Appears in the proceedings of the *22nd International Conference on Computer Design (ICCD '04)*, October 2004, pp. 119–124.

Comparative Study of Strategies for Formal Verification of High-Level Processors

Miroslav N. Velev
mvelev@ece.cmu.edu
http://www.ece.cmu.edu/~mvelev

Abstract

Compared are different methods for evaluation of formulas expressing microprocessor correctness in the logic of Equality with Uninterpreted Functions and Memories (EUFM) by translation to propositional logic, given recently developed efficient Boolean-to-CNF translations, in order to identify the best overall translation strategy from EUFM to CNF. The translation from EUFM to propositional logic is done by exploiting the property of Positive Equality, allowing us to treat most of the abstract word-level values as distinct constants while performing complete formal verification. For EUFM formulas from correct microprocessors, the best translation was by using the e_{ij} encoding of g-equations (dual-polarity equations), the nested-ITE scheme for elimination of uninterpreted predicates, preserving the ITE-tree structure of equation arguments, and Boolean-to-CNF translation by encoding the unobservability of logic blocks by merging them with adjacent gates on the only path to the primary output. For EUFM formulas from buggy microprocessors, the best translation was by using the e_{ij} encoding of g-equations, the Ackermann scheme for elimination of uninterpreted predicates, preserving the ITE-tree structure of equation arguments, and Boolean-to-CNF translation by applying optimizations to reduce the number of clauses—merging of ITE-trees with one level of their AND/OR leaves, and exploiting the polarity of gates and logic blocks to reduce the number of their clauses.

1. Introduction

Every time the design of computer systems was shifted to a higher level of abstraction, productivity increased. The logic of Equality with Uninterpreted Functions and Memories (EUFM) [8]—see Sect. 2—allows us to abstract functional units and memories, while completely modeling the control path of a processor. Restricting the modeling style for defining high-level processors with EUFM [27][28] resulted in correctness formulas where most of the terms (abstracted word-level values) appear only in positive equations (equality comparisons). Such terms can be treated as distinct constants [5], thus significantly pruning the solution space, and resulting in orders of magnitude speedup of the formal verification; this property is called *Positive Equality*. The modeling restrictions, together with techniques to model multicycle functional units, exceptions, and branch prediction [29], allowed our tool flow that exploits Positive Equality [31] to be used to formally verify a model of the M•CORE processor at Motorola [13], and detect three bugs, as well as corner cases that were not fully implemented.

Recent dramatic improvements in SAT-solvers [11][18][21]—see [16][32] for comparative studies—significantly sped up the solving of Boolean formulas generated in formal verification of high-level microprocessors [32]. However, as found in [32] and also confirmed in Sect. 5, the new efficient SAT-solvers would not have scaled for solving these Boolean formulas if not for the property of Positive Equality that results in at least 5 orders of magnitude speedup when formally verifying complex dual-issue superscalar processors. Efficient translations to CNF [34][35][36][37], exploiting the special structure of EUFM formulas produced with the modeling restrictions, resulted in additional speedup of up to 2 orders of magnitude.

The paper's contribution is a comparison of translations from EUFM to propositional logic, given recently developed efficient Boolean-to-CNF translations [34][35][36][37], in order to identify the best overall translation strategy from EUFM to CNF.

2. Background

2.1 Translation From EUFM to Propositional Logic

The syntax of EUFM [8] includes *terms* and *formulas*. Terms are used to abstract word-level values of data, register identifiers, memory addresses, and 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*, *term1*, *term2*) will evaluate to *term1* if *formula = true*, and to *term2* if *formula = false*. The syntax for terms can be extended to model memories by means of the interpreted functions *read* and *write* [8][30] that satisfy the *forwarding property* of the memory semantics—that a *read* gets the data value written by the most recent *write* to the same address, or the value from the initial memory state otherwise. Formulas are used to model the control path of a microprocessor, and to express the correctness condition. A formula can be an Uninterpreted Predicate (UP) applied to a list of argument terms, a Boolean variable, an ITE operator selecting between two argument formulas based on a controlling formula, or an *equation* (equality comparison) of two terms. Formulas can be negated and combined by 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 only the property of *functional consistency*—that equal inputs to the UF (UP) produce equal output values.

Syntactic equality is the subset of complete equality where a term variable is equal only to itself, and we will use $=_{SYN}$ to denote it. The syntactic equality between nested-*ITE* expressions that select term variables, is evaluated by pushing the equality to the *ITE* leaves, until each equation is between term variables and so can be replaced with their syntactic equality. For example, if *a* and *b* are syntactically distinct term variables, and *f* is a formula, then *ITE*(*f*, *a*, *a*) $=_{SYN}$ *a* will be evaluated by first being transformed to *ITE*(*f*, (*a* $=_{SYN}$ *a*), (*a* $=_{SYN}$ *a*)), and then replacing each low-level syntactic equality with *true* since both low-level equations compare the same term variable with itself, resulting in *ITE*(*f*, *true*, *true*), i.e., *true*; however, *ITE*(*f*, *a*, *b*) $=_{SYN}$ *a* will become *ITE*(*f*, (*a* $=_{SYN}$ *a*), (*b* $=_{SYN}$ *a*)), where (*b* $=_{SYN}$ *a*) will be replaced with *false* since *a* and *b* are syntactically distinct term variables, resulting in *ITE*(*f*, *true*, *false*), i.e., *f*.

Restrictions on the style for describing high-level processors

[27][28] reduced the number of terms that appear in both positive and negated equations (called *g-terms* for general terms), and increased the number of terms that appear only in positive equations (called *p-terms* for positive terms). The property of Positive Equality [27][28] allows us to treat syntactically different p-terms as not equal when evaluating the validity of an EUFM formula, thus achieving dramatic simplifications and orders of magnitude speedup (see [5] for correctness proof).

