Microarchitecture verification by Compositional Model Checking

Ranjit Jhala[1]* and Kenneth L. McMillan[2]

[1] University of California at Berkeley
[2] Cadence Berkeley Labs

Abstract. Compositional model checking is used to verify a processor microarchitecture containing most of the features of a modern microprocessor, including branch prediction, speculative execution, out-of-order execution and a load-store buffer supporting re-ordering and load forwarding. We observe that the proof methodology scales well, in that the incremental proof cost of each feature is low. The proof is also quite concise with respect to proofs of similar microarchitecture models using other methods.

1 Introduction

Compositional model checking methods reduce the proof of a complex system, through decomposition and abstraction, to a set of lemmas that can be verified by a model checker. It has been shown that the proof of systems with unbounded or infinite state can be reduced to tractable model checking problems on finite state abstractions. For example, an instruction processing unit using Tomasulo's algorithm [Tom67] was proved using the method [McM00] for unbounded resources. The proof was substantially simpler than that of a similar model using a general purpose theorem prover [AP99]. The safety proof involved just three simple lemmas verified by a model checker. The relative simplicity of the proof using compositional model checking owed principally to the lack of user generated inductive invariants and the lesser need for manual proof guidance. Nonetheless, the important question of the *scalability* of the method remains open. That is, does the manual proof effort increase in reasonable proportion to the size and complexity of a system?

We approach this question by considering the verification of a complete processor microarchitecture, containing most of the important features of a modern microprocessor. These include branch prediction, speculative execution, out-of-order execution (with in-order retirement and clean exceptions) and a load-store buffer supporting re-ordering and load forwarding. The question is whether the complexity of the proof increases by some reasonable increment with each new architectural feature, or whether it increases intractably, making proofs of complex systems impractical. We find that the incremental proof cost of each architectural feature is small (just a few additional lemmas) and that the interaction of these features, though complex, does not make the proof expand intractably.

* Supported by SRC contract 99-TJ-683.003, AFOSR MURI grant F49620-00-1-0327, NSF Theory grant CCR-9988172

The microarchitecture model that we verify is similar in its feature set to models that have been verified using theorem proving methods [HGS00,SH98]. We compare our proof to the proofs obtained by these methods, with emphasis on the use of inductive invariants and its effect on proof complexity.

Section 2 provides a brief overview of the proof method. Then section 3 describes the microarchitecture model that we verified, and its specification. In section 4 we discuss the proof, and consider the question of scalability. Section 5 compares the proof with proofs obtained previously for similar microarchitectures. In section 6 we conclude with some remarks on the strengths and weaknesses of the method, and how the weaknesses might be addressed.

2 Background

To verify the microarchitecture, we use the SMV proof assistant [McM00]. This tool supports the reduction of correctness conditions for unbounded or infinite-state systems to lemmas that can be verified by model checking. Here, we briefly sketch the proof methodology.

The general approach is to divide the intended computation into "units of work" that use only finite resources in the implementation, such as instructions in a processor, or packets in a packet router. Correctness of a given unit of work is then reduced to a finite state problem using a built-in collection of abstract interpretations. In effect, we disregard those components of the system state not involved in the given unit of work. Because specifications can be temporal, we avoid the need to write and verify an inductive invariant of the system. Instead, we exploit the model checker's ability to compute the reachable states (strongest invariant) of the abstract models. This greatly simplifies the proofs.

The proof methodology A system is specified with respect to a reference model. For a processor, this is an "instruction set architecture" (ISA) model that executes one instruction at a time in program order. The correctness condition is a temporal property relating executions of the implementation to executions of the reference model. We decompose correctness into "units of work" by specifying *refinement relations*. These are temporal properties specifying the data values at internal points in the implementation in terms of the reference model. For example, in a processor we may specify the operands read from the register file and the results computed by the ALU. To make such specifications possible, we may add auxiliary state variables that record the correct data values as they are computed by the reference model. A definitional mechanism in the proof assistant allows us to add auxiliary variables in a sound manner.

Mutually inductive temporal proofs The refinement relations are then proved by mutual induction over time. Each refinement relation is a temporal property of the form $G\phi$, meaning that ϕ is true at all times t. To prove that ϕ is true at time t, we may assume by induction that the other refinement relations hold for all times less than t. This is useful in a methodology based on model checking, because the notion that q up to time $t-1$ implies p at time t can be expressed in temporal logic as $\neg(q \,\mathcal{U}\, \neg p)$. Hence, this proposition can be checked by a model checker.[1] This mutually inductive approach is important

[1] In some cases we can also assume that another refinement relation holds for all times *less than or equal* to t, provided we do not do this in a circular way.

to the proof decomposition. It allows us to assume, for example, when proving correctness of an instruction's source operand, that the results of all earlier instructions have been correct. Note that this is quite different from the method of proof by invariant, in which we show that some state property at time $t-1$ implies itself at t. Here the properties are temporal, and the inductive hypotheses are assumed for all times less than t, and not just at $t-1$. This is important, since it allows us to avoid writing auxiliary invariants.

