Deductive Verification of Advanced Out-of-Order Microprocessors*

Shuvendu K. Lahiri and Randal E. Bryant

Carnegie Mellon University, Pittsburgh, PA
shuvendu@ece.cmu.edu, Randy.Bryant@cs.cmu.edu

Abstract. This paper demonstrates the modeling and deductive verification of out-of-order microprocessors of varying complexities using a logic of Counter Arithmetic with Lambda Expressions and Uninterpreted Functions (CLU). The microprocessors support combinations of out-of-order instruction execution, superscalar operation, branch prediction, execute and memory exceptions, and load-store buffering. We illustrate that the logic is expressive enough to model components found in modern processors. The paper describes the challenges in modeling and verification with the addition of different design features. The paper demonstrates the effective use of automatic decision procedure to reduce the amount of manual guidance required in discharging most proof obligations in the verification. Unlike previous methods, the verification scales well for superscalar processors with wide dispatch and retirement widths.

1 Introduction

In the last few years, several different techniques have been employed for the formal verification of advanced microprocessors. These include the use of symbolic model checking [3], compositional model checking [10], deductive verification methods based on theorem proving [1, 9, 13] and symbolic simulation with decision procedures for quantifier-free first order logic [5, 7].

Most of the previous efforts in verifying microprocessors with unbounded resources (e.g., with an arbitrarily large reorder buffer or load-store queue) are based on deductive verification methods and involve a general purpose theorem prover (PVS [12], ACL2 [4]) to discharge the proof obligations in the verification. This involves writing down large proof scripts to systematically prove the invariants. This is both time-consuming and requires very careful understanding of the theorem provers. Moreover, the lack of counter-examples for failed proofs renders the invariant strengthening method difficult and relies on the ingenuity of the user.

In earlier work [11], we used the logic of Counter Arithmetic with Lambda Expressions and Uninterpreted Functions (CLU), to model and verify a simple out-of-order execution unit. Deductive verification was used to prove the correctness of the processor by establishing a set of *refinement maps* [1, 10], to show that

* This research was supported in part by the Semiconductor Research Corporation, Contract RID 1029.001.

the implementation refines the specification. The refinement maps specify the correctness of signals or values in the implementation with respect to a sequential Instruction Set Architecture (ISA) model. All the proof obligations were discharged automatically using sound quantifier instantiation techniques and the decision procedure for CLU. Further, the use of the restricted logic enabled us to produce counterexamples for the failed proofs.

In this work, we investigate if the logic of CLU is expressive enough to model and verify out-of-order processors with advanced features such as speculation, superscalar behavior and buffered memory instructions. We show how the addition of new instruction types and design features add to the modeling and verification challenges using CLU. Unlike prior work in the verification of processors with unbounded resources, we demonstrate the verification of superscalar processors, where multiple instructions can be dispatched and retired simultaneously.

Category	Unbounded Resources	Speculation, Exceptions	Data Memory	Methodology
Sawada and Hunt. [13]		×	×	Deductive verification with ACL2
Skakkebaek et al. [14]	×			Correspondence checking with manual abstraction
Berezin et al. [3]	×			Finite State Model Checking
Arons et al. [1]	×	×		Deductive Verification with PVS
Hosabettu et al. [9]	×	×	×	Deductive Verification with PVS
Jhala, McMillan [10]	×	×	×	Compositional Model Checking
Velev [15]				Correspondence Checking
Lahiri et al. [11]	×			Deductive Verification with UCLID
Current	×	×	×	Deductive Verification with UCLID

Fig. 1. Previous efforts for out-of-order processor verification.

Related Work. Figure 1 shows a chronological listing of different approaches to out-of-order processor verification along with the features of the processors verified. In the next few paragraphs, we concentrate on previous works that verify an out-of-order processor supporting speculation, exceptions and memory instructions with load-store queues.

Jhala and McMillan [10] use compositional model checking (with Cadence SMV) to verify refinement maps between an out-of-order processor and a sequential ISA model. A finite state abstraction is generated by exploiting temporal case-splitting, data-type reduction and symmetry. Model checking is then used on the abstract state space to verify the properties. Although more automatic than the deductive verification based approaches, the method still requires the user to explicitly decompose the proof into smaller lemmas to alleviate the state explosion. Besides, the approach relies heavily on symmetry in the system. It can be ineffective for many practical systems, where symmetry is broken by the presence of priority encoders (e.g. processors with wide dispatch and retire widths) or the heterogeneity of deep pipelines. Since our verification does not explicitly use symmetry, our approach is robust in the presence of asymmetry in the design as we demonstrate in the verification of the superscalar processors.

