Modeling and Verification of Out-of-Order Microprocessors in UCLID

Shuvendu K. Lahiri[2], Sanjit A. Seshia[1], and Randal E. Bryant[1,2]

[1] School of Computer Science, Carnegie Mellon University, Pittsburgh, PA
{Randy.Bryant, Sanjit.Seshia}@cs.cmu.edu
[2] Electrical and Computer Engineering Department, Carnegie Mellon University, Pittsburgh, PA
shuvendu@ece.cmu.edu

Abstract. In this paper, we describe the modeling and verification of out-of-order microprocessors with unbounded resources using an expressive, yet efficiently decidable, quantifier-free fragment of first order logic. This logic includes uninterpreted functions, equality, ordering, constrained lambda expressions, and counter arithmetic. UCLID is a tool for specifying and verifying systems expressed in this logic. The paper makes two main contributions. First, we show that the logic is expressive enough to model components found in most modern microprocessors, independent of their actual sizes. Second, we demonstrate UCLID's verification capabilities, ranging from full automation for bounded property checking to a high degree of automation in proving restricted classes of invariants. These techniques, coupled with a counterexample generation facility, are useful in establishing correctness of processor designs. We demonstrate UCLID's methods using a case study of a synthetic model of an out-of-order processor where all the invariants were proved automatically.

1 Introduction

Present-day microprocessors are complex systems, incorporating features such as pipelining, speculative, out-of-order execution, register-renaming, exceptions, and multi-level caching. Several formal verification techniques, including symbolic model checking [4, 12], theorem proving [17, 2, 11], and approaches based on decision procedures for the logic of equality with uninterpreted functions [8, 6, 20] have been used to verify such microarchitectures.

In previous work, Bryant et al.[5, 6] presented PEUF, a logic of positive equality with uninterpreted functions. PEUF has been shown to be expressive enough to model pipelined processors and also has a very efficient decision procedure based on Boolean techniques. Lahiri et al. [13] demonstrate the use of this technique for the verification of the superscalar, deeply pipelined MCORE[1] processor, by

[1] MCORE is a registered trademark of Motorola Inc.

finding bugs in the real design. However, this approach cannot handle models with unbounded queues and reorder buffers, which limits its applicability to processors with bounded resources. To overcome this problem, we have generalized PEUF to yield a more expressive logic called CLU [7], which is a logic of Counter Arithmetic with Lambda Expressions and Uninterpreted Functions. UCLID is a system for modeling and verifying systems modeled in CLU. It can be used to model a large class of infinite-state systems, including those with unbounded resources, while retaining the advantage of having an efficient decision procedure.

In this paper, we explore the application of UCLID to out-of-order processor designs. First, we illustrate the fact that CLU is expressive enough to model different processor components with unbounded resources. This includes components with infinite resources (e.g. infinite memory) or resources with finite but arbitrary size (e.g. a circular queue of arbitrary length). Next, we show that UCLID has useful verification capabilities that build upon the efficient decision procedure and a counterexample generator. We demonstrate the successful use of *bounded property checking*, i.e., checking an invariant on all the states of the system which are reachable within a fixed (bounded) number of steps from the reset state. The efficiency of UCLID's decision procedure enables a completely automatic exploration of a much larger state space than is possible with other techniques which can model infinite state systems. UCLID can also be used for *inductive invariant checking*, for a restricted class of invariants of the form $\forall x_1 \ldots \forall x_k . \Psi(x_1, \ldots, x_k)$, where $\Psi(x_1, \ldots, x_k)$ is a CLU formula. In our experience, this class of invariant is expressive enough to specify most invariants about out-of-order processors with unbounded size. These are also the most frequently occurring invariants that we have encountered in our experience with UCLID.

As a case study, we present the modeling and verification of a synthetic out-of-order processor, OOO, with ALU instructions, infinite memory, arbitrary large data words and an unbounded-size reorder buffer (first with an infinite size queue, and then with a finite but arbitrary size circular buffer). Bounded property checking was used initially to debug the design. The processor model was then formally verified by inductive invariant checking, by showing that it *refines* an instruction set architecture (ISA) model. The highlight of the verification was that all the invariants were *proved* fully automatically. Moreover, very little manual effort was needed in coming up with auxiliary invariants, which were inferred fairly easily from counterexample traces.

Related Work. Jhala and McMillan [12] use compositional model checking to verify a microarchitecture with speculative, out-of-order execution, load-store buffers and branch prediction. Apart from requiring the user to write down the refinement maps and case-splits to prove lemmas, the rest of the verification is automatically performed using Cadence SMV. The out-of-order processor we verify is similar in complexity to the model of Tomasulo algorithm McMillan verified using compositional reasoning [14]. The author acknowledges that the proof is not automatic and substantial human effort is required to decompose the proof into lemmas about small components of states. The main advantage

of using model checking is in automatically computing the strongest invariants for the most general state of the system; in our case, once the invariants have been figured out by the user, the rest of the proof is fully automatic and no manual decomposition is required. Berezin et al. [4] use special data-structures called *reference files*, along with other symmetry reduction techniques, to manually decompose a generic out-of-order execution model to a finite model, which is verified using a model checker. The manual guidance involved in decomposing the model limits the applicability of this approach to small, simple designs. Sawada and Hunt [17] use theorem proving methodology to verify the correctness of microarchitectures with out-of-order execution, load-store instruction and speculation. They use a trace-table based intermediate representation called MAETT to record both committed and in-flight instructions. This method requires extensive user guidance during the verification process, first in discovering invariants, and then in proving them using the ACL2 theorem prover. The authors claim that automating the proof of the lemmas would make the verification easier. Automating proof is central to our work and we illustrate it with the verification of an out-of-order unit. Hosabettu et al. [10, 11] use a *completion function* approach to verify advanced microarchitectures which includes reorder buffers, using the PVS [16] theorem prover. The method requires user ingenuity to construct a completion function for the different instruction types and then composing the different completion functions to obtain the abstraction function. The approach further requires extensive user guidance in discharging the proofs. Although the out-of-order unit we verify is of similar complexity as that in their original work [10], we shall show that the invariants required in our verification are few and simple, and they are discharged in a completely automatic manner. Arons et al. [1, 2] also verify out-of-order processors using refinement within PVS theorem prover. Our verification scheme is very similar to their approach as it also uses *prediction* to establish the correspondence with a sequential ISA. The model verified in [1] is similar in complexity to ours but once again substantial manual assistance is required to prove the invariants using PVS. Skakkebaek et al. [19] manually transform an out-of-order model of a processor to an intermediate in-order model, and use *incremental flushing* to show the correspondence of the intermediate model with the ISA model. The manual component in the entire process is significant in both constructing the intermediate model and proving correctness. Velev [20] has verified an out-of-order execution unit exploiting positive equality and rewrite rules. The model does not have register-renaming and still considers bounded (although very large) resources.