Temporal case splitting Next we specialize the properties we wish to prove, so that they depend on only a finite part of the overall state. For example, suppose there is a state variable v, which is read and written by processes $p_1 \ldots p_n$. We wish to prove a property $G\phi$ of v. We add an auxiliary state variable w which points to the most recent writer of variable v. Now, suppose we can prove for *all* process indices i that $G((w = i) \Rightarrow \phi)$. That is, ϕ holds whenever the most recent writer is p_i. Then $G\phi$ must hold, since at all times w must have some value. We call this "splitting cases" on the variable w, since it generates a parameterized property with one instance for each value of w. For a given value of i, we may now be able to abstract away all processes except p_i, since the case $w = i$ depends directly only on process p_i.

Abstract interpretation Finally, we wish to reduce the verification of each parameterized property to a set of tractable model checking problems. The difficulty is that there may be variables in the model with large or unbounded ranges (such as memory addresses) and arrays with a large or unbounded number of elements (such as memory arrays). We solve this problem by using abstract interpretation to reduce each data type to a small number of abstract values. For example, suppose we have a property with a parameter i ranging over memory addresses. We reduce the type A of memory addresses to a set containing two values: the parameter value i, and a symbol $A \setminus i$ representing all values other than i. In the abstract interpretation, accessing an array at location i will produce the value of that location, whereas accessing the array at $A \setminus i$ produces $\perp$, a symbol representing an unknown value.

In effect, for each time the user "splits cases" on a variable of a given type, there is one value in the abstract type and one element in each abstracted array indexed by that type. If there are two parameters i and j of type A, the proof assistant may split the problem into two cases: one where $i = j$ and one where $i \neq j$. Alternatively, it may consider separately the cases $i < j$, $i = j$ and $i > j$, if information about the order of these values is important to the property.

The abstractions used by the proof assistant are sound, in the sense that validity of a formula in the abstract interpretation implies validity in the concrete model for all valuations of the parameters. Of course, the abstraction may be too coarse to verify the given property (*i.e.*, the truth value in the abstract model may be $\perp$) even though the property is true. Note, however that the user does not need to verify the correctness of the abstraction, since this is drawn from a fixed set built into the proof assistant.

The proof process proceeds as followings. First, the user specifies refinement relations (and other lemmas, as necessary), which are proved by mutual temporal induction. These properties are parameterized by "splitting cases" on appropriate variables, so that any particular case depends on only a finite part of the system state. Finally, the proof assistant abstracts the model relative to the parameter values, reducing the types with large or unbounded ranges to small finite sets. The resulting proof obligations are discharged by a model checker.

We now consider how this methodology can be applied to processor microarchitectures with features such as speculative execution, out-of-order execution and load-store buffers.

3 The Processor Model

The processor microarchitecture that we model has out-of-order, speculative execution using a variant of Tomasulo's algorithm with a reorder buffer. It implements branch prediction and precise exceptions, and has an out-of-order load-store buffer with load forwarding. For simplicity, we separate program and data memories. The model is generic, in that many functions, such as the ALU (arithmetic-logic unit) and the instruction decoder have been replaced by uninterpreted function symbols. A specific ISA may be implemented by defining these functions appropriately. Our proof, however, is independent of these functions.

3.1 The Specification

The microarchitecture is specified with respect to a reference model, which executes one instruction per step in program order. The ISA consists of the following instruction classes. A *load* (LD) takes two register operands, source address and destination. It reads data memory at the source address, and loads the value into the destination register. A *store* (ST) takes two register operands, the source and the destination address. It stores the source value at the destination address in data memory. An *ALU* operation (ALU) takes two register operands and a destination register. This generic instruction models all the instructions using the ALU by a single uninterpreted function. Although we do not explicitly model immediate operands, these can be folded into the generic ALU function. A *branch* (BC) performs a test on its two register operands. If true, it sets the program counter to the branch target value. Both the test and the branch target computation are modeled by uninterpreted functions. A *jump* (JMP) sets the program counter to the address in the source register. This is to implement non-local jumps such as returns from exception handlers. Finally, an *output* operation (OUT) sends its register operand to the processor's output port. The LD, ST and ALU operations can cause an exception to be raised, in which case control is transferred to the exception handler address. Asynchronous interrupts are not modeled.