Deductive verification based methods [13, 9, 1] use general purpose theorem provers to establish the correctness of microprocessors. Sawada and Hunt [13] use the ACL2 theorem prover to verify the correctness of microprocessors with

bounded retirement and load-store buffers. They limit at most 15 instructions in the pipeline at any time. They use a trace-table based intermediate representation called MAETT to record redundant information for committed and in-flight instructions. This intermediate abstraction is used to specify invariants to relate the ISA and the pipelined implementation. It should be mentioned that they model external interrupts, which no one else (including our current models) handle at present. On the other hand, our work considers unbounded resources and requires significantly fewer lemmas compared to the almost 4000 lemmas in Sawada and Hunt's case. Hosabettu et al. [9] use a *completion function* approach to complete all the partially executed instructions in the system. The "flushed" state of the implementation is then compared against the ISA model. The method requires the user to construct an inductive completion function for the different instruction types (in different stages of execution) and then compose the different completion functions to obtain the abstraction function. The PVS [12] theorem prover is used to discharge the proofs. Both Sawada et al. and Hosabettu et al. use a variant of Burch-Dill method of "flushing" the pipeline and prove the equivalence of an empty pipeline with the ISA state. Our approach differs from these methods in two ways. First, we use refinement maps (similar to Arons et al. [1]) between the implementation and the sequential ISA to prove the correctness of the processor. Second, the use of decision procedure reduces the burden of proving most of the proof obligations that arise during the verification.

The rest of the paper is organized as follows. In Section 2, we provide a brief overview of the tool UCLID. The section outlines the logic and the method of verification. In Section 3, the description and modeling of the different out-of-order processors are presented. Finally, in Section 4, the verification is described. This includes the definition of different auxiliary fields, the refinement maps, description of invariants and their proofs.

2 Background

The tool UCLID [6, 11] uses the logic of CLU (described in Fig 2) to model and verify systems with unbounded resources. CLU is a fragment of quantifier-free first order logic extended with increment (**succ**), decrement (**pred**), equality and inequality operations over *terms* (integer expressions). *ITE* denotes the "if-then-else" constructor to choose between two terms depending on a boolean control. The uninterpreted function and predicate symbols can be used to specify an arbitrary value for the function state variables in the most general state or to abstract out combinational blocks [5, 10] (e.g., the ALU) in the system.

The presence of lambda expressions not only subsumes the interpreted **read** and **write** operators for unbounded arrays [6] but also allows us to model *parallel-update* memories. In a parallel-update memory M, an arbitrary number of entries satisfying some predicate (say P) can get updated (with some function D) in one step as follows:

$$M' = \lambda\, i\, .\, ITE(P(i),\, D(i),\, M(i))$$

Here M' denotes the next state of the memory M. This is very important for modeling the forwarding of result data to an arbitrary number of dependent

$$
\begin{aligned}
\textit{bool-expr} ::= {} & \mathbf{true} \mid \mathbf{false} \mid \neg \textit{bool-expr} \mid (\textit{bool-expr} \wedge \textit{bool-expr}) \\
& \mid (\textit{int-expr} = \textit{int-expr}) \mid (\textit{int-expr} < \textit{int-expr}) \\
& \mid \textit{predicate-expr}(\textit{int-expr}, \ldots, \textit{int-expr}) \\
\textit{int-expr} ::= {} & \textit{int-var} \mid \textit{ITE}(\textit{bool-expr}, \textit{int-expr}, \textit{int-expr}) \\
& \mid \mathbf{succ}(\textit{int-expr}) \mid \mathbf{pred}(\textit{int-expr}) \\
& \mid \textit{function-expr}(\textit{int-expr}, \ldots, \textit{int-expr}) \\
\textit{predicate-expr} ::= {} & \textit{predicate-symbol} \mid \lambda\, \textit{int-var}, \ldots, \textit{int-var}\,.\, \textit{bool-expr} \\
\textit{function-expr} ::= {} & \textit{function-symbol} \mid \lambda\, \textit{int-var}, \ldots, \textit{int-var}\,.\, \textit{int-expr}
\end{aligned}
$$

Fig. 2. CLU Syntax. Expressions can denote computations of Boolean values, integer values, or functions overs integers yielding Boolean or an integer value.

instructions in an out-of-order processor [11]. The **succ** and **pred** operations allow us to model ordered data structures such as queues of arbitrary length, by performing appropriate increment or decrement operations for the head and tail pointers for the queue. Various other data structures including content-addressable memories, and circular queues can be expressed in CLU [11].