The rest of the paper is organized as follows. We begin by describing the UCLID system in Section 2. This section outlines the underlying logic CLU in Section 2.1 and the verification techniques supported in the UCLID framework in Section 2.2. Modeling primitives for various processor components are described in Section 3. Section 4 describes the case study of the verification of an out-of-order processor unit (OOO) in detail. The section contains a description of the processor, all the invariants required, and the use of bounded property check-

ing and inductive invariant checking for the verification of the OOO unit. We conclude in Section 5.

2 The UCLID system

2.1 The CLU Logic

The logic of Counter Arithmetic with Lambda Expressions and Uninterpreted Functions (CLU) is a generalization of Logic of Equality with Uninterpreted Functions (EUF) [8] with constrained lambda expressions, ordering, interpreted functions for successor (**succ**) and predecessor (**pred**) operations, that we will refer to as *counter arithmetic*.

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

Fig. 1. CLU Syntax.

Expressions in CLU describe means of computing four different types of values. *Boolean* expressions, also termed *formulas*, yield **true** or **false**. *Integer* expressions, also referred to as *terms*, yield integer values. *Predicate* expressions denote functions from integers to Boolean values. *Function* expressions, denote functions from integers to integers. Figure 1 summarizes the expression syntax. The simplest Boolean expressions are **true** and **false**. Boolean expressions can also be formed by comparing two integer expressions for equality or for ordering, or by applying a predicate expression to a list of integer expressions, or by combining Boolean expressions using Boolean connectives. Integer expressions can be integer variables[2], or can be formed by applying a function expression (including interpreted functions **succ** and **pred**) to a list of integer expressions, or by applying the *ITE* (for "if-then-else") operator. The *ITE* operator chooses between two values based on a Boolean control value, i.e., *ITE*(**true**, x_1, x_2) yields x_1 while *ITE*(**false**, x_1, x_2) yields x_2. Function (predicate) expressions can be either function (predicate) symbols, representing uninterpreted functions (predicates), or lambda expressions, defining the value of the function (predicate) as an integer

[2] Integer variables are used only as the formal arguments of lambda expressions

(Boolean) expression containing references to a set of argument variables. We will omit parentheses for function and predicate symbols with zero arguments, writing a instead of $a()$.

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. We say that an expression is *well-formed* when it contains no unbound variables. The value of a well-formed expression in CLU is defined relative to an interpretation I of the function and predicate symbols. Let $\mathcal{Z}$ denote the set of integers. Interpretation I assigns to each function symbol of arity k, a function from $\mathcal{Z}^k$ to $\mathcal{Z}$, and to each predicate symbol of arity k a function from $\mathcal{Z}^k$ to $\{\mathbf{true}, \mathbf{false}\}$. The value of a well-formed expression E in CLU relative to an interpretation I, $[E]_I$ is defined inductively over the expression structure. We shall omit the details in this paper. A well-formed formula F is *true under interpretation* I if $[F]_I$ is **true**. It is *valid* when it is true under all possible interpretations.

It can be easily shown that CLU has a *small-model property*, i.e. a CLU formula F_{clu} is valid iff F_{clu} is valid over all interpretations whose domain size equals the number of distinct terms in F_{clu}. The decision procedure for CLU checks the validity of a well-formed formula F by translating it to an equivalent propositional formula. The structure of the formula is exploited for *positive equality* [5] to dramatically reduce the number of interpretations to consider, yielding a very efficient decision procedure for CLU [7]. For brevity, we will not discuss the decision procedure in this paper.

2.2 Verification with UCLID

The UCLID specification language can be used to specify a state machine, where the state variables either have primitive types — Boolean, enumerated, or (unbounded) integer — or are functions of integer arguments that evaluate to these primitive types. The concept of using functions or predicates as state variables has previously been used in Cadence SMV, and in theorem provers as well. A system is specified in UCLID by describing initial-state and next-state expressions for each state variable.

The UCLID verification engine comprises of a symbolic simulator that can be "configured" for different kinds of verification tasks, and a decision procedure for CLU. We shall illustrate the use of two particular techniques for the verification of out-of-order processors. The reader is referred to [7] for more details.

1. *Bounded property checking*: The system is symbolically simulated for a fixed number of steps starting from the reset state. At each step, the decision procedure is invoked to check the validity of some safety property. If the property fails, then we can generate a counterexample trace from the reset state.
2. *Inductive invariant checking*: The system is started from the most general state which satisfies the invariants and then simulated for one step. The invariants are checked at the next step to ensure that the state transition

preserves the invariant. If the invariants hold for the reset state, and the invariants are preserved by the transition function, then the invariants hold for any reachable state of the model. As we shall see in the next section, we can express an interesting class of invariants with universal quantifiers and can automatically decide that the transition function preserves the invariants.

Counterexample Generation. One of the useful features of UCLID is its ability to generate counterexample traces, much like a model checker. A counterexample to a CLU formula F_{clu} is a partial interpretation I to the various function and predicate symbols in the formula. If the system has been symbolically simulated for k steps, then the interpretation I generated above can be applied to the expressions at each step, thereby resulting in a complete counterexample trace for k steps. The counterexample generation is useful in both bounded property checking to discover bugs in the design and in inductive invariant checking for adding more auxiliary invariants.