3.2 The Implementation Model

The microarchitecture is depicted in figure 1. It is out-of-order, in that instructions are executed when their operands are available, not necessarily in program order. Instruction execution begins by fetching the instruction from program memory at the program counter address (PC). The instruction is then decoded to determine the operation type, the operand registers, the branch target, *etc.* The program counter is updated by incrementing its current value. Since the increment depends on the instruction width, we model incrementation by an

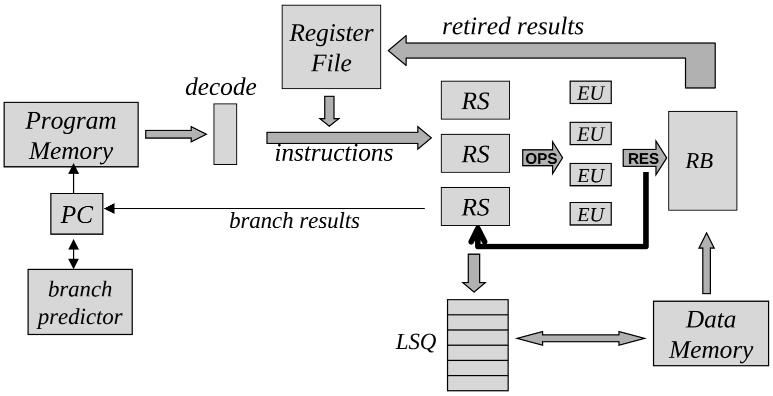

Fig. 1. Microarchitecture

uninterpreted function. In case of a conditional branch, however, the branch predictor guesses the value of the branch condition. Thus we continue fetching instructions even though the actual branch condition is not yet known, at the risk of having to cancel the ensuing instructions if the guess is incorrect. If the predicted branch condition is true, the PC is loaded from the branch target. Since branch predictions do not affect correctness, the branch predictor is modeled as a non-deterministic choice, though this can be replaced by any desired function.

The instruction then reads its source operands from the register file, and is loaded into the next available *reservation station* (RS) to await execution. A source register may contain an actual data value, or it may contain a *tag*, pointing to the RS that will produce the data value when it completes. In the case of a tag, the RS must wait until the corresponding data value returns on the result bus (RES). When both operand values are available, the instruction may be *issued* to an execution unit. When the result of the operation is computed, it returns on the result bus, with its tag, and may be forwarded to any instructions holding that tag. The result is stored in the *reorder buffer* (RB) until the instruction *retires*. At retirement, the result is written to the register file. Instructions are retired in program order, so that the state of the register file is always consistent. This allows clean recovery from exceptions or mispredicted branches.

When a branch instruction retires, we compare the computed value of the branch condition to the predicted value. If these are not the same, subsequent instructions may have been fetched from an incorrect program counter. Thus, they must be *flushed*. When this happens, the program counter is set to the alternative that was *not* chosen at fetch time.

Load and store operations are recorded in a *load-store buffer* (LSQ) in program order. In our model, this buffer is unbounded, however it could be refined by any fixed size buffer. Loads and stores are not necessarily executed in program order. A load operation may execute after it has issued (*i.e.*, its operands have been obtained) and after all earlier stores *to the same address* have executed. Alternatively, a load instruction may execute by *forwarding* the data value from the most recent store to that address, even if that store has not yet executed. A store instruction can execute after it has issued, and after all previous loads *and* stores to the same address have executed.[2]

The above conditions avoid the classic hazard conditions (RAW, WAR and WAW), guaranteeing correct operation even when operations occur out of program order. In addition, we must ensure that a store cannot execute until the instruction has actually retired, since the store cannot be undone if the instruction were to be flushed. When a store instruction retires, it is marked *committed* in the load-store buffer, and cannot subsequently be flushed. The choice of which available operation to execute is non-deterministic, though this could be replaced by any desired scheduling policy.

4 Verification

Our correctness criterion is that the sequence of output values produced by the reference model and the microarchitecture model should be the same, for corresponding initial states. The reference model chooses non-deterministically at each time whether to take a step. By witnessing this choice, we align the reference model's operation temporally with that of the implementation.

The two most interesting aspects of the proof deal with speculative execution and with partially ordered operations, such as register reads/writes or memory loads/stores. We introduce proof decompositions to handle these situations, using compositional model checking.[3]

4.1 Specifying refinement relations

Our basic approach is to decompose the proof into "units of work", in this case instructions. We prove correctness of a single instruction, relative to the reference model, given that all earlier instructions execute correctly. To reduce the verification complexity, we may further decompose the instruction into smaller steps, such as operand read, result computation, memory load, *etc.* We then write refinement relations, specifying the data values at various points in the implementation, in terms of the reference model.