A well-formed CLU formula[1] F is *valid* (denoted as $\models F$) when it is true under all possible interpretations of symbols in F. The *decision procedure* for CLU checks the validity of a well-formed formula F by a validity-preserving translation to a propositional formula and using Boolean techniques to evaluate the formula. Counterexamples for invalid formulas are mapped to the state-variables to produce counter-example traces. Details about the logic and decision procedure can be found in earlier work [6].

Deductive Verification. A system is verified by proving a set of invariants inductively. Let $\Upsilon_1(s), \ldots, \Upsilon_n(s)$ be a set of invariants on a state s of the system. To prove the base case, we show:

$$\models \Upsilon_1(s_0) \wedge \Upsilon_2(s_0) \ldots \wedge \Upsilon_n(s_0) \tag{1}$$

where s_0 is the start (reset) state of the system. For the induction step, we start with a general state s and then symbolically simulate the system for one step to obtain state $\delta(s)$, where δ is the transition function. We then show:

$$\models \Upsilon_1(s) \wedge \ldots \wedge \Upsilon_n(s) \implies \Upsilon_i(\delta(s)) \tag{2}$$

for each $1 \leq i \leq n$. Usually, the property to be verified is specified as one of these invariants. The other invariants are added (manually) to *strengthen* the inductive invariant.

Similar to our previous work [11], we restrict the invariants to be of the form $\forall x_1 \ldots \forall x_k . \Phi(x_1, \ldots, x_k)$, where $x_1, \ldots, x_k$ are integer variables *free* in the CLU formula $\Phi(x_1, \ldots, x_k)$. To prove that such an invariant is inductive (in Equation 2), we need to decide formulas of the form

$$\models \forall x_1 \ldots \forall x_m \Psi(x_1, \ldots, x_m) \implies \forall y_1 \ldots \forall y_k \Phi(y_1, \ldots, y_k) \tag{3}$$

[1] An integer variable x is said to be *bound* in expression E when it occurs inside a lambda expression for which x is one of the argument variables. An expression is *well-formed* when it contains no unbound variables.

Since checking validity for first-order formulas of the form (3) is undecidable [8], we perform a sound translation of the formula in (3) to a CLU formula, which can be checked by a decision procedure.

The reduction of the formula in Equation 3 involves replacing the universal quantifiers to the right of the implication with fresh *skolem constants* $\widehat{y}_1, \ldots, \widehat{y}_k$. The antecedent of the implication, Ψ is instantiated over terms appearing in the consequent Φ. Details of the quantifier instantiation can be found in earlier work [11]. If $\mathcal{A}_{x_i}$ denote the set of terms to instantiate the variable x_i, then the final formula becomes:

$$\left[\bigwedge_{t_1 \ldots t_m \in \mathcal{A}_{x_1} \times \ldots \times \mathcal{A}_{x_m}} \Psi(t_1, \ldots, t_m) \right] \implies \Phi(\widehat{y}_1, \ldots, \widehat{y}_k) \tag{4}$$

3 OOO Processor Description

OOO (shown in Figure 3) is a model of an out-of-order processor that employs Tomasulo's algorithm, supports speculative execution, exceptions, memory instructions and load-store queues. On each step, the system arbitrarily chooses

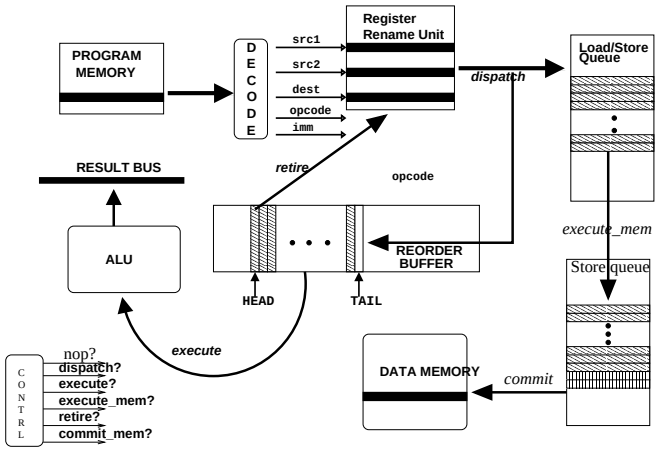

Fig. 3. An Out-of-order processor with exceptions, speculation and memory instructions.

