Appears in the proceedings of *Automated Reasoning with Analytic Tableaux and Related Methods (TABLEAUX '03)*, M.C. Mayer, and F. Pirri, *eds.*, LNAI 2796, Springer-Verlag, September 2003, pp. 196–213.

Automatic Abstraction of Equations in a Logic of Equality

Miroslav N. Velev
mvelev@ece.cmu.edu
Department of Electrical and Computer Engineering
Carnegie Mellon University, Pittsburgh, PA 15213, U.S.A.

Abstract. The paper presents a method to automatically abstract equations when translating formulas with equality to equivalent Boolean formulas, allowing the use of a SAT-checker to determine the validity of the original formula. The equations are abstracted with a special interpreted predicate that satisfies the properties of symmetry, reflexivity, transitivity, and functional consistency. This abstraction is both sound and complete. In contrast to previous methods that encode only low-level equations between term variables, the presented abstraction directly encodes top-level equations where the arguments can be nested-*ITE* expressions that select term variables. The automatic abstraction was used to formally verifying the safety of pipelined, superscalar, and VLIW processors, and reduced the CNF clauses by up to 50%, while speeding up the formal verification by up to an order of magnitude relative to the e_{ij} method where a new Boolean variable is used to encode each unique low-level equation between term variables. A heuristic for partial transitivity resulted in additional speedup for correct benchmarks that require transitivity.

1 Introduction

In formal verification of microprocessors, equations (equality comparisons) are used: 1) in the control logic, to express forwarding and stalling conditions, based on equality between a source and a destination register; 2) in mechanisms for correcting wrong speculations, when a predicted data value is not equal to the actual one; and 3) in the correctness formula, to compare the final architectural states of the implementation and the specification. The logic of Equality with Uninterpreted Functions and Memories (EUFM) [7] allows us to abstract functional units and memories, while completely modeling the control path of a processor. In EUFM, word-level values are abstracted with expressions called terms (see Sect. 2), whose only property is that of equality with other terms. In our previous work on using EUFM to formally verify pipelined, superscalar, and VLIW microprocessors [21][23], we imposed some simple restrictions on the style for defining high-level processors. The result was a significant reduction in the number of terms that appear in both positive and negated equations—and are so called *g-terms* (for general terms)—while increasing the number of terms that appear only in positive (not negated) equations—and are so called *p-terms* (for positive terms). We will refer to equations that appear in both positive and negated polarity as *g-equations*, and to those that appear only in positive polarity as *p-equations*. The property of Positive Equality [21] allowed us to treat syntactically different p-terms as not equal when checking the validity of an EUFM formula, thus achieving significant simplifications, and orders of magnitude speedup—see [5] for a correctness proof.

In the current paper, the implementation and specification are described in the high-

level hardware description language AbsHDL [25][27], based on the logic of EUFM. The formal verification is done with an automatic tool flow, consisting of: 1) the term-level symbolic simulator TLSim [25], used to symbolically simulate the implementation and specification, and to produce an EUFM correctness formula; 2) the decision procedure EVC [25] that exploits Positive Equality and other optimizations to translate the EUFM correctness formula to an equivalent Boolean formula, which has to be a tautology for the implementation to be correct; and 3) an efficient SAT-checker. This tool flow was used at Motorola [13] to formally verify a model of the M•CORE processor, and detected bugs. The tool flow was also used in an advanced computer architecture course [27][28], where undergraduate and graduate students designed and formally verified pipelined DLX [10] processors, including variants with exceptions and branch prediction, as well as dual-issue superscalar implementations.

While SAT-checkers are very quick to find a counterexample for a bug [26], they can be orders of magnitude slower when proving unsatisfiability of CNF formulas from correct designs. This paper proposes an approach to speed up the formal verification of correct models by abstracting the g-equations in a sound and complete way that results in a conceptually simpler solution space, fewer CNF clauses, and up to an order of magnitude reduction in the SAT-checking decisions and conflicts, relative to previous methods for encoding g-equations with Boolean variables [9][16].

2 Background

The formal verification is done by correspondence checking—comparison of a pipelined implementation against a non-pipelined specification, using flushing [7][8] to automatically compute an *abstraction function* that maps an implementation state to an equivalent specification state. The safety property (see Figure 1) is expressed as a formula in the logic of EUFM, and checks that one step of the implementation corresponds to between 0 and k steps of the specification, where k is the issue width of the implementation. F_{Impl} is the transition function of the implementation, and F_{Spec} is the transition function of the specification. We will refer to the sequence of first applying the abstraction function and then exercising the specification as the *specification side* of the commutative diagram in Figure 1, and to the sequence of first exercising the implementation for one step and then applying the abstraction function as the *implementation side* of the commutative diagram.

The safety property is a proof by induction, since the initial implementation state, Q_{Impl}, is completely arbitrary. If the implementation is correct for all transitions that can be made for one step from an arbitrary initial state, then the implementation will be correct for one step from the next implementation state, Q'_{Impl}, since that state will be a special case of an arbitrary state as used for the initial state, and so on for any number of steps. For some processors, e.g., where the control logic is optimized by using unreachable states as don't-care conditions, we may have to impose a set of *invariant constraints* for the initial implementation state in order to exclude unreachable states. Then, we need to prove that those constraints will be satisfied in the implementation state after one step, Q'_{Impl}, so that the correctness will hold by induction for that state, and so on for all subsequent states. See [1][2] for a discussion of correctness criteria.

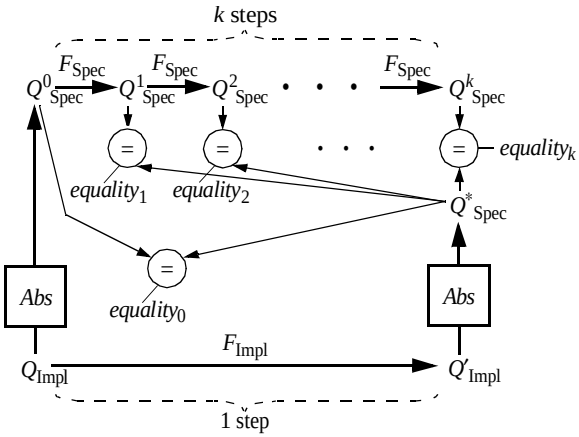

Fig. 1. The safety correctness property for an implementation with issue width k: one step of the implementation should correspond to between 0 and k steps of the specification, when the implementation starts from arbitrary initial state Q_{Impl} that may be restricted by invariant constraints