Of course, to specify data items in the implementation, we must determine their correct values. This is done by defining *auxiliary variables* that record the correct data values as computed by the reference model. For example, when an instruction is fetched, the reference model executes it atomically, computing the correct operand and result values. The instruction is then stored in an RS. We record the correct operands and result for that RS. For example, here is the SMV code that does this:

$$
\begin{array}{l}
\mathbf{if}(\neg \mathit{stallout} \wedge \mathit{iopin}\ \mathbf{in}\ \{\mathit{ALU},\mathit{LD},\mathit{ST},\mathit{BC}\})\{\\
\quad \mathbf{next}(\mathit{aux}[\mathit{st\_choice}].\mathit{opra}) := \mathit{opra};\\
\quad \mathbf{next}(\mathit{aux}[\mathit{st\_choice}].\mathit{oprb}) := \mathit{oprb};\\
\quad \mathbf{next}(\mathit{aux}[\mathit{st\_choice}].\mathit{res}) := \mathit{res};\}
\end{array}
$$

[2] Note this implies that the actual address operands of all earlier stores (and loads) must be known before a load (store) can execute.

[3] Proof and prover may be found at `http://www-cad.eecs.berkeley.edu/~kenmcmil`

Here, *st_choice* is the index of the reservation station, and *opra*, *oprb* and *res* are values from the reference model. We now specify that, when the reservation station holds an operand value, it is equal to the stored correct value in the *aux* structure (and similarly for result values).

To do this, we must take into account speculative execution. That is, if an instruction occurs after an exception or a mispredicted branch, we say it is *shadowed*. A shadowed instruction does not correspond to any instruction executed by the reference model. Thus we cannot specify its correct operand and result values. In fact, these values are spurious, and must never affect the register file or memory. To write refinement relations, we must know whether an instruction in the implementation is shadowed. Fortunately, this is easy to determine. We set an auxiliary state bit *shadow* when the predicted branch condition differs from the correct branch condition, or when an exception occurs. The *shadow* bit is cleared when a flush occurs. Here is the SMV description:

$$\begin{array}{l} \mathbf{init}(\mathit{shadow}) := 0; \\ \mathbf{next}(\mathit{shadow}) := \neg \mathit{flush} \wedge (\mathit{shadow} \vee \\ \quad \neg \mathit{stallarch} \wedge (\ \mathit{exn\_raised} \vee (\mathit{opin} = \mathit{BC} \wedge \mathit{taken} \neq \mathit{itaken}))); \end{array}$$

Here, *taken* is the correct branch condition (from the reference model) and *itaken* is the predicted branch condition. Now, any instruction fetched while *shadow* is true is marked shadowed, by setting the auxiliary bit $aux[st\_choice].shadow$. While *shadow* is set, we stall the reference model, since no valid instructions are being executed.

Now we write the refinement relation for operands. We specify that if a *non-shadowed* RS holds an operand value, it must be the correct value. Here is the specification for the a operand:

$$\begin{array}{l} \mathbf{forall}(k \ \mathbf{in}\ \mathit{TAG})\ \mathbf{layer}\ \mathit{lemma1}: \\ \quad \mathbf{if}(st[k].valid \wedge st[k].opra.valid \wedge \neg aux[k].shadow) \\ \quad\quad st[k].opra.val := aux[k].opra; \end{array}$$

This specifies the a operand value for RS k, when it is *valid* (holding and instruction), and when the a operand is valid, and when it is not shadowed. Otherwise the value is unspecified. We can write a similar specification for the result value, and for other data values in the machine as necessary.

4.2 Verifying operand correctness

Now we must verify the above lemma. To verify data, we split cases on the possible sources of the data. Here, an operand value we read is generated by the most recent instruction to write the source register. We can identify this instruction's RS by recording the tag of the most recent RS to write each register. We then assume, by induction, that results computed at earlier times are correct. We need one additional fact, however: that the most recent writer in execution order is in fact the most recent writer in program order. If this is the case, then we must read the same value read by the reference model.

One way to establish this is to split cases on both the most recent writer in the implementation *and* the most recent writer in program order. Since the implementation retires instructions in program order, these two must be the same, hence correct values are always read. However, there is a complexity problem:

the abstraction in this case will involve three distinct tag values, and hence the states of three distinct RS's. In practice, we found the time and space required to verify this model prohibitive. Instead, we used an intermediate lemma to simplify the problem. We observed that a register value is only read when no writes to the register pending, in which case its value is up-to-date with respect to the reference model. Thus, we specified the register contents as follows:

$$\textbf{forall } (i \textbf{ in } \mathit{REG}) \textbf{ layer } \mathit{uptodateReg}: \\ \quad \textbf{if } (\neg ir[i].\mathit{resvd})\ ir[i].\mathit{val} := r[i];$$

That is, if no write is pending to register $ir[i]$, its value matches reference model register $r[i]$. This is verified using the case split described above, which is given to SMV as follows:

$$\textbf{subcase } \mathit{uptodateReg}[i][k][c] \textbf{ of } ir[j].\mathit{val}//\mathit{uptodateReg} \\ \quad \textbf{for } \mathit{auxLastIssuedRS}[j]{=}i \wedge \mathit{auxLastWriterRS}[j]{=}k \wedge r[j]{=}c;$$

That is, we let i be the last writer to register j in program order, k the last writer in the implementation, and c the correct data value. In this case there are only two distinguished tag values, i and k, so the abstraction contains only two RS's.