between six different operations: *dispatch*, *execute*, *execute_mem*, *retire*, *commit_mem* and *nop*. During *dispatch*, an instruction is dispatched to the Reorder Buffer(ROB). An entry is also created in the Load-Store Queue (LSQ) if it is a memory instruction. During *execute*, a ready arithmetic or branch instruction in the ROB (with all operands valid) is scheduled for execution. During *execute_mem*, the memory instruction at the head of the LSQ is executed. Load instructions can obtain their data from the preceding store instructions when there is an address match in the Store-Queue (STQ). Otherwise the data comes from data memory. The store instructions are enqueued into the STQ after execution with the store address and the data. During *retire*, the instruction at the

head of the ROB is retired and the register state is updated. For store instructions, the entry in STQ is marked as *retired*. The actual memory update takes place during a *commit_mem* operation.

Instructions are retired in program order to track precise exceptions. An exceptions can be raised as a result of either arithmetic instruction execution, an illegal data address for a memory instruction or an illegal address for a branch instruction. If the currently retiring instruction raises an exception, the ROB and LSQ are flushed. All of the non-retired entries in the STQ are also dropped and all of the registers in the register file set the `reg.valid` bit to **true**.

The main components of the design are (i) a Reorder Buffer for inorder completion and precise exceptions, (ii) a Register Rename unit for out of order execution, (iii) a Load-store queue for non-executed memory instructions, (iv) a Store-Queue for stores which have finished execution. Below we describe the modeling of each of them in some detail.

Reorder Buffer. The reorder buffer (ROB) is modeled as a queue with head and tail pointers. It also supports simultaneous update of a subset of entries if an operand tag matches the currently executing instruction. Each entry in the ROB contains fields for the instruction type `rob.itype`, opcode `rob.opcode`, destination register `rob.dest`, immediate value `rob.imm`, operand values `rob.src1val`, and `rob.src2val`, program counter `rob.pc`, prediction flag `rob.mispredict` and target address `rob.target`. To indicate if the operands are ready, it maintains `rob.src1valid` and `rob.src2valid` bits and fields `rob.src1tag` and `rob.src2tag` indicating the instructions producing the data. The `rob.valid` bit indicates the instruction has finished execution. The field `rob.value` stores the write-back value for arithmetic and load instructions.

Register Rename Unit. The register rename unit consists of an infinite array of ⟨`reg.valid`, `reg.val`, `reg.tag`⟩ tuples. Every time an arithmetic or load instruction is dispatched with destination register d, the `reg.valid` bit for d is set to **false** and the tag for the dispatched instruction is recorded in the `reg.tag` field. A retiring instruction sets the `reg.valid` bit to **true** if its tag matches the tag stored in the register. A retiring mispredicted branch or an instruction with an exception, causes all of the registers to simultaneously reset their `reg.valid` bits to **true**.

Load-Store Queue. The load-store queue (LSQ) maintains the non-executed memory instructions in program order. Memory instructions are enqueued in the LSQ during *dispatch* and dequeued during *execute_mem*. Each LSQ entry contains a pointer to the corresponding ROB entry, called `lsq_rob_ptr`. The result of execution for a load instruction is written back in the `rob.value` field of the ROB entry pointed to by the `lsq_rob_ptr`. For store instructions, the data in `src2val` of the corresponding ROB entry and the address is enqueued into the STQ.

Store Queue. The store queue (STQ) maintains the store instructions which have finished execution, in program order. A pointer `stq_retire_ptr` points to the *first* non-retired instruction in the STQ. During a squash operation (due to

an exception or misspeculation for the retiring instruction), all of the non-retired instructions in the STQ are abandoned.

Since the system supports forwarding data from a preceding store instruction to a dependent load, there is a need to (i) find if an address is present in the STQ and (ii) identify the latest entry with the matching address. Since an associative lookup can't be modeled directly for infinite structures in UCLID, we maintain a map structure, `stq_pos`, which maps each memory address to the position of the *latest* entry (if present) in the STQ. An address A is present in the STQ iff $(stq\_head \leq stq\_pos(A) < stq\_tail) \wedge (stq\_addr(stq\_pos(A)) = A)$.

This way of modeling the associative lookup however has one problem. During a squash operation, the non-retired instructions in the STQ are abandoned. This would require resetting `stq_pos` for a given address A to the *latest* retired entry in the STQ which matches the address A (if present). To circumvent this problem, we maintain another map `stq_nonspec_pos` which maps an address A to the latest retired entry in the STQ, which matches A (if present). It is updated every time a store instruction is retired. During squash, the map `stq_nonspec_pos` is copied onto `stq_pos`. Note that this map is not required if stores are committed to memory during retire.