To illustrate the safety property in Figure 1, let the implementation and specification have three architectural state elements—program counter (PC), register file, and data memory. Let PC^i_{Spec}, $RegFile^i_{\text{Spec}}$, and $DMem^i_{\text{Spec}}$ be the state of the PC, register file, and data memory, respectively, in specification state Q^i_{Spec} (i = 0, ..., k) along the specification side. Let PC^*_{Spec}, $RegFile^*_{\text{Spec}}$, and $DMem^*_{\text{Spec}}$ be the state of the PC, register file, and data memory in specification state Q^*_{Spec}, reached after the implementation side of the diagram. Then, each disjunct $equality_i$ (i = 0, ..., k) is defined as:

$$equality_i \leftarrow pc_i \wedge rf_i \wedge dm_i,$$

where

$$pc_i \leftarrow (PC^i_{\text{Spec}} = PC^*_{\text{Spec}}),$$
$$rf_i \leftarrow (RegFile^i_{\text{Spec}} = RegFile^*_{\text{Spec}}),$$
$$dm_i \leftarrow (DMem^i_{\text{Spec}} = DMem^*_{\text{Spec}}).$$

That is, $equality_i$ is the conjunction of the pair-wise equality comparisons for all architectural state elements, thus ensuring that the architectural state elements are updated in synchrony by the same number of instructions. In processors with more architectural state elements, an equality comparison is conjuncted similarly for each additional state element. Hence, for this implementation, the safety property is:

$$pc_0 \wedge rf_0 \wedge dm_0 \;\vee\; pc_1 \wedge rf_1 \wedge dm_1 \;\vee\; \ldots \;\vee\; pc_k \wedge rf_k \wedge dm_k \;=\; true. \tag{1}$$

The syntax of EUFM [7] includes *terms* and *formulas*. Terms are used to abstract word-level values of data, register identifiers, memory addresses, as well as the entire states of memory arrays. A term can be an Uninterpreted Function (UF) applied to a list of argument terms, a term variable, or an *ITE* operator selecting between two argument terms based on a controlling formula, such that $ITE(formula, term_1, term_2)$ will evaluate to $term_1$ if $formula = true$, and to $term_2$ if $formula = false$. The syntax for

terms can be extended to model memories by means of the interpreted functions *read* and *write* [7][24]. Formulas are used to model the control path of a processor, as well as to express the correctness condition. A formula can be an Uninterpreted Predicate (UP) applied to a list of argument terms, a propositional variable, an *ITE* operator selecting between two argument formulas based on a controlling formula, or an equation between two terms. Formulas can be negated and combined with Boolean connectives. We will refer to both terms and formulas as *expressions*.

UFs and UPs are used to abstract functional units by replacing them with "black boxes" that satisfy no particular properties other than that of *functional consistency*—that the same combinations of values to the inputs of the UF (or UP) produce the same output value. Then, it no longer matters whether the original functional unit is an adder, or a multiplier, etc., as long as the same UF (or UP) is used to replace it in both the implementation and the specification. Thus, we will prove a more general problem—that the processor is correct for any functionally consistent implementation of its functional units. However, this more general problem is easier to prove.

Two possible ways to impose the property of functional consistency of UFs and UPs are Ackermann constraints [3], and nested *ITE*s [21]. The Ackermann scheme replaces each UF (UP) application in the EUFM formula F with a new term variable (Boolean variable) and then adds external consistency constraints. For example, the UF application $f(a_1, b_1)$ will be replaced by a new term variable c_1, another application of the same UF, $f(a_2, b_2)$, will be replaced by a new term variable c_2. Then, the resulting EUFM formula F' will be extended as $[(a_1 = a_2) \wedge (b_1 = b_2) \Rightarrow (c_1 = c_2)] \Rightarrow F'$. In the nested-*ITE* scheme, the first application of the above UF is still replaced by a new term variable c_1. However, the second is replaced by $ITE((a_2 = a_1) \wedge (b_2 = b_1), c_1, c_2)$, where c_2 is a new term variable. A third one, $f(a_3, b_3)$, is replaced by $ITE((a_3 = a_1) \wedge (b_3 = b_1), c_1, ITE((a_3 = a_2) \wedge (b_3 = b_2), c_2, c_3))$, where c_3 is a new term variable, and so on. UPs are eliminated similarly, but with new Boolean variables.

To compare the sequence of *write* operations that form the final states of memories after the implementation and specification sides of the diagram, the decision procedure EVC [25] automatically introduces a new term variable to serve as *comparison address* for each memory. Let *cmp_addr* be the new term variable introduced for the register file. Then, each equation $(RegFile^{i}_{\mathrm{Spec}} = RegFile^{*}_{\mathrm{Spec}})$, is replaced with $(read(RegFile^{i}_{\mathrm{Spec}}, cmp\_addr) = read(RegFile^{*}_{\mathrm{Spec}}, cmp\_addr))$, thus checking whether an arbitrary address in the register file is modified in the same way by both sides of the diagram. EVC replaces a *read* from a sequence of *write*s with a sequence of nested *ITE*s, according to the forwarding property of the memory semantics, such that each *ITE* is controlled by an equation between the new term variable and the destination address of the eliminated *write*. These equations appear in dual polarity—positive when selecting the then-expression of the *ITE*, and negative when selecting the else-expression—i.e., are g-equations that need to be encoded with Boolean variables.

We will call *complete equality* the usual equality, where two (syntactically) different term variables a and b can be either equal or not equal to each other, and will use = to denote it. Reasoning about complete equality requires a case split, in order to account for both cases, and so the need to encode it with Boolean variables when translating an EUFM formula to an equivalent Boolean formula. We will call *syntactic equality* the

subset of complete equality where a term variable is equal only to itself, and will use $=_{SYN}$ to denote it. We will call *delta equality* the difference between complete equality and syntactic equality, and will use $=_{\Delta}$ to denote it. That is, if t_1 and t_2 are two terms consisting of *ITE* operators, term variables, and formulas controlling the *ITE* operators, then $(t_1 =_{\Delta} t_2)$ is defined as $(t_1 = t_2) \wedge \neg(t_1 =_{SYN} t_2)$, or equivalently, complete equality $(t_1 = t_2)$ is defined as $(t_1 =_{SYN} t_2) \vee (t_1 =_{\Delta} t_2)$. We will call *hybrid equality* the extension of syntactic equality with a proper subset of the delta equality between two terms, and will denote it with $=_{HYB}$.

The property of Positive Equality is due to the observation that the correctness formula (1) consist of top-level p-equations that are combined with the monotonically positive connectives of conjunction and disjunction, but are not negated. Then, if the formula is valid (true) when the complete equality in the top-level p-equations is replaced with syntactic equality, the formula will also be valid with the original complete equality in those equations, since then the formula can only get bigger due to the omitted delta equality that will be added through monotonically positive connectives. However, the benefit from using only syntactic equality for the top-level p-equations is the significant reduction of the solution space, resulting in orders of magnitude speedup. Similarly, we exploit syntactic functional consistency when eliminating UFs and UPs in that the property of functional consistency is enforced only for the cases of syntactic equality between corresponding arguments in applications of the same UF/UP, unless both arguments are g-terms. Syntactic functional consistency is a conservative approximation, since functional consistency is enforced only for a subset of the conditions for complete functional consistency (based on complete equality). If F is a formula obtained after eliminating all UFs/UPs by accounting for only syntactic functional consistency, and F is valid, then so will be the formula obtained from F by accounting for complete functional consistency, e.g., by extending F with Ackermann constraints for complete functional consistency. However, g-equations could be either *true* or *false*, and need to be encoded with Boolean variables [9][16].

