$\psi_2 : \forall pr : BB[pr] \longrightarrow \exists s : DB[s].oc \wedge DB[s].dest = pr$$

Similarly, if a dispatch buffer entry is busy then it points to an occupied active list entry which is not *done*:

$$\psi_3 : \forall s : DB[s].oc \longrightarrow AL[DB[s].al].oc \wedge \neg AL[DB[s].al].done$$

In the first stage of the 'top-down' methodology we prove the invariance of a number of system properties. We then progress to the second stage where our goal is to show that the value and predicted value agree in every physical register which is occupied (not in the free list) and not pending (its busy bit is not set).

$$\phi_1 : \forall pr : \neg RF[pr].free \wedge \neg BB[pr] \longrightarrow RF[pr].v = RF[pr].pv$$

This depends on the register file having the same predicted value as the dispatch buffer targeting it,

$$\phi_2 : \forall s : DB[s].oc \wedge DB[s].dest \neq 0 \longrightarrow DB[s].pv = RF[DB[s].dest].pv$$

and on the predicted value in the dispatch buffer being correct, that is, being the value obtained by applying the instruction operation to the values of the registers stored as its operands. In proving that an invariant holds in the next state we assume it holds in the current state, and so we can use ϕ_1 to assert that the values and predicted values of the operand registers are equal since that they are not busy (we must also prove that they are not free). It then suffices to prove that:

$$\begin{aligned}\phi_3 : \forall s : DB[s].oc \longrightarrow DB[s].pv = \\ calc\_val(DB[s].op, RF[DB[s].arg[1].pr].pv, RF[DB[s].arg[2].pr].pv)\end{aligned}$$

Proving ϕ_2 and ϕ_3 invariant require a number of properties of system IMPL which do not involve predicted values, such as the property that dispatch buffer operands point to occupied registers. Some of these properties have already been proved invariant in the first step of the 'top-down' algorithm others are proved when needed. Further details of these proofs are omitted.

In addition to proving that values are predicted correctly, we also prove that we correctly predict whether branches are correctly predicted. Noting that the *pred* fields in the branch stack is a boolean indicating whether the branch is predicted taken, and that the value of a branch instruction is one if it should be taken, zero otherwise, we prove that

$$\begin{aligned}\phi_4 : \forall bs : BS[bs].oc \longrightarrow (&BS[bs].br\_pr \leftrightarrow \\ &BS[bs].pred = if\ DB[BS[bs].slot].pv = 1\ then\ \mathbf{true}\ else\ \mathbf{false})\end{aligned}$$

System IMPL **refines system** SEQ We synchronized our model with a sequential model, allowing both to dispatch at the same time. As long as no mispredicted branches are outstanding the predicted values of the registers pointed to by the register translation table equal the register values in the sequential system (*reg*). If there is a mispredicted branch then the predicted values of the registers indicated by the branch stack entry for the first mispredicted branch equal those of the sequential system. The sequential system idles while IMPL dispatches and executes instructions following an incorrectly predicted branch.

The observation functions of the abstract and concrete systems are

$$\mathcal{O}_A = (reg, pc)$$

$$\begin{aligned}\mathcal{O}_C = if\quad & bid = 0\ then\ ((\lambda r : RF[RTT[r].al].pv), pc)\\ else\quad & \qquad\qquad\quad ((\lambda r : RF[BS[bid].RTT[r].al].pv), BS[bid].pc)\end{aligned}$$

where *pc* is the program counter and *bid* is the branch stack index of the first mispredicted branch, zero if there are no mispredicted branches. We use the *br_pr* auxiliary field to determine the correct value for *bid*.

Once we have proved the equivalence of the observation functions we can use the invariants ϕ_1, ψ_2 and ψ_3 to prove that when the active list is empty (the system contains no partially executed instructions) no registers are busy and therefore the register files of the two systems agree:

$$AL.head = AL.tail \wedge \neg AL.wrap \longrightarrow reg = (\lambda r : RF[RTT[r].AL].v)$$

Note: For simplicity we divide the IMPL transition into three sub-transitions, dispatch, execute, and retire, which we assume are taken separately. This technique was used in previous proofs ([AP99,PA98]) where we also proved that every joint transition is equivalent to a finite sequence of sub-transitions. This equivalence has not yet been proved for this model.