4 Verification of the Processors

We establish refinement maps to prove the correctness of the implementations with respect to a sequential instruction set architecture (ISA). The ISA state components consists of the register file, data memory and, program counter.

We built the models of the out-of-order processors incrementally to study the effect of different features on both the modeling and the verification effort. Figure 4 shows the different models and the features that were *added* with each model.

First, we describe the various auxiliary data structures and variables added and later express the correctness criteria for the augmented system.

4.1 Auxiliary Fields

We had to add a number of auxiliary variables to the system to enable the verification. These variables do not affect the system operation, but are added for two principal reasons, discussed below.

Model	Features
OOO.base	Out-of-order execution, inorder retirement
OOO.ex	Arithmetic exceptions
OOO.ex_br	Branch Prediction
OOO.ex_br_mem_simp	Memory instructions, LSQ, STQ, Stores commit during *retire*
OOO.ex_br_mem	Stores commit during *commit_mem*

Fig. 4. Description of different models.

First, we add auxiliary variables to express invariants about the correctness of values of variables in the system (similar to other works [1, 10]). These additional state variables, called *shadow* variables in our case (prefixed with `shdw.`), predict the correct value for some actual state variables. For instance, the shadow entry

`shdw.src1val` predicts the correct value for the state variable `rob.src1val` — for any index t in the ROB, if `rob.src1val(t)` contains a valid entry, then `rob.src1val(t)` = `shdw.src1val(t)`. The *shadow* variables for an instruction are usually updated by the ISA machine during *dispatch* [11].

Second, we need to add additional variables to express an invariant of the form $Q_1 x_1, \ldots, Q_n x_n . \Phi(x_1, \ldots, x_n)$, where $\Phi(x_1, \ldots, x_n)$ is a CLU formula and at least one of the Q_i is $\exists$. For instance, consider the invariant — *for every non-executed memory instruction t in the* ROB*, there exists a "corresponding" entry in the* LSQ. This is hard to express without existential quantifiers. We maintain a pointer `aux.rob_lsq_ptr` for each ROB entry to point to the corresponding entry in the LSQ. We can restate the invariant as — *for every non-executed memory instruction t in the* ROB, `aux.rob_lsq_ptr(t)` *is present in the* LSQ. The variables in this category are prefixed with `aux.`, e.g. (`aux.rob_lsq_ptr`). These variables essentially act as *witnesses* for the existential quantifiers.

Below we describe the auxiliary fields that were added with each design feature.

1. OOO.base. The auxiliary fields required were `shdw.src1val`, `shdw.src2val` and `shdw.value` for expressing the correct values of the data operands and result in each ROB entry.

2. OOO.ex, OOO.ex_br. First we maintained a pointer into the ROB, named `shdw.exn_mpred_tag` to keep track of the earliest instruction that will raise an exception or cause a misprediction. We also required a map `shdw.reg.tag`, which maps a register to the *latest nonspeculative* instruction in the ROB that modifies the register. Notice that in the presence of misprediction or exceptions, the instruction in the `reg.tag` may not be the last instruction to write into the register before flushing the system.

3. OOO.ex_br_mem_simp. The addition of load and store instructions required the addition of the largest number of auxiliary variables. First, with each ROB entry, we maintain two additional pointers, `aux.rob_lsq_ptr` and `aux.rob_stq_ptr` to point to an entry in the LSQ and STQ respectively. Second, we also add *inverse* pointer `aux.stq_rob_ptr` from a STQ entry to the corresponding ROB entry. Note that `lsq_rob_ptr` is already present in the actual model. Third, a map `shdw.mem_tag` is added which maps each memory address to the *latest* non-speculative instruction in the ROB which modifies the address. As we shall see later, this is required to express the refinement map for the data memory. Fourth, for every load instruction in the ROB, we maintain two pointers `shdw.ld_tag` and `aux.ld_stq_ptr`. The first pointer points to the store instruction (if any) in the ROB that would forward the data (due to an address match) to the load. The second pointer points to the STQ entry that forwards the data to the load.

4. OOO.ex_br_mem. We did not require any further auxiliary structure to prove this model.

4.2 Correctness via Refinement Maps

The correctness of the implementation is proved by establishing three refinement maps with the ISA model.