A *low-level g-equation* is one where both arguments are term variables. A *top-level g-equation* is one where the arguments can be either term variables or nested-*ITE* expressions selecting term variables. Previous methods for encoding g-equations with Boolean variables [9][16] eliminate top-level g-equations by pushing them to the argument term variables, and then encode the resulting low-level g-equations. The e_{ij} encoding [9] replaces each unique low-level equation between different term variables v_i and v_j with a new Boolean variable, called e_{ij}. The property of symmetry of equality is accounted for by sorting v_i and v_j according to their indices, e.g., so that $i < j$, before introducing a Boolean variable; and the property of transitivity of equality—if $v_i = v_j$ and $v_j = v_k$ then $v_i = v_k$—is enforced with transitivity constraints of the form $e_{ij} \wedge e_{jk} \Rightarrow e_{ik}$. In the small-domain encoding [16], each g-term variable is assigned a set of constants that it can take on in a way that allows it to be either equal to or different from any other g-term variable that it can be transitively compared for equality with. If a g-term variable is assigned a set of N constants, then those can be indexed with $\lceil \log_2(N) \rceil$ Boolean variables. Two g-term variables are equal if their indexing Boolean variables select simultaneously a common constant. The property of transitivity is automatically enforced in this encoding. Depending on the structure of the g-term

equality-comparison graphs, the small-domain encoding may introduce fewer Boolean variables than the e_{ij} encoding. That could mean a smaller search space. However, now a low-level g-equation is replaced with a Boolean formula—enumerating all cases when the argument g-term variables evaluate to a common constant—instead of a single Boolean variable. In our previous work [26], we found the e_{ij} encoding to outperform the small-domain encoding when formally verifying microprocessors. For other benchmarks, Seshia et al. [18] proposed a hybrid encoding, such that the e_{ij} and small-domain encodings are each used on a different connected component of low-level g-equations in the same correctness formula. The decision procedure EVC adds all transitivity constraints for the e_{ij} variables to the CNF correctness formula, while the decision procedure CVC [4] iteratively analyzes counterexamples, and includes transitivity constraints incrementally—just as many as to prevent the recurrence of a counterexample. However, Seshia et al. [18] found the incremental approach to result in significant overhead when checking validity of complex formulas.

3 Automatic Abstraction of Equations

In this section, we will assume that the interpreted functions *read* and *write*, as well as all UFs and UPs, have been eliminated from the EUFM correctness formula. That is, each term in the formula is either a term variable, or a nested-*ITE* expression selecting term variables. In this formula, instead of encoding low-level g-equations, we can automatically abstract the top-level g-equations with the special interpreted predicate *abs_equality* that satisfies the properties of transitivity, syntactic functional consistency, syntactic symmetry, and (syntactic) reflexivity. Note that the abstracted complete equality is: 1) transitive—if $a = b$ and $b = c$ then $a = c$; 2) symmetric—if $a = b$ then $b = a$; 3) reflexive, i.e., $a = a$ is *true*; and 4) functionally consistent—given two equations $a = b$ and $c = d$, the equality between arguments in the same positions, i.e., $a = c$ and $b = d$, implies that the two equations have equal values, as follows from the property of transitivity, since the four equations form a cycle, and if three of them are *true* then the fourth should also be *true*, while if one is *false* and two are *true* then the fourth should be *false* or otherwise there will be a cycle of three equations that are *true*, implying that the first should be *true* and so contradicting its value. Also note that the property of reflexivity is equivalent to syntactic equality, since the property holds when exactly the same term variable appears on both sides of an equation.

We can extend either the nested-*ITE* or the Ackermann-constraint scheme for elimination of uninterpreted predicates in order to eliminate applications of *abs_equality* by accounting for its properties of syntactic functional consistency, syntactic symmetry, and reflexivity. Transitivity can be imposed as in the case of low-level g-equations [26]—by triangulating the equality-comparison graph (where each vertex is a term used in a top-level g-equation, and each edge corresponds to a g-equation between two terms) with extra edges, added in a greedy manner to turn every two edges with a common vertex into a triangle (cycle of length 3), and then imposing three transitivity constraints for the CNF variables representing the values of g-equations in a triangle.

In order to adopt the nested-*ITE* scheme, each *ITE*-controlling formula is extended to account for syntactic symmetry, while the top *ITE* expression is disjuncted with the

condition for syntactic equality between the two arguments, thus ensuring reflexivity. That is, the first application of *abs_equality*(t_1, t_2), where t_1 and t_2 are terms, is eliminated with $(t_1 =_{SYN} t_2) \vee E_1$, where E_1 is a new Boolean variable, and the disjunction of $(t_1 =_{SYN} t_2)$ ensures reflexivity. A second application *abs_equality*(t_3, t_4) is eliminated with $(t_3 =_{SYN} t_4) \vee ITE((t_3 =_{SYN} t_1) \wedge (t_4 =_{SYN} t_2) \vee (t_4 =_{SYN} t_1) \wedge (t_3 =_{SYN} t_2), E_1, E_2)$, where E_2 is a new Boolean variable, and the disjunction of $(t_3 =_{SYN} t_4)$ ensures reflexivity. In the controlling formula, the expression $(t_3 =_{SYN} t_1) \wedge (t_4 =_{SYN} t_2)$ ensures syntactic functional consistency, as in the original nested-*ITE* scheme for elimination of UFs and UPs, and the disjunction of $(t_4 =_{SYN} t_1) \wedge (t_3 =_{SYN} t_2)$ ensures syntactic symmetry. A third application *abs_equality*(t_5, t_6) is eliminated with $(t_5 =_{SYN} t_6) \vee ITE((t_5 =_{SYN} t_1) \wedge (t_6 =_{SYN} t_2) \vee (t_6 =_{SYN} t_1) \wedge (t_5 =_{SYN} t_2), E_1, ITE((t_5 =_{SYN} t_3) \wedge (t_6 =_{SYN} t_4) \vee (t_6 =_{SYN} t_3) \wedge (t_5 =_{SYN} t_4), E_2, E_3))$, where E_3 is a new Boolean variable.