Comparison with alternative proofs for this system Our first attempt at verifying this system involved using partial orders: the sub-instructions (dispatch, execute, retire) of the IMPL system can be re-ordered until we arrive at a sequential ordering in which each instruction is completed before the next is dispatched. While this approach is elegant in theory, in practice it involved numerous complicated proofs and was eventually abandoned. The proof using predicted values is simpler by an order of magnitude.

We are unaware of any other attempts to formally verify a model of this type.

5 Appropriate use of predicted values

We have developed the 'top-down' methodology for verifying out-of-order execution systems, and have found that it is very often enhanced by coupling it with the use of predicted values. However, there have been systems which we preferred to verify without using predicted values [PA98,AP00]. In this section we evaluate the costs and benefits of using predicted values.

The biggest advantage of predicted values is that they allow the concrete and abstract systems to be synchronized at *dispatch time* while still matching the two systems on value fields – with the concrete system using predicted value fields. This obviates the need for flushing mechanisms typically used in the verification of out-of-order scheduling algorithms, mechanisms which entail a significant overhead. The completion functions approach of Hosabettu et al [GHS99], another mechanism to overcome the need for flushing, effectively uses recursive functions to calculate the predicted values when they are needed, in a process which we believe is more complicated than the use of auxiliary variables.

Synchronizing at dispatch time simplifies refinement proofs in that an immediate correspondence can be made between values in the abstract and concrete systems. When synchronizing at retirement time it is frequently necessary to prove explicitly the relationship between the value and the code – the value equals the result of applying the operation of an instruction at a particular program counter to the values of the source registers in this line of code. Properties of this type generally need not be proved explicitly when synchronizing at dispatch time: It is sufficient that both systems calculate the same value and increment the program counter identically.

The use of predicted values also dovetails well with the 'top-down' methodology. The primary invariant to be proved is the correctness of prediction, which is easy to define and prove using this methodology.

In the three proofs discussed in Section 4 the overhead of using predicted values was very low – predicted value fields were paired with the relevant value fields and a few easy to prove lemmas sufficed to show that prediction was correct. Generally, proving correctness of prediction is not a difficult task. These three proofs have another common denominator: in each case it is easy to locate and track the values to be matched with the abstract system. When this is not the case using predicted values is less straightforward.

The second Pentium proof in [AP00] used neither induction nor predicted values, instead synchronizing with the sequential system at retirement time. Our choice in this matter was motivated in part by the difficulty of locating the relevant values at dispatch time. The register file in this system is updated only at retirement time and thus is not an appropriate structure for predicted values. We would have to maintain an auxiliary 'dispatch time' register file, defining its relationship to other data structures, especially subsequent to the dispatch of flush-generating instructions. While this would not be a very complex procedure, it does imply the building and maintaining of an artificial structure, and a proof which, being based on this structure, is likely to appear somewhat contrived. In [SH98] an auxiliary MAETT table was used to verify a similar model. Instructions are added to this table at dispatch time and are removed if a flush occurs. Here far more, and more complex, auxiliary variables are used than in our proofs. However, the auxiliary variables are not used to 'predict' future values, but rather to record completed instructions in a manner which later allows refinement to a sequential execution. Thus they do not, directly, encounter our need to locate which instructions will be flushed, rather 'undoing' erroneous executions when the occasion arises.

While predicted values are particularly appropriate to dispatch time synchronization, synchronizing at retirement time has the advantage that one can compare 'real', rather than predicted, values. Thus, one can prove that, after each step, the concrete system stores the correct values. When using predicted values and synchronizing at dispatch time we have a rigorous proof of the *eventual* correctness of the system, once all instructions have completed. Although, formally, we do not need to have a proof of the correctness of each state to prove the correctness of the system, such a proof has an intuitive advantage.

6 Conclusion

We have applied our 'top-down' methodology to four different out-of-order execution systems, each of increasing complexity. We believe that our methodology is appropriate to this class of systems as a whole; as familiarity with this class of systems and their verification increases the application of the methodology becomes more efficient allowing the verification of increasingly complex systems.