The correctness for the register file is established by the following lemma:

$$\forall r : ISA.rf(r) = \begin{cases} reg.val(r) & \text{if } \texttt{reg.valid(}r\texttt{)} \text{ is } \textbf{true} \\ - & \text{Otherwise} \end{cases}$$

The lemma states that if a register is not the destination of any of the instructions in the ROB, then the values in the implementation model and the ISA model are the same.

For the data memory, recall that `shdw.memtag` maps a memory address a to a position in the ROB (if present), containing the latest nonspeculative store instruction to write to address a. Thus, if there are no (non-retired) nonspeculative instructions in the ROB which modifies a and there are no (retired) instructions in the STQ which modifies a, then both the ISA memory and the implementation memory should have the same value for a. The refinement map for data memory can be stated as :

$$\forall a : ISA.mem(a) = \begin{cases} mem(a) & \text{if } \texttt{shdw.memtag(}a\texttt{)} \text{ is not present in ROB and} \\ & a \text{ is not present in the STQ} \\ - & \text{Otherwise} \end{cases}$$

Similarly, there is a refinement map for the program counter:

$$ISA.pc = \begin{cases} pc & \text{If } \texttt{shdw.exn\_mpred\_tag} \text{ is not present in ROB} \\ & \text{and } \texttt{squash} \text{ is } \textbf{false} \\ exn\_pc & \text{If } \texttt{squash} \text{ is } \textbf{true} \\ - & \text{Otherwise} \end{cases}$$

Note that `shdw.exn_mpred_tag` is present in the ROB iff some instruction in the ROB would raise an exception or cause misprediction. The signal `squash` is asserted after an exception is raised by the retiring instruction.

4.3 Invariants

In this section, we shall discuss the main categories into which we classified the invariants. The main categories of invariants are:

1. Consistency Invariants. These invariants express the relationship between the state variables of the actual system. These invariants can be stated without the help of auxiliary or shadow structures. For instance, the invariant that *every executed instruction has ready operands* can be stated without adding any auxiliary information to the model.

2. Ordering Invariants. Since the model contains three ordered data structures (ROB, LSQ, STQ), it is very important to maintain the program order of entries in the three queues. For example, if an instruction I_1 precedes another instruction I_2 in the LSQ, then I_1 should precede I_2 ROB as well.

3. Bijective Invariants. These invariants establish the relationship between two functions that act almost as *inverses* of each other. Examples include the pair of maps `aux.rob_stq_ptr` and `aux.stq_rob_ptr`. For any valid STQ index x, `aux.rob_stq_ptr(aux.stq_rob_ptr(x)) = x`. For any ROB index y with an executed store instruction, `aux.stq_rob_ptr(aux.rob_stq_ptr(y)) = y`.

4. Value Invariants. These invariants mainly specify the correctness of the data values in the model with reference to the *shadow* (predicted) values. For

instance, for an executed arithmetic instruction, the value in `rob.value` should equal the `shdw.value` field for the same ROB entry. Value invariants involving memory instructions are more involved.

5. PC Invariants. These invariants specify the primary and auxiliary invariants for the correctness of the program counter. The invariants relate the program counter (`pc`) and the exception program counter (`exn_pc`) with the program counter of the ISA model in the presence and absence of misprediction or exception.

6. Misprediction Invariants. These invariants state the relationship of the `shdw.exn_mpred_tag` with the instructions in the ROB. For example, if any instruction in the ROB would raise an exception or be mispredicted, then the `shdw.exn_mpred_tag` points to the earliest such instruction in the program order.

7. Register Tag Invariants. These invariants relate the `reg.tag` with the shadow entry `shdw.reg.tag`. For example, if none of the instructions in the ROB raises an exception or is mispredicted, then `reg.tag(r) = shdw.reg.tag(r)`, for any register r that would be modified by an instruction in the ROB.

8. Memory Tag Invariants. These invariants relate the different maps for the memory instructions, namely `shdw.mem_tag`, `shdw.ld_tag`, `aux.ld_stq_ptr`, `stq_pos`, `stq_nonspec_pos` and `stq_retire_ptr`. These are the most involved invariants for the verification of the processor models with memory instructions.

9. Other Invariants. These invariants cannot be categorized into one of the classes mentioned above. They were mostly obtained from failed proofs after analyzing the counterexamples.

Figure 5 illustrates the classification of the invariants into the different logical categories described above for each of the benchmarks. The purpose of the classification is to understand the complexity of invariants for different parts of the design. The number of value invariants grows dramatically when we introduce memory instructions in the model, due to the need to define the correct values for load instructions in the presence of store forwarding. Moreover, since we need the additional map `stq_nonspec_pos` for model `OOO.ex_mem_br`, we have more value and memory-tag invariants for this model compared to the model `OOO.ex_mem_br_simp`.