Invariant Checking and Quantifiers. The logic of CLU has been restricted to be quantifier-free. Hence a *well-formed* formula in this logic can be decided for validity using the small-model property of CLU. Although this restriction is not severe in the modeling of the out-of-order processors we consider, the need for quantifiers become apparent when UCLID is used for invariant checking. The invariants we encounter are frequently of the form $\forall x_1 \forall x_2 \ldots \forall x_k \Phi(x_1, \ldots, x_k)$, 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 actually preserved by the state transition function, we need to decide the validity of formulas of the form

$$\forall x_1 \ldots \forall x_m \Psi(x_1, \ldots, x_m) \implies \forall y_1 \ldots \forall y_k \Phi(y_1, \ldots, y_k) \tag{1}$$

where $\Psi(x_1, \ldots, x_m), \Phi(y_1, \ldots, y_k)$ are CLU formulas, $x_1 \ldots x_m$ and $y_1 \ldots y_k$ are *free* in $\Psi(x_1, \ldots, x_m)$ and $\Phi(y_1, \ldots, y_k)$ respectively. In general, the problem of checking validity of first-order formulas of the form (1), with uninterpreted functions is undecidable [9]. Note that this class of formulas cannot be expressed in CLU, since CLU is a quantifier-free logic. However, UCLID has a preprocessor for formulas of the form (1), which are translated to a CLU formula, which is a more conservative form of the original formula, i.e. if the CLU formula is valid then the original formula is valid. As we shall demonstrate, this has proved very effective for automatically checking the class of invariants encountered in our verification with out-of-order processors.

We employ a very simple heuristic to convert formulas of the form (1) to a CLU formula. First, the universal quantifiers to the right of the implication in (1) are removed by *skolemization* to yield the following formula, which is equivalent to the formula in (1)

$$\forall x_1 \ldots \forall x_m \Psi(x_1, \ldots, x_m) \implies \Phi(\widehat{y}_1, \ldots, \widehat{y}_k) \tag{2}$$

where $\widehat{y}_1, \ldots, \widehat{y}_k$ are fresh function symbols of arity 0. Second, as in deductive verification, we *instantiate* $x_1 \ldots x_m$ with concrete terms and the universal quantifiers to the left of the implication are replaced by a finite conjunction over these

concrete terms. The resulting formula is a CLU formula, whose validity implies the validity of (1). The set of terms over which to instantiate the antecedent is chosen as follows.

Let $\mathcal{T}(F_{clu})$ be the set of all terms (integer expressions) which occur in a CLU expression F_{clu}. For each bound variable x_i in $\forall x_1 \ldots \forall x_m \Psi(x_1, \ldots, x_m)$, we denote $\mathcal{F}^j_{x_i} = \{ \mathbf{f} \mid \mathbf{f}$ is an function or predicate symbol and x_i occurs as j^{th} argument to $\mathbf{f}$ in $\Psi(x_1, \ldots, x_m)\}$. Further, for each function or predicate symbol $\mathbf{f}$ which occurs in $\Psi(x_1, \ldots, x_m)$, denote $\mathcal{G}^k_f = \{ \mathbf{T} \mid \mathbf{T} \in \mathcal{T}(\Phi)$, and appears as the k^{th} argument to $\mathbf{f}$ in $\Phi(\widehat{y_1}, \ldots, \widehat{y_k}) \}$. The set of arguments that each bound variable x_i takes is given by $\mathcal{A}_{x_i} = \bigcup_j \{ \mathbf{T} \mid \mathbf{T} \in \mathcal{G}^j_f$ for some $\mathbf{f} \in \mathcal{F}^j_{x_i}\}$. Finally, $\Psi(x_1, \ldots, x_m)$ is instantiated over all the terms in Cartesian product, $\mathcal{A}_{x_1} \times \mathcal{A}_{x_2} \ldots \times \mathcal{A}_{x_m}$.

For example, consider the following quantified formula

$$\forall x_1 \forall x_2 . f(x_1, x_2) = g(x_2, x_1) \implies \forall y . f(h_2(y), h_1(y)) = g(h_1(y), h_2(y))$$

where $\Psi \equiv f(x_1, x_2) = g(x_2, x_1)$ and $\Phi \equiv f(h_2(y), h_1(y)) = g(h_1(y), h_2(y))$. In this case,$\mathcal{F}^1_{x_1} = \{f\}$, $\mathcal{F}^2_{x_1} = \{g\}$ and $\mathcal{F}^1_{x_2} = \{g\}$, $\mathcal{F}^2_{x_2} = \{f\}$. Similarly, $\mathcal{G}^1_f = \{h_2(\widehat{y})\}$, $\mathcal{G}^2_f = \{h_1(\widehat{y})\}$ and $\mathcal{G}^1_g = \{h_1(\widehat{y})\}$, $\mathcal{G}^2_g = \{h_2(\widehat{y})\}$. Finally, $\mathcal{A}_{x_1} = \{h_2(\widehat{y})\}$ and $\mathcal{A}_{x_2} = \{h_1(\widehat{y})\}$. Hence the bound variables x_1, x_2 are instantiated over $\{h_2(\widehat{y})\}$ and $\{h_1(\widehat{y})\}$ respectively. Hence the CLU formula becomes :

$$f(h_2(\widehat{y}), h_1(\widehat{y})) = g(h_1(\widehat{y}), h_2(\widehat{y})) \implies f(h_2(\widehat{y}), h_1(\widehat{y})) = g(h_1(\widehat{y}), h_2(\widehat{y}))$$

which is valid.

It is easy to see that this method would cause a blowup which is exponential in the number of bound variables in $\forall x_1 \ldots \forall x_m \Psi(x_1, \ldots, x_m)$. However our experience shows that the form of invariants we normally consider have very few bound variables which the decision procedure for UCLID can handle. More importantly, we will demonstrate in Section 4.2 that this simple translation to CLU formula helps us decide many equations of the form (1).

3 Modeling Components of Microprocessors

This section presents techniques to model commonly found structures of modern superscalar processor designs. Primitive constructs have been drawn from a wide spectrum of industrial processor designs, including those of the MIPS R10000, PowerPC 620, and Pentium Pro [18].