In the proofs of three of these systems we used predicted values. While we do not claim that these systems could not have been verified without predicted values, we do believe that the combination of predicted values and the 'top-down' methodology allowed us to systematically construct comparatively clean and simple proofs. By obviating the need for a flushing mechanism typically used in such proofs predicted values significantly reduce the proof complexity; the fact that an intermediate abstraction is unnecessary is a further advantage. We do not know of any other methodology which would have been as appropriate.

In all three proofs analyzed in this paper the concrete and abstract systems were synchronized at dispatch time, allowing maximum advantage to be derived from using predicted values. Furthermore, the values in the system were in structures which made the predicted values easy to check and maintain. This aided use of predicted values: where the structures are not so compatible auxiliary data structures may be needed. Incompatible structures complicate, but do not prevent, the use of predicted values.

One disadvantage of predicted values, as we have used them, is that we prove the correctness only of flushed, or final, states in the system. This is sufficient to prove the correctness of the system, however there is no explicit proof of the correctness of every state in the system.

Thus, we propose the 'top-down' methodology as an appropriate scheme for verifying out-of-order execution systems, noting that the use of predicted values, when appropriate, results in relatively short, simple and systematic proofs.

References

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

[AP99] T. Arons and A. Pnueli. Verifying Tomasulo's algorithm by refinement. In N. Sherwani and A. Kumar, editors, *Twelfth International Conference on VLSI Design*, pages 306–309, Goa, India, Jan. 1999.

[AP00] T. Arons and A. Pnueli. A comparison of two verification methods for speculative instruction execution. In S. Graf and M. Schwartzbach, editors, *Sixth International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '00)*, volume 1785 of *Lect. Notes in Comp. Sci.*, pages 487–502, Berlin, Gemany, Mar. 2000. Springer-Verlag.

[ABHS99] A. Aziz, J. Baumgartner, T. Heyman, and V. Singhal. Model checking the ibm gigahertz processor : An abstraction algorithm for high performance netlists. In N. Halbwachs and D. Peled, editors, *Proc.* 11^{th} *Intl. Conference on Computer Aided Verification (CAV'99)*, volume 1633 of *Lect. Notes in Comp. Sci.*, pages 72–83. Springer-Verlag, 1999.

[BBL98] I. Beer, S. Ben-David, and A. Landver. On-the-fly model checking of RCTL formulas. In A. Hu and M. Vardi, editors, *Proc.* 10^{th} *Intl. Conference on Computer Aided Verification (CAV'98)*, volume 1427 of *Lect. Notes in Comp. Sci.*, pages 184–194. Springer-Verlag, 1998.

[BBCZ98] S. Berezin, A. Biere, E. Clarke, and Y. Zhu. Combining symbolic model checking with uninterpreted functions for out-of-order processor verification. In G. Gopalakrishnan and P. Windley, editors, *Proc. 2nd Intl. Conference on Formal Methods in Computer-Aided Design (FMCAD'98)*, volume 1522 of *Lect. Notes in Comp. Sci.*, pages 369–386. Springer-Verlag, 1998.

[BCRZ99] A. Biere, E. Clarke, R. Riami, and Y. Zhu. Verifying safety properties of a powerpc microprocessor using symbolic model checking without BDDs. In N. Halbwachs and D. Peled, editors, *Proc.* 11^{th} *Intl. Conference on Computer Aided Verification (CAV'99)*, volume 1633 of *Lect. Notes in Comp. Sci.*, pages 60–71. Springer-Verlag, 1999.

[BD94] J. R. Burch and D. L. Dill. Automatic verification of pipelined microprocessor control. In *Proc.* 6^{th} *Intl. Conference on Computer Aided Verification (CAV'94)*, volume 818 of *Lect. Notes in Comp. Sci.*, pages 68–80. Springer-Verlag, 1994.