In fact, the first attempt to check this property produced a counterexample in which some abstracted instruction causes a flush, cancelling the instruction that should write register j. The abstract model allows this because the states of RS's other than i and k are unknown. To deal with this, we introduce a *non-interference* lemma, stating that no unshadowed instruction is flushed:

$$\textbf{forall}(i \textbf{ in } \mathit{TAG})\ \mathit{lemma5}[i] : \textbf{assert G} \\ \quad (\mathit{flush} \Rightarrow \mathit{shadow} \wedge (\mathit{complete\_st}{\neq}i \Rightarrow \neg(st[i].\mathit{valid} \wedge \neg \mathit{aux}[i].\mathit{shadow})));$$

Here, *complete_st* is the tag of the RS causing the flush. We prove this by splitting cases on the flushing instruction. This eliminates the above counterexample to the up-to-date register property, leaving another counterexample in which a shadowed instruction writes register j and corrupts its value. This calls for another lemma stating that no shadowed instruction retires:

$$\mathit{lemma6} : \textbf{assert G } (\ \mathit{retiring} \Rightarrow \neg \mathit{aux}[\mathit{complete\_st}].\mathit{shadow});$$

This can be proved by splitting cases on the currently retiring instruction and the instruction that set the *shadow* bit (*e.g.* a mispredicted branch). That is, the latter must retire and cause a flush before the shadowed instruction can retire. With this additional lemma, the up-to-date register property is verified. Now operand correctness is easily proved by splitting cases on the source register and the operand's tag, which indicates the data source when forwarding from the result bus:

$$\textbf{subcase } \mathit{lemma1}[i][j][c] \textbf{ of } st[k].\mathit{opra.val}//\mathit{lemma1} \\ \quad \textbf{for } st[k].\mathit{opra.tag} = i \wedge \mathit{aux}[k].\mathit{srca} = j \wedge \mathit{aux}[k].\mathit{opra} = c;$$

The specification for results returning from execution units can be verified using operand correctness. This requires a non-interference lemma stating that unexpected results are never returned.

4.3 Verifying memory data correctness

We also specify the the results returning from the data memory, as follows:

$$\textit{lemma4} : \textbf{assert G} \; (\; \neg \textit{mqaux}[\textit{mq\_head}].\textit{shadow} \wedge \textit{mem\_ld} \wedge \textit{mem\_enable} \\ \wedge \; \textit{load\_from\_mem} \Rightarrow \textit{mem\_rd\_data} = \textit{mqaux}[\textit{mq\_head}].\textit{data});$$

Here, *mq_head* points to the currently executing operation in the load-store queue. That is, if the current operation is an unshadowed load, then the data from memory are the correct data stored in the auxiliary array *mqaux*. We break this into two cases – when data are read from memory and when data are forwarded from the load-store queue. Here we consider only the former case, although the latter is similar.

This property is similar to the one specifying values read from the register file. Here, we must prove that, for any load, the most recently executed store to the same address (call it S_E) is also the most recent in program order (call it S_P). As before, we use auxiliary variables to identify S_E and S_P in the queue. Splitting cases on these two stores and the current load, we should be able to prove that S_E and S_P are the same, hence read data are correct.

Unfortunately, the abstract model with two stores and one load is too large to model check. We cannot solve this problem as before by writing an "up-to-date" lemma for the memory, since we may read the memory when it is not up-to-date. Instead, we split cases only on the current load L and on S_E. This produces a counterexample in which $S_E < S_P < L$ in program order. That is, at the time L occurs, S_E has executed but S_P has not. This cannot really happen, because the unexecuted store S_P would block load L. However, since S_P is abstracted, this information is lost. To avoid splitting cases on S_P, we simply state as a lemma that $S_P \leq S_E$. In SMV, we say:

$$\textit{lemma4a} : \textbf{assert G} \; (\; \neg \textit{mqaux}[\textit{mq\_head}].\textit{shadow} \wedge \textit{mem\_ld} \wedge \textit{mem\_enable} \\ \wedge \; \textit{load\_from\_mem} \Rightarrow (\textit{imtag}[\textit{mem\_addr}] \geq \textit{mqaux}[\textit{mq\_head}].\textit{lastWrite} \;));$$

Here $\textit{imtag}[\textit{mem\_addr}]$ is S_E, while $\textit{mqaux}[\textit{mq\_head}].\textit{lastWrite}$ is S_P. This can be proved using another lemma, stating that stores always occur in program order. All three properties can be proved using just two memory queue elements. We reduce the problem further by writing a refinement relation for the data in the load-store queue. This allows us to abstract out the RS's when proving memory properties. This required a lemma stating that unshadowed queue elements are never flushed, which follows directly from the fact that unshadowed RS's are never flushed. The resulting abstract models can be handled easily by the model checker. At the cost of additional lemmas, we have reduced an intractable problem to a tractable one.