3.1 Terms, Uninterpreted Functions and Data Abstraction

Microprocessors are described using the standard *term-level* modeling primitives [17, 12, 21], where data-words and bit-vectors are abstracted with *terms*, and functional units abstracted with *uninterpreted functions*.

3.2 Memories

In this section, we look at a few different formulations of memories found in processors and show how the lambda notations offer a very natural modeling capability for memories.

Indexed Memories. Data memory and register file are examples of *indexed* memories. The set of operations supported by this form of memory are **read, write**. At any point in system operation, an *indexed* memory is represented by a function expression M denoting a mapping from addresses to data values. The initial state of the memory is given by an uninterpreted function symbol m_0 which denotes an arbitrary memory state. The effect of a write operation with integer expressions A and D denoting the address and data values yields a function expression M':

$$M' = \lambda\, addr\, .\, ITE(addr = A,\, D,\, M(addr))$$

where $M(addr)$ denotes a read from the memory at an address $addr$.

Content Addressable Memories. Register Rename units and Translation Lookaside Buffers (TLBs) are examples of Content Addressable Memory (CAM), that store associations between key and data. We represent a CAM as a pair $C = \langle C.data, C.present \rangle$, where $C.present$ is a predicate expression such that $C.present(k)$ is true for any key k that is stored in the CAM, and $C.data$ is a function expression such that $C.data(k)$ yields the data associated with key k, assuming the key is present. The next state components of a CAM for different operations are shown in Figure 2.

Operation	$C'.present$	$C'.data$
$Insert(C, K, D)$	$\lambda\, key\, .\, (key = K) \vee C.present(key)$	$\lambda\, key\, .\, ITE(key = K,\, D,\, C.data(key))$
$Delete(C, K)$	$\lambda\, key\, .\, \neg(key = K) \wedge C.present(key)$	$C.data$

Fig. 2. CAM operations

Simultaneous-update arrays. Many structures such as reorder buffers, reservation stations in processors, snoop on the result bus to update an arbitrary number of entries in the array at a single instant. At any point in time, the entry at index i in M can be updated with a data $D(i)$ if the predicate $P(i)$ is satisfied. The next state of the array is denoted as:

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

Note that an arbitrary subset of entries in the array can get updated at any time.

3.3 Queues and FIFO Buffers

Processors which employ out-of-order execution mechanisms or prefetching use a variety of queues in the microarchitecture. Instruction buffers, reorder buffers,

queues for deferring store instructions to memory, load queues to hold the load instructions which suffer a cache miss are found in most modern processors.

Queues. A finite circular queue of arbitrary length can be modeled by augmenting a CAM with two pointers to point to the head and the tail of the queue. Insertion (*push*) of data takes place only at the tail of the queue, and deletion (*pop*) takes place only at the head. Thus a circular queue can be modeled as a record $Q = \langle Q.data,\ Q.present,\ Q.head,\ Q.tail \rangle$. $Q.data$ and $Q.present$ are defined exactly as in Section 3.2. $Q.head$ is the index of the head of the queue, $Q.tail$ is the index of the tail (next insertion point) of the queue. Let the symbolic constants s and e represent the start and end points of the array over which the circular queue is implemented. The queue is empty when $Q.head = Q.tail$ and $Q.present(Q.head) = \mathbf{false}$. The queue is full when $Q.head = Q.tail$ and $Q.present(Q.head) = \mathbf{true}$.

To model the effect of **succ** and **pred** *modulo* certain integer, we define the the *modulo* increment and decrement functions $\mathbf{succ}_{[s,e]}$ and $\mathbf{pred}_{[s,e]}$ as follows:

$$\mathbf{succ}_{[s,e]} := \lambda\, i\ .\ ITE(i = e,\ s,\ \mathbf{succ}(i))$$
$$\mathbf{pred}_{[s,e]} := \lambda\, i\ .\ ITE(i = s,\ e,\ \mathbf{pred}(i))$$

Popping data item from Q returns a new queue Q' whose components have the value:

$Q'.head = \mathbf{succ}_{[s,e]}(Q.head)$	$Q'.present = \lambda\, i\ .\ \neg(i = Q.head) \wedge Q.present(i)$
$Q'.tail = Q.tail$	$Q'.data = Q.data$

Pushing a data item X into Q returns a new queue Q' where

$Q'.head = Q.head$	$Q'.present = \lambda\, i\ .\ (i = Q.tail) \vee C.present(i)$
$Q'.tail = \mathbf{succ}_{[s,e]}(Q.tail)$	$Q'.data = \lambda\, i\ .\ ITE(i = Q.tail,\ X,\ Q.data(i))$

This formulation of queue is used when the the index to the queue is used as a key in the system. The reorder buffers in processors follow this formulation, because the index in the reorder buffer uniquely identifies the instruction at the index. It is easy to see that for the case when $\mathbf{succ}_{[s,e]} = \mathbf{succ}$ and $\mathbf{pred}_{[s,e]} = \mathbf{pred}$, we obtain an unbounded infinite queue. $Q.present$ would be redundant in that situation.

FIFO Buffers. Alternate formulation of queues where the index in the queue is not used as a key (normally referred as FIFO Buffers) are also found in processors. Instruction buffers and load buffers are some examples of this form of queue. Every time an entry is dequeued, the entire content of the queue is shifted by one place towards the head of the queue. If the symbolic constant *max* denotes the maximum length of the queue, then the queue is full when $(Q.tail = max)$ and is empty when $(Q.tail = Q.head)$. The other operations of the queue are given below.