The Ackermann-constraint scheme can be customized similarly. Each of the three applications, *abs_equality*(t_1, t_2), *abs_equality*(t_3, t_4), and *abs_equality*(t_5, t_6), will be eliminated with a new Boolean variable—E_1, E_2, and E_3, respectively. Let F' be the resulting EUFM formula. To account for the properties of reflexivity, syntactic symmetry, and syntactic functional consistency of *abs_equality*, we define separate constraints, conjunct them in formula *constraints*, and use it to restrict F', i.e., prove the validity of *constraints* $\Rightarrow F'$. In particular, to enforce reflexivity of the first, second, and third applications of *abs_equality*, we use the constraints $(t_1 =_{SYN} t_2) \Rightarrow E_1$, $(t_3 =_{SYN} t_4) \Rightarrow E_2$, and $(t_5 =_{SYN} t_6) \Rightarrow E_3$, respectively. To account for syntactic functional consistency and syntactic symmetry of the second application with respect to the first, we use $[(t_3 =_{SYN} t_1) \wedge (t_4 =_{SYN} t_2) \vee (t_4 =_{SYN} t_1) \wedge (t_3 =_{SYN} t_2)] \Rightarrow (E_1 \Leftrightarrow E_2)$. To account for syntactic functional consistency and syntactic symmetry of the third application with respect to the first, we use $[(t_5 =_{SYN} t_1) \wedge (t_6 =_{SYN} t_2) \vee (t_6 =_{SYN} t_1) \wedge (t_5 =_{SYN} t_2)] \Rightarrow (E_1 \Leftrightarrow E_3)$. Finally, to account for syntactic functional consistency and syntactic symmetry of the third application with respect to the second, we add the constraint $[(t_5 =_{SYN} t_3) \wedge (t_6 =_{SYN} t_4) \vee (t_6 =_{SYN} t_3) \wedge (t_5 =_{SYN} t_4)] \Rightarrow (E_2 \Leftrightarrow E_3)$.

Example: Let t_1, t_2, t_3, t_4, t_5, and t_6 be six terms defined as follows:

$$t_1 \leftarrow a \qquad t_2 \leftarrow b$$
$$t_3 \leftarrow ITE(f_1, c, a) \qquad t_4 \leftarrow c$$
$$t_5 \leftarrow ITE(f_2, d, a) \qquad t_6 \leftarrow ITE(f_3, a, b)$$

where a, b, c, and d are term variables, and f_1, f_2, and f_3 are formulas. Let $(t_1 = t_2)$, $(t_3 = t_4)$, and $(t_5 = t_6)$ be top-level g-equations in an EUFM formula.

To apply the e_{ij} encoding, the top-level g-equations will be pushed to their argument term variables: the first equation will remain unchanged, $(a = b)$, since both arguments are term variables, and will be replaced with the new Boolean variable e_{ab}; the second will become $ITE(f_1, c = c, a = c)$, i.e., $ITE(f_1, true, a = c)$, and will be replaced with $f_1 \vee e_{ac}$, after $a = c$ is encoded with the new Boolean variable e_{ac}; the third will become $ITE(f_2, ITE(f_3, a = d, b = d), ITE(f_3, a = a, a = b))$, and will be replaced with $f_2 \wedge f_3 \wedge e_{ad} \vee f_2 \wedge \neg f_3 \wedge e_{bd} \vee \neg f_2 \wedge f_3 \vee \neg f_2 \wedge \neg f_3 \wedge e_{ab}$, after $a = d$ is encoded with e_{ad} and $b = d$ is encoded with e_{bd}.

Using the special interpreted predicate *abs_equality*, the top-level g-equations will be abstracted as *abs_equality*(t_1, t_2), *abs_equality*(t_3, t_4), and *abs_equality*(t_5, t_6).

Then, using the nested-*ITE* scheme, the first application of *abs_equality* will be eliminated with $(a =_{SYN} b) \vee E_1$, which evaluates to E_1, since *a* and *b* are two (syntactically) different term variables, so that their syntactic equality evaluates to *false*. The second application will be eliminated with $(ITE(f_1, c, a) =_{SYN} c) \vee ITE((ITE(f_1, c, a) =_{SYN} a) \wedge (c =_{SYN} b) \vee (c =_{SYN} a) \wedge (ITE(f_1, c, a) =_{SYN} b), E_1, E_2)$, which evaluates to $f_1 \vee ITE(\neg f_1 \wedge false \vee false \wedge false, E_1, E_2)$, i.e., to $f_1 \vee E_2$, where f_1 expresses the condition for syntactic equality between the two arguments, while E_2 encodes the two possible values of the delta equality between the two argument terms. Eliminating the third application of *abs_equality*, and simplifying the resulting expression, we get $\neg f_2 \wedge f_3 \vee ITE(\neg f_2 \wedge \neg f_3, E_1, E_3)$, where formula $\neg f_2 \wedge f_3$ expresses the conditions for syntactic equality between the two arguments, while the *ITE* operator will select Boolean variable E_1 if formula $\neg f_2 \wedge \neg f_3$ is *true*, i.e., in the cases of syntactic functional consistency with the arguments of the first application of *abs_equality*, while the new Boolean variable E_3 encodes the delta equality between the two arguments in the cases when the arguments do not satisfy conditions for syntactic functional consistency or syntactic symmetry with respect to previous pairs of arguments.

Using Ackermann constraints to enforce reflexivity, syntactic functional consistency, and syntactic symmetry for *abs_equality*, the first, second, and third applications will be replaced with the new Boolean variables E_1, E_2, and E_3, respectively. Then, the resulting formula will be evaluated under the constraints: $f_1 \Rightarrow E_2$, enforcing reflexivity for *abs_equality*(t_3, t_4); $\neg f_2 \wedge f_3 \Rightarrow E_3$, enforcing reflexivity for *abs_equality*(t_5, t_6); and $\neg f_2 \wedge \neg f_3 \Rightarrow (E_1 \Leftrightarrow E_3)$, enforcing syntactic functional consistency between *abs_equality*(t_3, t_4) and *abs_equality*(t_5, t_6). □

THEOREM 1. Let *F* be an EUFM formula that contains term variables, Boolean variables, logic connectives, *ITE* operators, and equations. Then, abstracting the top-level g-equations in *F* with the interpreted predicate *abs_equality* is sound and complete.

Proof: Let formula F' be obtained from *F* after abstracting the top-level g-equations with the interpreted predicate *abs_equality.*

Soundness—the validity of F' implies the validity of *F.* If F' is valid, then so will be any formula obtained from F' after replacing *abs_equality* with any predicate that has two arguments, and satisfies the properties of transitivity, reflexivity, syntactic symmetry, and syntactic functional consistency, including the original complete equality. Note that by its definition, *abs_equality* satisfies the property of syntactic equality, i.e., reflexivity. What is missing from complete equality are two constraints: 1) if the delta equality between terms *a* and *b* is *true*, then *abs_equality*(*a*, *b*) should be *true*; and 2) if the complete equality between terms *a* and *b* is *false*, then *abs_equality*(*a*, *b*) should be *false*. However, if F' is valid without such constraints, it will be valid with them:

$$[((a =_{\Delta} b) \Rightarrow abs\_equality(a, b)) \wedge (\neg(a = b) \Rightarrow \neg abs\_equality(a, b))] \Rightarrow F',$$

where the resulting formula is trivially valid, since F' is already valid.