4.4 Remaining steps

For the program counter (PC), we write a refinement relation stating that, when the *shadow* bit is not set, the implementation PC equals the reference model PC:

$$\textbf{layer} \; \textit{opok} : \textbf{if}(\neg \textit{shadow}) \; \textit{ipc} := \textit{pc};$$

Since the PC can be loaded from an RS (in case of a flush) or from a register (for a JMP), we split cases on the most recent reservation station to and on the source register of the previous instruction. We also use the two lemmas about

Model	Proof size
A (baseline)	5700 bytes
B = A + out-of-order	7000 bytes
C = B + speculation	13K bytes
D = C + load-store buffer	18K bytes

Table 1. Proof size *vs.* feature set.

speculation. Further refinement relations specify the decoded instruction and branch target. This isolates the uninterpreted functions computing these values.

Finally, we must prove our overall correctness criterion, correctness of outputs. The OUT instruction reads a register and sends its value to the output port. Thus, the up-to-date register property suffices to prove output correctness. Overall, the proof[4] consists of the following elements: (1) refinement maps for the program counter, instruction decoder, register file, RS's and load-store queue, (2) two non-interference lemmas for speculative execution, two for the result bus, and four for the load-store queue (3) case splitting instructions for the above and hints for adjusting the abstractions, and (4) auxiliary variable declarations. All told, this information comprises less than 18K bytes, somewhat *less* than the size of the microarchitecture model and its specification.

To summarize, our strategy is to reduce the verification problem "units of work", in this case instructions. Since each instruction uses only finite resources, we can verify its correctness using a finite abstraction of the system. We identify the resources used by the instruction (*e.g.* RS's, registers, *etc.*), by introducing auxiliary variables. Once we "split cases" on these resources, the pointer types and arrays are automatically reduced, yielding a finite abstract model.

The novel aspects of this proof are in the treatment of speculation, and of read/write hazards. We handled speculation by introducing an auxiliary *shadow* bit for each instruction in the machine. We then show two key facts about the system: that unshadowed instructions are never canceled, and that shadowed instructions never retire. To handle read/write hazards, we use an abstraction strong enough to prove that the most recent writes to an address in execution and program order are the same.

Finally, to address the question of scalability, we consider four designs of increasing complexity: design A is a simple in-order processor, design B adds Tomasulo's algorithm for out-of-order execution, design C adds speculative execution and design D adds a load-store buffer. Table 1 shows the textual size of the proofs we obtained for these four designs. Adding Tomasulo's algorithm is the simplest step, involving only a few additional case splits and two non-interference lemmas. Adding speculation and the load-store buffer is more complex, because of the register and memory ordering properties we must prove. Nonetheless, we find that the complexity of the interactions between these features does not make the proof intractable. Rather, the proof increment associated with adding a feature remains moderate, at least for this example.

[4] By "proof", we mean all the input used to guide a mechanical prover, and not a proof in the mathematical sense.

5 Comparison with Other Approaches

We now compare our proof with proofs of similar microarchitecture models using other methods. We consider proofs by Sawada and Hunt [SH98], Velev and Bryant [VB00] and Hosabettu *et al.* [HGS00]. All of these proofs are variations in some form on the method of Burch and Dill [BD94], in which an abstraction function is constructed by "flushing" the implementation, *i.e.*, inserting null operations until all pending instructions are completed. This yields a "clean" state which can be compared to the reference model state. One then proves a *commutative diagram*, that is, that taking one implementation step and then applying the abstraction function yields the same state as applying the abstraction function followed by zero or more reference model steps. This can be done in an almost fully automated way for simple pipelines, and has the advantage that the abstraction function is mechanically constructed.

However, the method has two distinct disadvantages. First, for complex architectures, the abstraction function is generally not strong enough to be inductively invariant. It must be manually strengthened with information about reachability of control states. In our method, no such information is required. Second, the the abstraction function depends on the entire machine state, including all the instructions that are currently in the machine. For complex architectures, it becomes intractable to deal with it automatically. In our method, we reason about only one or two instructions. Thus, the proof obligations are local, and can be handled by model checking. By contrast, most recent work using abstraction functions manually decomposes the flushing function into smaller, more tractable parts. Thus the Burch and Dill method's advantage of full automation is lost. To see this, we consider the extant proofs in more detail. A comparison of textual sizes of models and proofs is given in table 2.