[CGZ96] E. Clarke, S. German, and X. Zhao. Verifying the SRT division algorithm using theorem proving techniques. In R. Alur and T. Henzinger, editors, *Proc.* 8^{th} *Intl. Conference on Computer Aided Verification (CAV'96)*, Lect. Notes in Comp. Sci., pages 111–122. Springer-Verlag, 1996.

[DP97] W. Damm and A. Pnueli. Verifying out-of-order executions. In H. Li and D. Probst, editors, *Advances in Hardware Design and Verification: IFIP WG 10.5 Internatinal Conference on Correct Hardware Design and Verification Methods (CHARME)*, pages 23–47, Montreal, 1997. Chapmann & Hall.

[GHS99] G. Gopalakrishnan, R. Hosabettu, and M. Srivas. A proof of correctness of a processor implementing tomasulo's algorithm without a reorder buffer. In L. Pierre and T. Kropf, editors, *Correct Hardware Design and Verification Methods (CHARME '99)*, volume 1703 of *Lect. Notes in Comp. Sci.*, pages 209–233, Germany, Sept. 1999. Springer-Verlag.

[Gwe95] L. Gwennap. Intel's P6 uses decoupled superscalar design. *Microprocessor Report*, 9(2):9–15, 1995.

[KLM98] M. Kaufmann, T. Lynch, and J. S. Moore. A mechanically checked proof of the correctness of the kernel of the amd5k86 floating-point division program. *IEEE Transactions on Computers*, 47(9):913–926, 1998.

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

[McM98] K. McMillan. Verification of an implementation of Tomasulo's algorithm by compositional model checking. In A. Hu and M. Vardi, editors, *Proc.* 10^{th} *Intl. Conference on Computer Aided Verification (CAV'98)*, volume 1427 of *Lect. Notes in Comp. Sci.*, pages 110–121. Springer-Verlag, 1998.

[ML96] P. Miner and J. Leathrum. Verification of IEEE compliant subtractive division algorithms. In M. Srivas and A. Camilleri, editors, *Proc.* 1^{st} *Intl. Conference on Formal Methods in Computer-Aided Design (FMCAD'96)*, volume 1166 of *Lect. Notes in Comp. Sci.*, pages 64–78. Springer-Verlag, 1996.

[ORSS94] S. Owre, J. M. Rushby, N. Shankar, and M. K. Srivas. A tutorial on using PVS for hardware verification. In R. Kumar and T. Kropf, editors, *Theorem Provers in Circuit Design (TPCD '94)*, volume 910 of *Lecture Notes in Computer Science*, pages 258–279, Bad Herrenalb, Germany, Sept. 1994. Springer-Verlag.

[PA98] A. Pnueli and T. Arons. Verification of data-insesitive circuits: An in-order-retirmenet case study. In G. Gopalakrishnan and P. Windley, editors, *Proc. 2nd Intl. Conference on Formal Methods in Computer-Aided Design (FMCAD'98)*, volume 1522 of *Lect. Notes in Comp. Sci.*, pages 351–368. Springer-Verlag, 1998.

[SH98] J. Sawada and W.A. Hunt. Processor verification with precise excpetions and speculative execution flushing. In A. Hu and M. Vardi, editors, *Proc. 10^{th} Intl. Conference on Computer Aided Verification (CAV'98)*, volume 1427 of *Lect. Notes in Comp. Sci.*, pages 135–146. Springer-Verlag, 1998.

[SJD98] J. Skakkebaek, R. Jones, and D. Dill. Formal verification of out-of-order execution using incremental flushing. In A. Hu and M. Vardi, editors, *Proc. 10^{th} Intl. Conference on Computer Aided Verification (CAV'98)*, volume 1427 of *Lect. Notes in Comp. Sci.*, pages 98–110. Springer-Verlag, 1998.

[SJD98b] J. Skakkebaek, R. Jones, and D. Dill. Reducing Manual Abstraction in Formal Verification of Out-of-Order Execution. In G. Gopalakrishnan and P. Windley, editors, *Proc. 2nd Intl. Conference on Formal Methods in Computer-Aided Design (FMCAD'98)*, volume 1522 of *Lect. Notes in Comp. Sci.*, pages 2–17. Springer-Verlag, 1998.

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