Completeness—a counterexample in F' can be mapped to a counterexample in *F.* A counterexample in F' consists of assignments to variables E_i, used when eliminating *abs_equality*, and assignments to the other Boolean variables that also appear in *F.* The arguments of each abstracted g-equation are either term variables or nested-*ITE*

expressions that select term variables, where the *ITE* operators are controlled by formulas that depend on applications of *abs_equality* and on the other Boolean variables. Hence, each counterexample results in an assignment to the *ITE*-controlling formulas, thus selecting some term variable *a* as the first argument of an application of *abs_equality*, and another term variable *b* as the second argument. Then, we can assign the value of that particular application of *abs_equality* to the low-level equation $a = b$ and can replace that application of *abs_equality* with the original complete equality. The correctness formula is a Directed Acyclic Graph (DAG), so that the value of the introduced top-level g-equation does not affect the arguments of that equation. Hence, the *ITE*-controlling formulas in the arguments will keep their values, and so *a* and *b* will still appear on the two sides of that g-equation, which will have the same value as the replaced application of *abs_equality*. We can similarly map the value of each remaining applications of *abs_equality* to a value of a low-level g-equation between the term variables selected by the nested-*ITE* arguments, and then replace that application of *abs_equality* with a top-level g-equation, which will get the same value as the one assigned to the low-level g-equation, i.e., as the one of the eliminated application of *abs_equality*. Thus, all abstractions of top-level g-equations will be undone, and we will get the original formula *F*. What remains to be proved is that these assignments to low-level g-equations will not violate the properties of equality. First, reflexivity is always preserved, since syntactic equality between the two arguments is always accounted for, and an application of *abs_equality* is forced to be *true* when exactly the same term variable is selected to appear on both sides of the abstracted g-equation, i.e., it is impossible for the same term variable to appear as both arguments of an application of *abs_equality* that evaluates to *false*. Second, constraints for syntactic functional consistency ensure that if the same pair of term variables is selected as arguments of different applications of *abs_equality*, then those applications will have the same value, i.e., it is impossible for the same low-level g-equation between term variables to get assigned contradicting values from different applications of *abs_equality*. Third, because of constraints for syntactic symmetry, it is similarly impossible for two symmetric low-level equations, $a = b$ and $b = a$, to get assigned different values. Fourth, transitivity will never be violated, since constraints for transitivity of equality ensure that applications of *abs_equality* do not violate transitivity, and, as described above, a counterexample determines a 1-to-1 mapping of every cycle of abstracted top-level g-equations to an isomorphic cycle of low-level g-equations, each having value identical to that of the corresponding abstracted top-level g-equation. □

Note that each counterexample maps the value of an abstracted top-level g-equation to exactly one low-level g-equation between term variables in the support of the top-level g-equation. The low-level g-equations that are left unassigned are don't-care conditions. They do not affect the counterexample, and can be left unassigned or given any value that does not violate transitivity, when interpreting the counterexample.

As an optimization, we can choose not to enforce transitivity, or reflexivity, or both; these properties are not needed for models with in-order execution, as shown in the experiments (see Sect. 5). Alternatively, we can enforce partial transitivity—a heuristic for that is presented in Sect. 4.3, and was found to speed up the formal verification of processors with out-of-order execution and completion.

4 Using the Automatic Abstraction

4.1 Identifying Connected Equality-Comparison Components

We will again assume that the formula contains term variables, Boolean variables, logic connectives, *ITE* operators, and equations. Each equation is classified as a p-equation or a g-equation, if it was reached under an even or odd number of negations, respectively, before the uninterpreted functions and uninterpreted predicates were eliminated. Syntactic equations introduced when eliminating uninterpreted functions and uninterpreted predicates are also classified as p-equations. The arguments of g-equations are either term variables or nested-*ITE* expressions selecting term variables. For each g-equation, all term variables that can be selected to appear as an argument are grouped into an *equivalence class*. Equivalence classes that have common term variables are merged and their g-equations are marked to belong to the same connected equality-comparison component. The properties of transitivity, functional consistency, and symmetry need to be enforced only within a connected component, since the values of equations from a connected component have no way of affecting equations from another connected component. That is, we can use a different version of interpreted predicate *abs_equality* for each connected component.

In processors with branch prediction, the equations for the PC states, pc_i, in correctness formula (1) will contain term variables that are arguments to g-equations introduced by the mechanism for correcting branch mispredictions—if the actual and predicted branch targets are equal, then the prediction is correct and any speculative instructions are allowed to complete; otherwise, the speculative instructions are squashed. To ensure that such p-equations have values that are consistent with those of abstracted g-equations that control the speculation and have common term variables as arguments, we need to promote p-equations to g-equations. That is, for each p-equation, determine the equivalence class of term variables that may appear as an argument; if this equivalence class has a common element with another equivalence class that identifies a connected component of g-equations, then merge the two equivalence classes, and promote the p-equation to a g-equation from that connected component.

4.2 Mapping Abstract Counterexamples to Concrete Ones

A counterexample for the abstract model, where top-level g-equations are abstracted with *abs_equality*, is expressed by an assignment to the E_i variables—used when eliminating applications of *abs_equality*—and an assignment to the other Boolean variables—representing initial state of control signals, or introduced when eliminating uninterpreted predicates. We can map an abstract counterexample to a concrete one for the original model by following:

Step 1. Compute the value of each application of *abs_equality*, based on the counterexample assignment to E_i and other Boolean variables in the abstract model.

Step 2. For each application of *abs_equality* (the arguments are either term variables, or nested-*ITE* expressions selecting term variables), compute the values of *ITE*-controlling formulas in the two arguments.

Step 3. For each application of *abs_equality*, find the two term variables that will be

selected for equality comparison in the abstracted g-equation, given the values of *ITE*-controlling formulas computed in Step 2. If this application of *abs_equality* evaluates to *true*, then the two term variables should be equal in order to trigger a corresponding counterexample in the concrete model; otherwise, they are not equal.

According to Theorem 1, the above steps will result in consistent assignments to low-level g-equations, without violating the properties of equality.

4.3 Heuristic for Partial Transitivity

In processors with out-of-order completion, the specification side of the commutative diagram (Figure 1) completes the instructions in program order—assuming the abstraction function completes the instructions in program order—while the implementation side may reorder them. In a correct implementation, out-of-order execution and completion occur only if that would not introduce write-after-read or write-after-write hazards [10]. That is, destination registers of younger instructions are compared for equality with both source and destination registers of older instructions (appearing earlier in program order). A younger instruction is issued/completed only if each older instruction is issued/completed, or if the younger instruction will not introduce a hazard for an older instruction. The absence of a write-after-write hazard, when the destination registers of two instructions are not equal, implies that term variable *cmp_addr*, used as comparison address for the final states of the register file (see Sect. 2), may equal only one of these destination registers, but not both, or that will violate a transitivity constraint. That is, if $dest_1$ and $dest_2$ are destination registers compared for equality by logic for preventing write-after-write hazards, then the comparison of the final register file states will introduce equations ($dest_1$ = *cmp_addr*) and ($dest_2$ = *cmp_addr*). However, at most one of them can be *true*, since transitivity of equality implies that $\neg(dest_1 = dest_2) \wedge (cmp\_addr = dest_1) \Rightarrow \neg(cmp\_addr = dest_2)$ and $\neg(dest_1 = dest_2) \wedge (cmp\_addr = dest_2) \Rightarrow \neg(cmp\_addr = dest_1)$.