Sawada and Hunt The work of Sawada and Hunt [SH98] is perhaps the first formal proof a "modern" microprocessor architecture. Their processor model uses Tomasulo's algorithm, branch prediction, precise exceptions and a load store buffer with forwarding. The model is qualitatively similar to ours, with a few differences. They model asynchronous interrupts, while we do not. They use a fixed set of execution units (one per instruction type) while we do not. Thus, they associate RS's statically with execution units, while we choose the execution unit at issue time, to maximize use of the execution units. Also, their load-store buffer holds two loads and one store, while we model an arbitrary number of entries.

The model is defined by a collection of Common LISP functions in the theorem prover ACL2 [KM96]. We report in table 2 the approximate textual size of the functions describing the processor architecture, excluding theorems and generic functions not related to processor modeling. This is roughly three times the textual size of our model in the SMV language. In our estimation, this difference is largely accounted for by the greater conciseness of the SMV language as a hardware description language. However, some details present in the Sawada and Hunt model, such as an explicit instruction decoding function, are not present in our model, since we model them generically using uninterpreted functions. Defining these functions explicitly would increase the description size, but would not affect the proof.

Sawada and Hunt use an intermediate abstraction called a MAETT, a table tracking of the status of all instructions being executed in the machine. They then relate the MAETT to the implementation and the reference model using invariants, which are proved by induction. We do not use an intermediate abstraction, although our auxiliary variables do contain information similar to that in the MAETT. The chief difficulty reported by Sawada and Hunt is that the invariant must be strengthened by auxiliary invariants of the implementation state. No such invariants occur in our proof (although we do need a few lemmas concerning which events may occur in certain states). This leads to a stark difference in the textual size of the proofs: their proof (for the FM9801 processor) is roughly 1909K bytes, of which nearly a megabyte is the inductive invariant. Our proof is less that 20K bytes, smaller than the model description itself. This difference of two orders of magnitude is more than enough to account for differences in models, the succinctness of representation, whitespace, *etc.* By another measure, the Sawada and Hunt proof has roughly 4000 lemmas, whereas ours has approximately 18 (depending on how one counts).

Velev and Bryant The approach of Velev and Bryant [VB00] is closely based on the Burch-Dill technique. They focus on efficiently checking the commutativity condition for complex microarchitectures by reducing the problem to checking equivalence of two terms in a logic with equality, and uninterpreted function symbols. Under certain conditions, their decision algorithm is able to check equivalence of the massive formulas obtained from flushing complex models. Some manual work is required, however, to put the problem in a form suitable for the tool. They handle architectures with deep and multiple pipelines, multiple-issue, multi-cycle execution units, exceptions and branch prediction, for fixed finite models (note, we treat models with unbounded resources). Notably, they do not treat out-of-order execution, or load-store buffers. We conjecture that this is due to the complexity of the flushing functions, and the need for complex auxiliary invariants in these cases.

Hosabettu *et al.* Hosabettu *et al.* have published a series of papers on microprocessor verification, based on the "completion functions" approach. The microarchitecture they model in [HGS00] is similar to ours in that it has out-of-order execution, branch prediction, precise exceptions and it buffers stores (but not loads, which are atomic). Stores are executed in program order, while in our model they can be out-of-order. Also, they model a processor status word, while we do not.

Hosabettu *et al.* prove a commutative diagram, but decompose the abstraction function into *completion functions* for each instruction in the machine. A completion function specifies the future effect of an unfinished instruction on the observable state. They define completion functions for each instruction type, in terms of the present status of the instruction in the machine, and also whether that instruction will *squash* subsequent instructions, ensuring they do not affect the program state. The abstraction function is the composition of the comple-

Technique Used	Proof Assistant	Size of Machine Spec	Size of Proof
Sawada & Hunt [SH98]	ACL2	˜60K bytes	1909K bytes
Hosabettu *et al.* [HGS00]	PVS	˜70K bytes	˜2300K bytes
Compositional Model Checking	SMV	20K bytes	18K bytes

Table 2. Textual sizes of the Models and Proofs

tion functions. A commutative diagram is proved using PVS [ORSvH95] for the decomposed abstraction function.

This approach has the advantage of avoiding applying a decision procedure to the entire flushing function. However, proofs of the commutativity obligations require auxiliary invariants that characterize the reachable states of the model. To reason about the composite abstraction function, one must enumerate manually the various instructions in a particular state, the exact transitions they might make, the position of the "squashing" instruction, and so on. While decomposing the abstraction function makes reasoning about each case simpler, considerable manual effort is still required in stating invariants and guiding the prover.