Technique	OOO.base	OOO.ex	OOO.ex_br	OOO.ex_br_mem_simp	OOO.ex_br_mem
Consistency	5	5	7	6	6
Ordering	-	-	-	6	6
Bijective	-	-	-	4	4
Value	6	7	7	10	13
PC	3	4	7	8	8
Misprediction	-	8	8	8	8
Register Tag	-	8	8	8	8
Memory Tag	-	-	-	15	16
Other	3	2	2	2	2
Total #	17	34	39	67	71

Fig. 5. Classification of the number of invariants for different models.

4.4 Proving the Invariants

The invariants are proved by the method described in Section 2. This method of instantiating the quantifiers with concrete sub-terms in the formula often generates the necessary terms to prove the valid formulas.

The number of combinations to instantiate can be exponential in the number of bound variables in the antecedent of Equation 3. The bound variables are indices to the different memories and unbounded arrays in the system. For the models without memory, the bound variables are register identifiers and ROB indices and the maximum number of combinations to instantiate was limited to 14 in these cases. With the introduction of LSQ and STQ, we needed upto 8 bound variables — to include LSQ and STQ indices and memory address. The number of combinations to instantiate increased up to 28800 for `OOO.ex_br_mem`. It is crucial to have a fast decision procedure for the large cases.

For most of the models, all the invariants were proved using the instantiation schemes described in Section 2. For the more complex models with multiple ordered structures, we had to resort to manual instantiation of a few invariants — 4 invariants for `OOO.ex_br_mem_simp` and 8 invariants for `OOO.ex_br_mem`. In most of these cases, at most two additional terms were needed. The terms were obtained easily by inspecting the counterexamples produced during the failed attempts.

Figure 6 summarizes the verification effort for the different models by several criteria. Note that the final models with the memory instructions are significantly more complicated to prove because of the large number of ordered data structures (LSQ and STQ in addition to ROB). Proving `OOO.ex_br_mem` is more complicated than `OOO.ex_br_mem_simp` because the presence of the retired instructions in the STQ[2]. Some of the criteria (# of invariants, person-effort) listed in Fig 6 are

Category	OOO. base	OOO. ex	OOO. ex_br	OOO. ex_br_mem_simp	OOO. ex_br_mem
# of invariants	17	34	39	67	71
Time Taken to Prove (sec)	54	235.76	403	1594.24	2200
UCLID Proof Script Size (KB)	9.91	20.06	23.59	68.67	66.79
Total Time Spent (Person Days)	2	5	2	15	10

Fig. 6. Proof Effort for different models. Time taken to prove denotes the time taken by UCLID to prove all the invariants inductive. Proof script size consist of the definition of invariants, auxiliary state variables and the proofs. The number of "Person Days" is the added effort for each model and is not cumulative.

very subjective and would differ from user to user depending on the knowledge of the design, dexterity with the tool or theorem prover and his/her ingenuity. But the ability to relieve the user from proving most of the proof obligations, results in a much smaller proof script size compared to previous attempts using a general-purpose theorem prover. For processors comparable to `OOO.ex_br_mem`, proof-script sizes for Sawada and Hunt [13] and Hosabettu et al. [9] are 2300KB and 1909KB respectively, as reported in [10].

[2] The proof-script size of `OOO.ex_br_mem` is smaller than `OOO.ex_br_mem_simp` because of some cleanups in the former scripts

One of the important contributions of this work is that we can use the automation and efficiency of the integer decision procedure for CLU to decide the large quantifier-free formulas. Previous attempts involving SVC [2], were unsuccessful, because of the rational interpretation of variables[3]. This produces numerous spurious counterexamples, since it does not properly model the integer semantics of the array indices.

4.5 Verifying Superscalar Processors

Previous attempts at the verification of out-of-order processors with unbounded resources consider processors that could only dispatch and retire a single instruction at each step [4]. Increasing the dispatch-width or the retirement width results in a more complex control logic for additional data forwarding. It also breaks the symmetry of entries in the reorder buffer, because of the explicit priority among the instructions in the dispatch-width. This reduces the effectiveness of approaches which use symmetry to reduce the complexity of the state space. Since our technique does not explicitly depend on symmetry, we can handle out-of-order processors with superscalar nature. To illustrate this, we generated a set of models with different dispatch and retirement widths on top of the processor model `OOO.base`. Unlike `OOO.base`, an arbitrary number of instructions can execute on any step. Moreover, dispatch, execute and retirement can occur concurrently instead of the interleaving model previously considered.