Operation	$Q'.head$	$Q'.tail$	$Q'.data$
$Push(Q, X)$	$Q.head$	$\mathbf{succ}(Q.tail)$	$\lambda\, i\ .\ ITE(i = Q.tail,\ X,\ Q.data(i))$
$Pop(Q)$	$Q.head$	$\mathbf{pred}(Q.tail)$	$\lambda\, i\ .\ Q.data(\mathbf{succ}(i))$

4 OOO: A Synthetic Out-of-Order Processor

OOO is a simple, unspeculative, out-of-order execution unit with unbounded resources, depicted in Figure 3. The only instructions permitted are arithmetic and logical (ALU) instructions with two source operands and one destination operand. As shown in Figure 3, an instruction is read from program memory,

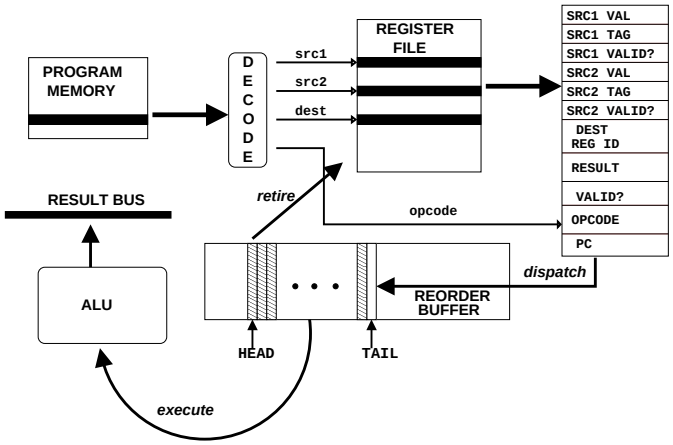

Fig. 3. OOO: An Out-of-order execution unit.

decoded, and dispatched to the end of the reorder buffer, which is modeled as an infinite queue. Instructions with ready operands can execute out-of-order. Finally, an instruction is retired (the program state updated), once it is at the head of the reorder buffer. On each step, the system nondeterministically chooses to either dispatch a new instruction, execute an instruction, or retire an instruction.

The register file is modeled as an infinite memory indexed by register ID. Each entry of the register file has a bit, `reg.valid`, a value `reg.val` and a tag `reg.tag`. If `reg.valid` bit is true, the `reg.val` contains a valid value, else, `reg.tag` would hold the tag of the most recent instruction that will write to this register.

The reorder buffer has two pointers, `rob.head`, which points to the oldest instruction in the reorder buffer, and `rob.tail`, where a newly dispatched instruction would be added. The index of an entry in the reorder buffer serves as its tag. Each entry in the reorder buffer has a valid bit `rob.valid` indicating if the instruction has finished execution. It has fields for the two operands `rob.src1val`, `rob.src2val`. The bit `rob.src1valid` indicates if the first operand is ready. If the first operand does not have valid data, `rob.src1tag` holds the tag for the instruction which would produce the operand data. There is a similar bit for the second operand. Each entry also contains the destination register identifier `rob.dest` and the result of the instruction `rob.value` to be written back. Further, each entry also stores the program counter (PC) for each entry in `rob.PC`.

When an instruction is dispatched, if a source register is marked valid in the register file, the contents of that register are filled into the corresponding operand

field for the instruction in the reorder buffer and it is marked valid. If the instruction which would write to the source register has finished execution, then the corresponding operand field copies the result of that instruction and the operand is marked valid. Otherwise, the operand copies the the tag present with the source register into its tag field and the operand is marked invalid. When an instruction executes, it updates its result, and broadcasts the result on the result bus so that all other instructions in the reorder buffer that are waiting on it can update their operand fields. Finally, when a completed instruction reaches the head of the reorder buffer, it is retired. If the tag of the retiring instruction matches the `reg.tag` for the destination register, the result of the instruction is written back into the destination register, and that register is marked valid. Otherwise, the register file remains unchanged.

4.1 Bounded Property Checking of OOO

The verification of the OOO model was carried out in two phases. In the first phase, we applied bounded property checking to eliminate most of the bugs present in the original model of OOO. For instance, in the original model, a dispatched instruction only looked at the register file for its source operands. If the source was invalid, it was enqueued into the reorder buffer with its operand invalid. The counterexample trace demonstrated that an instruction in the ROB can hold the tag of an already retired instruction.

The purpose of bounded property checking is not only to discover bugs, but can also serve as a very useful semi-formal verification tool. We can argue that for a model with a circular ROB of size k, all the states of the OOO where (i) the length of the ROB is anywhere between $0, \ldots, k$, (ii) the value of the control bits `rob.src1valid`, `rob.src2valid`, `rob.valid` are arbitrary for each entry in the ROB and (iii) the control bit of each register `reg.valid` is arbitrary, can be reached within $2k$ steps from the reset state. $2k$ steps are needed to reach the state when the ROB is full and all the instructions in the ROB have finished execution. Thus a property verified upto $2k$ steps gives a *reasonable guarantee* that it would always hold for a implementation of OOO where the number of ROB entries is bound by k. This also means that if there is a bug for a particular implementation of OOO where the size of the ROB is bound by k, then there is a high likelihood of the bug being detected within $2k$ steps of bounded-property checking. In Fig 4, we demonstrate that the efficiency of the decision procedure enables UCLID to perform bounded property checking for a reasonable number of steps (upto 20), thus providing guarantee for OOO models with upto 10 ROB entries. Figure 4 shows the result for checking the following two properties:

1. **tag-consistency:**
   $\forall r_1 \forall r_2 [((r_1 \neq r_2) \wedge \neg reg.valid(r_1) \wedge \neg reg.valid(r_2)) \implies (reg.tag(r_1) \neq reg.tag(r_2))]$
2. **rf-rob:**
   $\forall r [\neg reg.valid(r) \implies rob.dest(reg.tag(r)) = r]$

The experiments were performed on a 1400MHz Pentium with 256MB memory running Linux. zChaff [15] was used as the SAT solver within UCLID. To compare the performance of UCLID's decision procedure, we also used SVC [3] to

decide the CLU formulas. Although SVC's logic is more expressive than CLU (includes bit-vectors and linear arithmetic in addition to CLU constructs), the decision procedure for CLU outperforms SVC for checking the properties of interest in bounded property checking. The key point to note is that UCLID (coupled with powerful SAT solvers like zChaff) enables automatic exploration of much larger state spaces than was previously possible with other techniques.