The authors report that the proof took much less time than that of Sawada and Hunt. However, the textual size is comparable. The proof uses approximately 300K bytes of PVS specifications, and 2000K bytes of proof script (manual prover guidance). The latter, while generated manually, contains considerable redundancy. Thus its large size may not accurately reflect the effort needed to create it. We conjecture the large proof size results from the need for auxiliary invariants, and the theorem prover's greater need for manual guidance *vis-à-vis* model checkers.

6 Conclusion

We have shown that compositional model checking methods can verify a processor microarchitecture with most of the architectural features of a modern microprocessor. We introduced proof strategies to handle speculative execution (using shadow bits) and to handle read/write hazards (case splitting on the most recent writes in program and execution order). The proof methodology scales well in that the incremental proof cost associated with each processor feature is low. Moreover, the proof is concise relative to proofs using other methods (and is smaller than the model description itself). Although proof size is not necessarily an indication of the human effort required, we consider the difference of two orders of magnitude to reflect a qualitative difference in proof complexity. We ascribe this difference to several factors.

First, as reported both by Sawada and Hunt and by Hosabettu *et al.*, one of the most time consuming aspects of their methods is specifying auxiliary invariants. We exploit the model checker's ability to compute reachable states to avoid writing such invariants. Second, by stating refinement relations as temporal properties we can decompose the proof into "units of work", such as instructions, that are temporally and spatially distributed but use finite resources. This avoids reasoning about the entire state of the machine, and allows us to use small, finite-state abstractions. Finally, we exploit the fact that model checkers require less manual guidance than theorem provers do.

Nonetheless, there remains much room for improvement. For example, some lemmas in our proof could be eliminated if the model checker were able to handle three instructions in the abstraction instead of two. We have found that the symbolic model checker can handle abstract models with only about half the number of state bits that can be handled with concrete models. The reason for this is unclear, though it may be that the abstract state spaces are less sparse, or that there is greater nondeterminism in the transition relation. This does not affect the scalability of the proof methodology, but the "constant factor" would be improved if the model checker could handle larger abstract models.

To handle asynchronous interrupts, it would be useful to implement "prophecy variables", so that the witness function that stalls the reference model could depend on the future of the implementation. Also, to implement a specific instruction set architecture, we must substitute concrete functions for the uninterpreted functions in our model. Support for this is currently lacking in the prover, though it would be straightforward to implement.

On the whole, although proofs of this sort are considerably more laborious than model checking finite state machines, we feel that the methodology scales well, and that additional processor features, such as a first-level cache, an address translation unit, or multiple-issue could be handled in a straightforward manner, with the addition of a few lemmas for each feature.

References

[AP99] T. Arons and A. Pnueli. Verifying tomasulo's algorithm by refinement. In *12th Int. Conf. on VLSI Design (VLSI'99)*, pages 306–9. IEEE Comput. Soc., June 1999.

[BD94] J. R. Burch and D. L. Dill. Automated verification of pipelined microprocessor control. In D. L. Dill, editor, *Computer-Aided Verification (CAV94)*, LNCS 818, pages 68–80. Springer-Verlag, 1994.

[HGS00] R. Hosabettu, G. Gopalakrishnan, and M. Srivas. Verifying advanced microarchitectures that support speculation and exceptions. In E. A. Emerson and A. P. Sistla, editors, *Computer-Aided Verification (CAV2000)*, LNCS 1855, pages 521–37. Springer-Verlag, 2000.

[KM96] M. Kaufmann and J. S. Moore. ACL2: An industrial strength version of Nqthm. In *Conf. on Computer Assurance (COMPASS-96)*, pages 23–34. IEEE Comp. Soc. Press, 1996.

[McM00] K. L. McMillan. A methodology for hardware verification using compositional model checking. *Sci. of Comp. Prog.*, 37(1–3):279–309, May 2000.

[ORSvH95] S. Owre, J. Rushby, N. Shankar, and F. von Henke. Formal verification for fault tolerant architectures: Prolegomena to the design of PVS. *IEEE Trans. on Software Eng.*, 21(2):17–125, Feb 1995.

[SH98] J. Sawada and W. D. Hunt. Processor verification with precise exceptions and speculative execution. In A. J. Hu and M. Y. Vardi, editors, *Computer-Aided Verification (CAV98)*, LNCS 1427, pages 135–146. Springer, 1998.

[Tom67] R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM J. of Research and Development*, 11(1):25–33, Jan. 1967.

[VB00] M. Velev and R. E. Bryant. Formal verification of superscalar microprocessors with multicycle functional units, exceptions and branch prediction. In *37th Design Automation Conference (DAC 2000)*. IEEE, June 2000.