Width		max #	max	Time	
Dispatch	Retire	instant	prop-vars	Total	Conversion
1	1	10	439	58.69	53.90
1	2	28	682	93.56	84.98
1	4	88	1060	249.42	201.42
1	6	180	1433	470.44	362.63
1	8	304	1993	800.31	553.80
2	1	12	551	86.63	84.98
2	2	28	798	137.43	118.04
2	4	88	1152	308.55	232.46
2	6	180	1660	675.86	506.96
2	8	304	2098	1040.6	605.91

Fig. 7. Effect of processor width on verification. "Width" denotes the width of the processor. "max # instant" denotes the maximum number of instantiations to prove any invariant, "max prop-vars" is the maximum number of propositional variables in the boolean encoding of the formulas. The "conversion" component of the time is the time the decision procedure spends encoding a CLU formula to a boolean formula.

The verification of these superscalar processors proceeded automatically with the proof script for `OOO.base`. This is because the invariants express relationship between state variables and they are not affected by the change in control logic. As we see in Fig 7, the verification scales to large enough dispatch and retirement widths. It should be noticed that the number of terms to instantiate grows as

[3] Private Communication with Robert Jones

[4] Velev [15] considers large dispatch and retire widths for processors with bounded resources. But his technique based on rewriting is very specific to the model in the paper and is hard to extend to models with even register renaming.

we increase the width. This is because more instructions explicitly affect a single instruction and the instantiation has to account for all of these instructions. But the total time required to verify grows only linearly and can scale to larger superscalar width.

References

1. T. Arons and A. Pnueli. A comparison of two verification methods for speculative instruction execution. In *Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS '00)*, LNCS 1785, 2000.
2. C. Barrett, D. Dill, and J. Levitt. Validity checking for combinations of theories with equality. In *Formal Methods in Computer-Aided Design (FMCAD '96)*, LNCS 1166, November 1996.
3. S. Berezin, A. Biere, E. M. Clarke, and Y. Zhu. Combining symbolic model checking with uninterpreted functions for out of order microprocessor verification. In *Formal Methods in Computer-Aided Design(FMCAD '98)*, LNCS 1522. Springer-Verlag, November 1998.
4. R. S. Boyer and J. Moore. A theorem prover for a computational logic. In *10th Conference on Automated Deduction (CADE '90)*, 1990.
5. R. E. Bryant, S. German, and M. N. Velev. Processor verification using efficient reductions of the logic of uninterpreted functions to propositional logic. *ACM Transactions on Computational Logic*, 2(1):1–41, January 2001.
6. R. E. Bryant, S. K. Lahiri, and S. A. Seshia. Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions. In *Proc. Computer-Aided Verification (CAV'02)*, LNCS 2404, July 2002.
7. J. R. Burch and D. L. Dill. Automated verification of pipelined microprocessor control. In *Computer-Aided Verification (CAV '94)*, LNCS 818, pages 68–80, June 1994.
8. Y. Gurevich. The decision problem for standard classes. *The Journal of Symbolic Logic*, 41(2):460–464, June 1976.
9. R. Hosabettu, G. Gopalakrishnan, and M. Srivas. Verifying advanced microarchitectures that support speculation and exceptions. In *(CAV 2000)*, LNCS 1855, July 2000.
10. R. Jhala and K. McMillan. Microarchitecture verification by compositional model checking. In *Computer-Aided Verification*, LNCS 2102, July 2001.
11. S. K. Lahiri, S. A. Seshia, and R. E. Bryant. Modeling and verification of out-of-order microprocessors in UCLID. In *Formal Methods in Computer-Aided Design (FMCAD '02)*, LNCS 2517, pages 142–159. Springer-Verlag, Nov 2002.
12. S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In *11th International Conference on Automated Deduction (CADE)*, June 1992.
13. J. Sawada and W. Hunt. Processor verification with precise exceptions and speculative execution. In *Computer-Aided Verification (CAV '98)*, LNCS 1427, June 1998.
14. J. U. Skakkaebaek, R. B. Jones, and D. L. Dill. Formal verification of out-of-order execution using incremental flushing. In *(CAV '98)*, LNCS 1427, June 1998.
15. M. N. Velev. Using rewriting rules and positive equality to formally verify wide-issue out-of-order microprocessors with a reorder buffer. In *Design, Automation and Test in Europe (DATE '02)*, pages 28–35, March 2002.