Property	#steps	F_{clu} size	F_{bool} size	UCLID time	SVC time
tag-consistency	6	346	1203	0.87	0.22
	10	2566	15290	10.80	233.18
	14	7480	62504	76.55	> 5hrs
	18	15098	173612	542.30	> 1 day
	20	19921	263413	1679.12	> 1 day
rf-rob	10	2308	14666	10.31	160.84
	14	7392	61196	71.29	> 8hr
	18	14982	171364	485.09	> 1day
	20	19791	260599	777.12	> 1day

Fig. 4. Experimental results for Bounded Property Checking with OOO. Here "steps" indicates the number of steps of symbolic simulation, "F_{clu}" denotes the CLU formula obtained after the symbolic simulation, "F_{bool}" denotes boolean formula obtained after translating a CLU formula to a propositional formula by the decision procedure, the "size" of a formula denotes the number of distinct nodes in the Directed Acyclic Graph (DAG) representing the formula. "UCLID time" is the time taken by UCLID decision procedure and "SVC time" is the time taken by SVC 1.1 to decide the CLU formula. "tag-consistency" and "rf-rob" denote the properties to be verified.

4.2 Verification of the OOO Unit by Invariant Checking

We verify the OOO processor by proving a *refinement map* between OOO and a sequential Instruction Set Architecture (ISA) model. The ISA contains a program counter `Isa.PC`, and a register file `Isa.rf`. The program counter `Isa.PC` is synchronized with the program counter for OOO. `Isa.rf` maintains the state of the register file when all the instructions in the reorder buffer (ROB) have retired and the ROB is empty. Every time an instruction $\texttt{I} \doteq (\texttt{r1,r2,d,op})$ is decoded and put into the ROB, the result of the instruction is computed and written to the destination register `d` in the ISA register file as follows:

$$\texttt{Isa.rf[d]} \leftarrow \mathit{Alu}(\texttt{op}, \texttt{Isa.rf[r1]}, \texttt{Isa.rf[r2]})$$

where, *Alu* is an uninterpreted function to abstract the actual computation of the execution unit.

To state the invariants for the OOO processor, we maintain some auxiliary state elements in addition to the state variables of the OOO unit. These structures are very similar to the auxiliary structures used by McMillan [14] and Arons [1] for verifying the correctness of out-of-order processors. We maintain the following structures to reason about the correctness.

1. A *shadow* reorder buffer, `Shadow.rob`, where each entry contains the *correct* values of the operands and the result. This structure is used to reason about the correctness of values in the ROB entries. `Shadow.rob` is a triple (`Shadow.value, Shadow.src1val, Shadow.src2val`). `Shadow.value(t)` contains the correct value of `rob.value(t)` in the ROB. Similarly, the other fields in the `Shadow.rob` contain the correct values for the two data operands. When an instruction `I` $\doteq$ (`r1,r2,d,op`) is decoded, the `Shadow.rob` structure at `rob.tail` is updated as follows:

$$\begin{aligned} \texttt{Shadow.value(rob.tail)} &\leftarrow Alu(\texttt{op}, \texttt{Isa.rf(r1)}, \texttt{Isa.rf(r2)}) \\ \texttt{Shadow.src1val(rob.tail)} &\leftarrow \texttt{Isa.rf(r1)} \\ \texttt{Shadow.src2val(rob.tail)} &\leftarrow \texttt{Isa.rf(r2)} \end{aligned}$$

2. A shadow program counter `Shadow.PC`, which points to the next instruction to be retired. It is incremented every time an instruction retires in OOO. The `Shadow.PC` is used to prove that OOO retires instruction in a sequential order.

Correctness criteria. The correctness is established by proving the following *refinement map* between the register file of the OOO unit and the ISA register file.

$$\forall r.[reg.valid(r) \implies (Isa.rf(r) = reg.val(r))] \qquad (\Psi_{Ha})$$

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

Inorder Retirement. We also prove that the OOO retires instruction in sequential order with the following lemma.

$$Shadow.PC = ITE(rob.head \neq rob.tail, rob.PC(rob.head), PC) \qquad (\Psi_{PC})$$

Note that this lemma is not required for establishing the correctness of OOO.

4.3 Invariants for the OOO unit

We needed to come up with 12 additional invariants to establish the correctness of the OOO model, and we describe all of them in this section. The invariants broadly fall under three categories. The first four invariants, Ψ_A, Ψ_{B1}, Ψ_C, Ψ_D are concerned with maintaining a *consistent* state within the OOO model. These invariants are required mainly due to the redundancy present in the OOO model. The invariants Ψ_E, Ψ_{Ga} establish the *correctness* of data in the register file and ROB. Lastly, invariants Ψ_{Gb}, Ψ_{Hc}, Ψ_{K1} are the *auxiliary* invariants, which were required to prove some of the invariants above. The invariant names have no special bearing, except Ψ_{B1}, Ψ_{E1} and Ψ_{K1} denote that there are similar invariants for the second operand. For the sake of readability, we define $\widetilde{\forall} t.\Phi(t)$ to be an abbreviation for $\forall t.((rob.head \leq t < rob.tail)) \implies \Phi(t)$.

Consistency Invariants. Invariant Ψ_A asserts that an instruction in the ROB can execute only when both the operands are ready.

$$\widetilde{\forall} t.[rob.valid(t) \implies (rob.src1valid(t) \wedge rob.src2valid(t))] \qquad (\Psi_A)$$

For any ROB entry t, if any operand is not valid, then the operand should hold the tag of an older entry which produces the data but has not yet completed execution. There is a similar invariant for the second operand.

$$\widetilde{\forall} t.[\neg rob.src1valid(t) \implies (\neg rob.valid(rob.src1tag(t)) \wedge (rob.head \leq rob.src1tag(t) < t)] \qquad (\Psi_{B1})$$

Invariant Ψ_C claims that if the instruction at index t writes to a register r : `rob.dest(t)`, then r can't have valid data and the tag carried by r would be either t or a newer entry.

$$\widetilde{\forall} t.[(t \leq reg.tag(rob.dest(t)) < rob.tail)) \wedge (\neg reg.valid(rob.dest(t))] \qquad (\Psi_C)$$

Invariant Ψ_D asserts that a register r can only be modified by an active instruction in the ROB which has r as the destination register.

$$\forall r.[\neg reg.valid(r) \implies ((rob.dest(reg.tag(r)) = r) \wedge (rob.head \leq reg.tag(r) < rob.tail))] \qquad (\Psi_D)$$

All the above invariants restrict the state of the OOO model to be a reachable state. Note that there is no reference to any *shadow* structure, because the *shadow* structures only provide correctness of values in the OOO model.