Similar cycles of 3 equations, comparing two destination registers and a source register, may be introduced by the control logic in processors with out-of-order execution or completion—the equation between the two destination registers by logic checking for write-after-write hazards, and the two equations between a source register and each of the destination registers by logic checking for read-after-write or write-after-read hazards. Constraints for transitivity of equality are needed to prevent simultaneous forwarding of data from two older destination registers that are not equal, and whose instructions are reordered, to a younger source register. Hence, we can use a *heuristic for enforcing partial transitivity*. Let the names of all destination and source register identifiers contain the substring “Dest” and “Src”, respectively. Then, we can automatically detect pairs of destination registers compared for equality by the control logic. We can enforce partial transitivity only for such destination registers and any source registers occurring in equations with them, including term variable *cmp_addr*. As noted earlier, partial transitivity is a conservative approximation, since it results in discarding constraints. If the resulting formula F' is valid, it will also be valid when extended with the omitted transitivity constraints, *extra_transitivity*, to a formula $extra\_transitivity \Rightarrow F'$.

5 Results

The benchmarks used in the experiments are: 1dlx_c_mc_ex_bp, a single-issue pipelined DLX [10] with multicycle ALU, Instruction Memory, and Data Memory, as well as with exceptions and branch prediction, modeled and formally verified as described in [22]; 2dlx_cc_mc_ex_bp, a dual-issue superscalar DLX with in-order execution, and two identical execution pipelines with all of the above features [22]; 9vliw_mc_ex_bp, a 9-wide VLIW processor that also has all of the above features, as well as the same number and types of functional units as the Intel Itanium [11][19], and imitates it in predicated execution, register remapping, and advanced loads—modeled and formally verified as described in [23]; xscale, a model of the Intel XScale processor [12] with specialized execution pipelines, scoreboarding [10], out-of-order completion, and imprecise exceptions—modeled and formally verified as described in [20]; 12pipe, a superscalar processor that can issue up to 12 instructions in program order on every clock cycle, and is capable of executing only ALU instructions [26]; and 8pipe_ooo, a superscalar model that can issue up to 8 instructions out of program order on every cycle, and is also capable of executing only ALU instructions [26].

The experiments were performed on a Dell OptiPlex GX260 with a 3.06-GHz Intel Pentium 4 processor that had a 512-KB on-chip level-2 cache, 2 GB of physical memory, and was running Red Hat Linux 9. The SAT-checker Siege [17], a top-performer in the SAT'03 competition [14], was found to have best performance on these benchmarks and was used for the experiments. The reader is referred to [26] for the translation to CNF format. All constraints for transitivity of equality were added to the CNF formulas from buggy implementations and correct models that require transitivity (xscale and 8pipe_ooo), but were manually switched off for correct models that do not require transitivity. Transitivity was enforced by triangulating the equality comparison graphs [6], and adding transitivity constraints for each resulting cycle of length 3. All models were formally verified by computing the abstraction function with controlled flushing [8], where the user provides a flushing schedule that avoids the triggering of stalling conditions, thus simplifying the correctness formula.

Table 1 presents the results with the e_{ij} encoding. "Trans" CNF clauses represent constraints for transitivity of equality. The e_{ij} Boolean variables ranged from 62 to 2,724; the total Boolean variables were between 142 and 2,844; the CNF variables between 1,148 and 115,915; the CNF clauses between 6,207 and 8,395,649; the decisions made by the SAT-checker Siege were between 5,000 and 167,000,000, while the conflicts that it resolved were between 2,000 and 15,000,000; and the total verification time was between 0.18 seconds and 41,886 seconds (i.e., 11.6 hours).

Table 2 summarizes the results when abstracting the top-level g-equations, and using the nested-*ITE* scheme to eliminate the applications of predicate *abs_equality*. The E_i Boolean variables—introduced when eliminating predicate *abs_equality*—ranged between 47 and 3,600, while the total number of Boolean variables increased accordingly. Three of the benchmarks—2dlx_cc_mc_ex_bp, 9vliw_mc_ex_bp, and xscale—required fewer CNF variables compared with the e_{ij} encoding, with a reduction of 38% for xscale. Five of the benchmarks had fewer CNF clauses relative to the e_{ij} encoding—with a reduction of approximately 50% in the case of xscale and 12pipe, and a 94% reduction of the transitivity clauses for xscale.

Table 1. Results from the e_{ij} encoding

Processor	Boolean Variables		CNF Vars	CNF Clauses		SAT-Checker Siege		CPU Time [sec]			
	e_{ij}	Total		Trans	Total	decisions	conflicts	TLSim	EVC	SAT	Total
1dlx_c_mc_ex_bp	62	142	1,148	0	6,207	5×10^3	2×10^3	0.04	0.04	0.1	0.18
2dlx_cc_mc_ex_bp	256	414	4,482	0	41,071	36×10^3	10×10^3	0.06	0.27	1.18	1.51
9vliw_mc_ex_bp	2,968	3,326	24,373	0	232,209	936×10^3	101×10^3	0.1	1.6	37	38.7
xscale	2,387	2,669	43,574	102,480	656,381	72×10^3	27×10^3	0.2	3.8	21	25
12pipe	2,724	2,844	115,915	0	8,395,649	167×10^6	15×10^6	0.4	57	41,829	41,886
8pipe_ooo	2,129	2,209	35,510	117,462	1,191,215	31×10^6	15×10^6	0.2	6	19,981	19,987

Table 2. Abstracting the top-level g-equations, and using the nested-*ITE* scheme. The speedup is the total time with the e_{ij} encoding divided by the new total time

Processor	Boolean Variables		CNF Vars	CNF Clauses		SAT-Checker Siege		CPU Time [sec]				Speedup
	E_i	Total		Trans	Total	decisions	conflicts	TLSim	EVC	SAT	Total	
1dlx_c_mc_ex_bp	47	128	1,188	0	6,415	5×10^3	1×10^3	0.04	0.04	0.04	0.12	1.50
2dlx_cc_mc_ex_bp	159	317	4,251	0	32,716	35×10^3	10×10^3	0.06	0.27	1	1.33	1.14
9vliw_mc_ex_bp	1,894	2,252	17,481	0	167,567	936×10^3	122×10^3	0.1	1.9	41	43	0.90
xscale	333	643	26,857	5,907	326,041	52×10^3	19×10^3	0.2	2.5	12.6	15	1.67
12pipe	3,600	3,720	136,800	0	4,216,460	16×10^6	1×10^6	0.4	835	2,215	3,050	13.73
8pipe_ooo	2,157	2,638	42,365	134,670	1,021,721	4×10^6	1×10^6	0.2	53	1,342	1,395	14.33