Correctness Invariants. Invariant Ψ_{E1} establishes the constraint between the `Shadow.src1val` and `rob.src1val`. It states that if any ROB entry has a valid operand, then it should be correct (equals the value in the Shadow structure for that entry). There is a similar invariant for the second operand.

$$\widetilde{\forall} t.[rob.src1valid(t) \implies (Shadow.src1val(t) = rob.src1val(t))] \qquad (\Psi_{E1})$$

The following invariant asserts that if an ROB entry has completed execution, then the result matches with the value in the shadow ROB.

$$\widetilde{\forall} t.[rob.valid(t) \implies (Shadow.value(t) = rob.value(t))] \qquad (\Psi_{Ga})$$

Auxiliary Invariants. We needed the following auxiliary invariants for the `Shadow.src1val`, `Shadow.value` and `Isa.rf` respectively to prove the previous invariants inductive.

$$\widetilde{\forall} t.[\neg rob.src1valid(t) \implies Shadow.src1val(t) = Shadow.value(rob.src1tag(t))] \qquad (\Psi_{K1})$$

The above invariant asserts that the correct value of a data operand which is not ready is the result of the instruction which would produce the data.

$$\widetilde{\forall} t.[(Shadow.value(t) = Alu(rob.opcode(t), Shadow.src1val(t), Shadow.src2val(t)))] \qquad (\Psi_{Gb})$$

The above invariant relates the result of execution to the correct value for any entry.

$$\forall r.[\neg reg.valid(r) \implies Isa.rf(r) = Shadow.value(reg.tag(r))] \qquad (\Psi_{Hc})$$

The invariant Ψ_{Hc} relates the value of a register r in the shadow register file with the result of the instruction which would write back to the register.

Finally, we conjoin all the invariants to make the monolithic invariant Ψ_{all}. Since $\forall$ distributes over $\wedge$, we pull the quantifiers out in the formula given here:

$$\Psi_{all} \doteq \forall r.\forall t.[\Psi_A(t) \wedge \Psi_{B1}(t) \wedge \Psi_{B2}(t) \wedge \Psi_C(t) \wedge \Psi_D(r) \wedge \Psi_{E1}(t) \wedge \Psi_{E2}(t) \wedge \Psi_{K1}(t) \wedge \Psi_{K2}(t) \wedge \Psi_{Ga}(t) \wedge \Psi_{Gb}(t) \wedge \Psi_{Ha}(r) \wedge \Psi_{Hc}(r)]$$

Proof of the invariants. Some of the invariants were manually deduced from a failure trace from the counterexample generator. The most complicated among them were the invariants for the shadow register file and shadow ROB entries. We spent two man-days to come up with all the invariants. The invariants were proved in a completely automatic way by automatically translating the invariants to a formula in CLU by the method described in Section 2.2, and using the decision procedure for CLU to decide the formula. As we claimed earlier, the translation of quantified formulas to a CLU formula does not blow up the formula in a huge way, since most of the formulas have at most two bound variables. For instance, consider the proof for the invariant Ψ_{Ha} as given in the UCLID framework:

```
decide(Inv_all => Inv_Ha_next(r1));
```

Here the invariant Ψ_{Ha} (written above as `Inv_Ha`) is checked in the next state if Ψ_{all} (written as `Inv_all`) holds in the current state for all registers `r` and all tags `t`. There are only two bound variables `r,t` in the antecedent. Since all our invariants are of the form $\forall r.\Phi(r)$ or $\forall t.\Psi(t)$, we had to consider at most two bound variables in the antecedent.

The final proof script had 13 such formulas (one for each invariant) to be decided, and they were discharged automatically by UCLID in 76.44 sec. on a 1400 MHz Pentium IV Linux machine with 256 MB of memory. The memory requirement was less than 20 MB for the entire run. There is still a lot of scope of improvement in the decision procedure. The proof script consisted of the shadow structures, definition of the invariants mentioned in the Section 4.3, and 13 lines of proof to prove all the invariants in the next state.

To prove the lemma Ψ_{PC} for the in-order retirement, we required two more auxiliary lemmas. nPC is an uninterpreted function to obtain the next sequential value of a program counter.

$$\widetilde{\forall} t.[(t > rob.head) \implies rob.PC(t) = nPC(rob.PC(t-1))] \qquad (\Psi_{PC1})$$

$$[(rob.head \neq rob.tail) \implies PC = nPC(rob.PC(rob.tail - 1))] \qquad (\Psi_{PC2})$$

4.4 Using a circular reorder buffer

The model verified in this section is somewhat unrealistic because of the infinite reorder buffer, since it never wraps around. Most reorder buffer implementations use a finite circular queue to model the reorder buffer. Thus tags are *reused* unlike the above model. Hence we re-did the verification using a model with a circular buffer of arbitrary size. We needed very little change to our original proof. First, the reorder buffer was modeled as a circular buffer with modulo successor and predecessor functions as defined in Section 3.3. Second, each ROB entry had an additional entry `rob.present` to indicate if the entry has a valid instruction, and to disambiguate between checking the ROB for full or empty. Third, the "$<$" operation was modified to take into account the wrap around for circular queues. Finally, we had to establish an invariant between `rob.present`, `rob.head` and `rob.tail` to ensure that an entry is present if and only if it lies between the `rob.head` and `rob.tail`. None of the invariants had to be modified except as mentioned above. Hence the proof of the processor with circular reorder buffer went through without any major changes to the model or invariants.

4.5 Liveness Proof

We give a high level proof sketch of liveness for the OOO processor similar to Hosabettu's proof [10] in PVS. Although this proof is not mechanical, it uses a high level induction which utilizes various invariants that have been proved in the previous section using UCLID.

Proposition 1. *Every dispatched instruction eventually gets executed and retired, assuming fair scheduling of instructions.*

- We show that every instruction eventually reaches the head of the ROB, and that the instruction at the head is eventually retired. Our proof proceeds by induction on the size of the ROB.
- The base case, when the ROB is empty is trivial.
- Let us assume that the proposition holds when the ROB has less than k entries in it. Now, consider the case when there are k entries in the reorder buffer. Observe that `rob.head` (`rob.tail`) is incremented only when an instruction is retired (dispatched).
  - A *fair* scheduler will attempt to retire the instruction at the head of ROB infinitely often, therefore the instruction at the `rob.head` is eventually retired if it gets executed. This is because the instruction at the head is retired if the `rob.valid` bit is set for the entry at the head.
  - Observe that a *fair* scheduler attempts to execute any instruction in the ROB infinitely often, thus an instruction at index t, executes if it has both the operands ready (i.e. `rob.src1valid(t)` and `rob.src2valid(t)` are **true**). But, invariant Ψ_{B1} and Ψ_{B2} assert that both `rob.src1valid` and `rob.src2valid` for `rob.head` are **true**. Thus the instruction at the head of the ROB eventually gets executed and thus, retired.

Thus for any finite sequence of instructions, the ROB eventually becomes empty. With an empty ROB, invariant Ψ_D ensures that all the registers have `reg.valid` bit as **true**. Hence, by invariant Ψ_{Ha}, we know that the state of the register files in both OOO and the ISA model would eventually match.

5 Conclusions and Future Work

We have demonstrated the use of UCLID in modeling and verifying out-of-order processor designs. We showed the use of two different verification techniques that provide varying correctness guarantees and degrees of automation, ranging from bounded property checking, which provides full automation and debugging facilities, to invariant checking, which allows for full correctness checking at the cost of a manual assistance in deriving the invariants. Our hope is that the automation provided by bounded property checking and the proof of invariants would be of great help in analyzing large designs. We are currently trying to extend the verification to an out-of-order unit with exceptions, load-store instructions and branch-prediction. We have also started the verification of the MIPS R10000 processor as an industrial case study.

Acknowledgments

This research was supported in part by the Semiconductor Research Corporation, Contract RID 684, and by the Gigascale Research Center, Contract 98DT-660. The second author was supported in part by a National Defense Science and Engineering Graduate Fellowship.

References

1. T. Arons and A. Pnueli. Verifying Tomasulo's algorithm by Refinement. In *Proc. VLSI Design Conference (VLSI '99)*, 1999.
2. 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 2000)*, March 2000.
3. C. Barrett, D. Dill, and J. Levitt. Validity checking for combinations of theories with equality. In M. Srivas and A. Camilleri, editors, *Formal Methods in Computer-Aided Design (FMCAD '96)*, LNCS 1166, pages 187–201. Springer-Verlag, November 1996.
4. 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.
5. R. E. Bryant, S. German, and M. N. Velev. Exploiting positive equality in a logic of equality with uninterpreted functions. In N. Halbwachs and D. Peled, editors, *Computer-Aided Verification (CAV '99)*, LNCS 1633, pages 470–482. Springer-Verlag, July 1999.

6. 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.
7. 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) (to appear)*, July 2002.
8. J. R. Burch and D. L. Dill. Automated verification of pipelined microprocessor control. In D. L. Dill, editor, *Computer-Aided Verification (CAV '94)*, LNCS 818, pages 68–80. Springer-Verlag, June 1994.
9. Y. Gurevich. The decision problem for standard classes. *The Journal of Symbolic Logic*, 41(2):460–464, June 1976.
10. R. Hosabettu, G. Gopalakrishnan, and M. Srivas. Proof of correctness of a processor with reorder buffer using the completion function approach. In N. Halbwachs and D. Peled, editors, *Computer-Aided Verification (CAV 1999)*, volume 1633 of *Lecture Notes in Computer Science*. Springer-Verlag, July 1999.
11. R. Hosabettu, G. Gopalakrishnan, and M. Srivas. Verifying advanced microarchitectures that support speculation and exceptions. In A. Emerson and P. Sistla, editors, *Computer-Aided Verification (CAV 2000)*, LNCS 1855. Springer-Verlag, July 2000.
12. R. Jhala and K. McMillan. Microarchitecture verification by compositional model checking. In G. Berry, H. Comon, and A. Finkel, editors, *Computer-Aided Verification*, volume 2102 of *Lecture Notes in Computer Science*, pages 396–410. Springer-Verlag, July 2001.
13. S. Lahiri, C. Pixley, and K. Albin. Experience with term level modeling and verification of the MCORE microprocessor core. In *Proc. IEEE High Level Design Validation and Test (HLDVT 2001)*, November 2001.
14. K. McMillan. Verification of an implementation of Tomasulo's algorithm by compositional model checking. In A. J. Hu and M. Y. Vardi, editors, *Computer-Aided Verification (CAV 1998)*, volume 1427 of *Lecture Notes in Computer Science*. Springer-Verlag, June 1998.
15. M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *38th Design Automation Conference (DAC '01)*, June 2001.
16. S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752. Springer-Verlag, June 1992.
17. J. Sawada and W. Hunt. Processor verification with precise exceptions and speculative execution. In A. J. Hu and M. Y. Vardi, editors, *Computer-Aided Verification (CAV '98)*, LNCS 1427. Springer-Verlag, June 1998.
18. J. P. Shen and M. Lipasti. *Fundamentals of Superscalar Processor Design.* In Press, 2001.
19. J. U. Skakkaebaek, R. B. Jones, and D. L. Dill. Formal verification of out-of-order execution using incremental flushing. In A. J. Hu and M. Y. Vardi, editors, *Computer-Aided Verification (CAV '98)*, LNCS 1427. Springer-Verlag, June 1998.
20. 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.
21. M. N. Velev and R. E. Bryant. Formal Verification of Superscalar Microprocessors with Multicycle Functional Units, Exceptions and Branch Predication. In *37th Design Automation Conference (DAC '00)*, June 2000.