Table 3. Abstracting the top-level g-equations, and using the Ackermann-constraint scheme. The speedup is the total time with the e_{ij} encoding divided by the new total time

Processor	Boolean Variables		CNF Vars	CNF Clauses		SAT-Checker Siege		CPU Time [sec]				Speedup
	E_i	Total		Trans	Total	decisions	conflicts	TLSim	EVC	SAT	Total	
1dlx_c_mc_ex_bp	47	128	1,254	0	6,592	5×10^3	1×10^3	0.04	0.06	0.05	0.15	1.20
2dlx_cc_mc_ex_bp	159	317	4,627	0	33,756	33×10^3	9×10^3	0.06	0.27	1.16	1.49	1.01
9vliw_mc_ex_bp	1,894	2,252	20,168	0	175,593	848×10^3	115×10^3	0.1	1.95	40	42	0.92
xscale	333	643	27,839	5,907	328,924	44×10^3	19×10^3	0.2	2.7	13.4	16	1.56
12pipe	3,600	3,720	192,105	0	4,382,554	17×10^6	1×10^6	0.4	839	3,699	4,538	9.23
8pipe_ooo	2,157	2,638	73,680	134,670	1,128,542	4×10^6	0.9×10^6	0.2	52	2,001	2,053	9.74

Table 4. Using the heuristic for partial transitivity, abstracting the top-level g-equations, and applying the nested-*ITE* scheme. The speedup is the total time with the e_{ij} encoding divided by the new total time

Processor	Boolean Variables		CNF Vars	CNF Clauses		SAT-Checker Siege		CPU Time [sec]				Speedup
	E_i	Total		Trans	Total	decisions	conflicts	TLSim	EVC	SAT	Total	
xscale	330	612	26,826	4,089	324,223	52×10^3	23×10^3	0.2	2.3	17.1	19.6	1.28
8pipe_ooo	1,684	1,764	41,491	2,772	889,823	4×10^6	0.9×10^6	0.2	25	973	998	20.03

The conceptually simpler solution space, resulting from the special interpreted predicate *abs_equality*, reduced the number of decisions for the last 3 benchmarks by up to an order of magnitude—in the case of 12pipe, the decisions went from 167 million down to 16 million, and the conflicts from 15 million down to 1 million, speeding up the verification 13.73×; in the case of 8pipe_ooo, the decisions were reduced from 31 million to 4 million, and the conflicts from 15 million to 1 million, with the speedup being 14.33 times. Note that the EVC time for translation to SAT increased significantly for the two most complex benchmarks—from 57 seconds to 835 seconds (14.6×) in the case of 12pipe, and from 6 seconds to 53 seconds (8.8×) in the case of 8pipe_ooo—but that was more than offset by the dramatic reduction in the SAT time.

Using Ackermann constraints instead of nested *ITE*s when enforcing reflexivity, functional consistency, and functional symmetry—see Table 3—required up to 74% more CNF variables, and up to 10% more clauses in the case of 8pipe_ooo, resulting in smaller speedups of 9.23× for 12pipe, and 9.74× for 8pipe_ooo.

Applying the heuristic for partial transitivity when formally verifying the two correct benchmarks that require transitivity—see Table 4—resulted in a 98% reduction in the number of transitivity clauses, and a 13% reduction in the total number of clauses for 8pipe_ooo, increasing the speedup to 20× for that model.

The above benchmarks do not require reflexivity of equality, since the g-equations are between source and destination register identifiers, which are separate instruction fields. However, in the M•CORE processor [15], a register identifier is used as both a source and destination register for the same instruction. Modifying both 12pipe and 8pipe_ooo, so that one of the source registers also served as destination register for the same instruction, resulted in automatically added reflexivity constraints, since the symbolic conditions for enforcing reflexivity did not simplify to *false* in EVC. However, those benchmarks also passed the safety check without reflexivity, since it is impossible for a register to be compared with itself in a correct implementation.

The mechanism for enforcing reflexivity was tested by modifying 8pipe_ooo to require this property after the model was extended with:

$t \leftarrow ITE(new\_var, a, b)$
$f_1 \leftarrow (t = a)$
$f_2 \leftarrow (t = b)$
$f_3 \leftarrow f_1 \vee f_2$

where a and b are arbitrary terms, *new_var* is a new Boolean variable, and formula f_3 was used as additional enabling condition in the forwarding logic of the processor. Note that when $t = a$ and $t = b$ are abstracted with *abs_equality*, and the property of reflexivity is enforced, then f_3 will evaluate to *true*, since f_1 will be *true* when *new_var* is *true*, while f_2 will be *true* when *new_var* is *false*. However, without reflexivity, f_3 will evaluate to a symbolic expression that will not be constrained to evaluate to *true*, and the modified forwarding logic will be incorrect. When reflexivity was not enforced, the SAT-checker Siege took 12 seconds to find a counterexample. However, with reflexivity constraints added automatically, only for the applications of *abs_equality* where the conditions for enforcing reflexivity (i.e., the syntactic equality between the two arguments) do not simplify to *false*, Siege took time comparable to that for the original 8pipe_ooo.

Similarly, the mechanism for enforcing transitivity was tested with a variant of 8pipe_ooo. One level of the forwarding logic—where a result is forwarded to the ALU if a source register *src* equals a destination register *dest*—was modified to:

$$regs\_equal\_original \leftarrow (src = dest)$$
$$f_1 \leftarrow (t = src) \wedge (t = dest)$$
$$regs\_equal \leftarrow f_1 \vee regs\_equal\_original$$

where t is an arbitrary term, such that the new formula *regs_equal* was used to control forwarding of data, as opposed to the original formula *regs_equal_original*. Note that if transitivity is enforced, then $((t = src) \wedge (t = dest)) \Leftrightarrow (src = dest)$, i.e., $f_1 \Leftrightarrow regs\_equal\_original$, so that $regs\_equal \Leftrightarrow regs\_equal\_original$, and the modified processor will function like the original, where formula *regs_equal_original* is used to control forwarding. However, without transitivity, f_1 may evaluate to *true* when *regs_equal_original* evaluates to *false*, so that data may be forwarded incorrectly. With partial transitivity, a counterexample was found in 15 seconds, but with complete transitivity, validity was proved in time comparable to that for the original 8pipe_ooo.

To evaluate the efficiency of *abs_equality* when formally verifying incorrect models, 10 buggy variants of 12pipe were created. While *abs_equality* reduced the number of SAT decisions by up to 2.5×, the number of conflicts by up to 5×, and the SAT-checking time by up to 5× as well, the total time was always longer compared with the e_{ij} encoding (up to 7×), due to the much increased time for SAT translation.

6 Conclusions

The paper presented a method for automatic abstraction of equations in a logic of equality by using a special interpreted predicate that satisfies the properties of transitivity, reflexivity, syntactic functional consistency, and syntactic symmetry. This abstraction is both sound and complete. The abstraction reduced the number of CNF clauses by up to 50%, and sped up the formal verification by up to an order of magnitude relative to the e_{ij} method, where a Boolean variable is used to encode each unique low-level equation between term variables. A heuristic for partial transitivity resulted in additional speedup for correct benchmarks that need transitivity. Abstracting the top-level equations had better performance, due to the concise encoding of many low-level equations between term variables with a single Boolean variable, thus resulting in an order of magnitude reduction in the number of decisions and the number of conflicts, resolved by a SAT-checker when evaluating the Boolean correctness formula.

References

1. M.D. Aagaard, N.A. Day, and M. Lou, "Relating Multi-Step and Single-Step Microprocessor Correctness Statements," *Formal Methods in Computer-Aided Design (FMCAD '02)*, M.D. Aagaard, and J.W. O'Leary, *eds.*, LNCS 2517, Springer-Verlag, November 2002.
2. M.D. Aagaard, B. Cook, N.A. Day, and R.B. Jones, "A Framework for Superscalar Microprocessor Correctness Statements," Software Tools for Technology Transfer, 2002.
3. W. Ackermann, *Solvable Cases of the Decision Problem*, North-Holland, 1954.
4. C. Barrett, D. Dill, and A. Stump, "Checking Satisfiability of First-Order Formulas by Incremental Translation to SAT," *Computer-Aided Verification (CAV '02)*, E. Brinksma, and K.G. Larsen, *eds.*, LNCS 2404, Springer-Verlag, July 2002, pp. 236–249.

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 (TOCL), Vol. 2, No. 1 (January 2001), pp. 93–134.
6. R.E. Bryant, and M.N. Velev, "Boolean Satisfiability with Transitivity Constraints," ACM Transactions on Computational Logic (TOCL), Volume 3, Number 4 (October 2002).
7. J.R. Burch, and D.L. Dill, "Automated Verification of Pipelined Microprocessor Control," *Computer-Aided Verification* (*CAV '94)*, D.L. Dill, *ed.*, LNCS 818, Springer-Verlag, 1994.
8. J.R. Burch, "Techniques for Verifying Superscalar Microprocessors," *33rd Design Automation Conference (DAC '96)*, June 1996, pp. 552–557.
9. A. Goel, K. Sajid, H. Zhou, A. Aziz, and V. Singhal, "BDD Based Procedures for a Theory of Equality with Uninterpreted Functions," *Computer-Aided Verification (CAV '98)*, A.J. Hu, and M.Y. Vardi, *eds.*, LNCS 1427, Springer-Verlag, June 1998, pp. 244–255.
10. J.L. Hennessy, and D.A. Patterson, *Computer Architecture: A Quantitative Approach*, 3rd edition, Morgan Kaufmann Publishers, San Francisco, CA, 2002.
11. Intel Corporation, *IA-64 Application Developer's Architecture Guide*, May 1999. http://developer.intel.com/design/ia-64/architecture.htm .
12. Intel Corporation, *Intel XScale Technology*, http://www.intel.com/design/intelxscale/.
13. S. Lahiri, C. Pixley, and K. Albin, "Experience with Term Level Modeling and Verification of the M•CORE™ Microprocessor Core," *6th Annual IEEE International Workshop on High Level Design, Validation and Test (HLDVT '01)*, November 2001, pp. 109–114.
14. D. Le Berre, and L. Simon, "Results from the SAT'03 SAT Solver Competition," *6th International Conference on Theory and Applications of Satisfiability Testing (SAT '03)*, May 2003. http://www.lri.fr/~simon/contest03/results/.
15. Motorola, Inc., *M•CORE™: microRISC Engine Programmer's Refernce Manual*, 1997.
16. A. Pnueli, Y. Rodeh, O. Strichman, and M. Siegel, "The Small Model Property: How Small Can It Be?", Journal of Information and Computation, Vol. 178, No. 1 (October 2002).
17. L. Ryan, Siege SAT Solver v.3, http://www.cs.sfu.ca/~loryan/personal/.
18. S.A. Seshia, S.K. Lahiri, and R.E. Bryant, "A Hybrid SAT-Based Decision Procedure for Separation Logic with Uninterpreted Functions," *40th Design Automation Conference (DAC '03)*, June 2003, pp. 425–430.
19. H. Sharangpani, and K. Arora, "Itanium Processor Microarchitecture," IEEE Micro, Vol. 20, No. 5 (September–October 2000), pp. 24–43.
20. S.K. Srinivasan, and M.N. Velev, "Formal Verification of an Intel XScale Processor Model with Scoreboarding, Specialized Execution Pipelines, and Imprecise Data-Memory Exceptions," *Formal Methods and Models for Codesign (MEMOCODE '03)*, June 2003.
21. M.N. Velev, and R.E. Bryant, "Superscalar Processor Verification Using Efficient Reductions of the Logic of Equality with Uninterpreted Functions to Propositional Logic," *Correct Hardware Design and Verification Methods (CHARME '99)*, L. Pierre, and T. Kropf, *eds.*, LNCS 1703, Springer-Verlag, September 1999, pp. 37–53.
22. M.N. Velev, and R.E. Bryant, "Formal Verification of Superscalar Microprocessors with Multicycle Functional Units, Exceptions, and Branch Prediction," *37th Design Automation Conference (DAC '00)*, June 2000, pp. 112–117.
23. M.N. Velev, "Formal Verification of VLIW Microprocessors with Speculative Execution," *Computer-Aided Verification (CAV '00)*, E.A. Emerson and A.P. Sistla, *eds.*, LNCS 1855, Springer-Verlag, July 2000, pp. 296–311.
24. M.N. Velev, "Automatic Abstraction of Memories in the Formal Verification of Superscalar Microprocessors," *Tools and Algorithms for the Construction and Analysis of Systems (TACAS '01)*, T. Margaria and W. Yi, *eds.*, LNCS 2031, Springer-Verlag, April 2001.
25. M.N. Velev, and R.E. Bryant, "EVC: A Validity Checker for the Logic of Equality with Uninterpreted Functions and Memories, Exploiting Positive Equality and Conservative Transformations," *Computer-Aided Verification (CAV '01)*, G. Berry, H. Comon, and A. Finkel, *eds.*, LNCS 2102, Springer-Verlag, July 2001, pp. 235–240.
26. M.N. Velev, and R.E. Bryant, "Effective Use of Boolean Satisfiability Procedures in the Formal Verification of Superscalar and VLIW Microprocessors," Journal of Symbolic Computation (JSC), Vol. 35, No. 2 (February 2003), pp. 73–106.
27. M.N. Velev, "Integrating Formal Verification into an Advanced Computer Architecture Course," *ASEE Annual Conference & Exposition*, June 2003.
28. M.N. Velev, "Collection of High-Level Microprocessor Bugs from Formal Verification of Pipelined and Superscalar Designs," *International Test Conference (ITC)*, October 2003.