Applications of the same UF or UP are eliminated with nested ITEs [28]. For example, if $p(a_1, b_1)$, $p(a_2, b_2)$, and $p(a_3, b_3)$ are three applications of UP p, where a_1, b_1, a_2, b_2, a_3, and b_3 are terms, then the first application will be eliminated with a new Boolean variable c_1, the second with $ITE((a_2 = a_1) \wedge (b_2 = b_1), c_1, c_2)$, where c_2 is a new Boolean variable, and the third with $ITE((a_3 = a_1) \wedge (b_3 = b_1), c_1, ITE((a_3 = a_2) \wedge (b_3 = b_2), c_2, c_3))$, where c_3 is a new Boolean variable. That is, the second, third, and any subsequent applications of the UP are eliminated with *ITE-chains* that enforce functional consistency. The same method for eliminating UFs and UPs is used in [14][15][23][31]. Alternatively, functional consistency can be enforced with Ackermann constraints [1]—the three applications of the UP will be replaced with the new Boolean variables c_1, c_2, and c_3; then, the functional consistency of the second application of the UP with respect to the first will be enforced by extending the resulting formula with the constraint $(a_2 = a_1) \wedge (b_2 = b_1) \Rightarrow (c_2 = c_1)$, with such constraints added for each pair of applications of that UP. This method for enforcing functional consistency is used in [3][12][20][26][38], but does not result in ITE-trees, and so will not benefit from the CNF translations in Sect. 2.3, and 3. ITE-trees also result after eliminating a *read* from a sequence of *write*s by accounting for the forwarding property of the memory semantics, and from modeling conditional instruction flow.

After the UFs are eliminated, the terms consist of only ITE operators and term variables. A *low-level equation* is one where both arguments are term variables. A *top-level equation* is one where at least one of the arguments is a nested-*ITE* expression having term variables as leaves. In earlier EUFM decision procedures that exploit Positive Equality [14][15][23][31], top-level equations are eliminated by pushing the equations to the ITE leaves, and replacing the original equation with a disjunction of formulas. For example, given terms $ITE(c_1, a_1, a_2)$ and $ITE(c_2, b_1, b_2)$, where c_1 and c_2 are formulas, and a_1, a_2, b_1, and b_2 are term variables, the equation $ITE(c_1, a_1, a_2) = ITE(c_2, b_1, b_2)$ will be replaced with the formula $c_1 \wedge c_2 \wedge (a_1 = b_1) \vee c_1 \wedge \neg c_2 \wedge (a_1 = b_2) \vee \neg c_1 \wedge c_2 \wedge (a_2 = b_1) \vee \neg c_1 \wedge \neg c_2 \wedge (a_2 = b_2)$. Note the structure of the resulting formula—*a disjunction of conjunctions of formulas*, such that the ITE-tree structure of the original equation arguments is lost.

Equations between g-term variables (g-equations) can be either *true* or *false*, and can be encoded with Boolean variables by the following methods:

***e*$_{ij}$ encoding [10].** After the top-level g-equations are eliminated, each low-level g-equation, $g_i = g_j$, where g_i and g_j are g-term variables, is replaced by a new Boolean variable e_{ij}. Transitivity of equality, i.e., the property $(g_i = g_j) \wedge (g_j = g_k) \Rightarrow (g_i = g_k)$ has to be enforced additionally, e.g., by triangulating the equality comparison graph of the e_{ij} variables that affect the final Boolean formula and then enforcing transitivity for each of the resulting triangles [7]. The triangulation is done iteratively, in a greedy manner, such that at each step: nodes of degree 1 and their single edges are removed, since such nodes are not part of cycles for which transitivity of equality has to hold; the node of the smallest degree $n \geq 2$ is found; up to n-1 extra edges are added, if they do not exist already, in order to form n-1 triangles with the node's edges; the node and its edges are removed, and the procedure is applied to the remaining nodes by considering the newly added edges; finally, the original and the extra edges are put together to form the triangulated equality comparison graph. Although not every correct microprocessor requires transitivity for its correctness proof, that property is needed in order to avoid false negatives for buggy designs or for processors that do need transitivity.

small-domain encoding [20]. Top-level g-equations are again eliminated. Every 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 with which it can be transitively compared for equality with. If there are N constants in the set for a g-term variable, those can be indexed with $\lceil \log_2(N) \rceil$ new Boolean variables that will be used to control nested *ITE* operators selecting a mapping of the g-term variable to a constant in the set. Then, two g-term variables will be equal if their indexing variables simultaneously select the same common constant. Hence, g-term variables in a cycle can be equal if they simultaneously evaluate to the same common constant, so that transitivity of equality is automatically enforced in this encoding. Depending on the structure of the equality comparison graph, the small-domain encoding might introduce fewer primary Boolean variables than the e_{ij} encoding. That would mean a smaller search space. However, now many g-equations will get replaced by a Boolean formula—a disjunction of conjunctions, each consisting of many Boolean variables or their complements, and encoding the possibility that the two g-term variables evaluate to the same common constant. In contrast, in the e_{ij} encoding, a g-equation always gets replaced by a single Boolean variable.

interpreted predicate *abs_equality*() [33]. Each top-level g-equation is automatically abstracted with an application of a special interpreted predicate *abs_equality*() that satisfies the properties of transitivity, reflexivity, syntactic symmetry, and syntactic functional consistency. This abstraction is sound and complete. To adopt the nested-*ITE* scheme for eliminating the applications of *abs_equality*(), 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. Alternatively, syntactic symmetry and syntactic functional consistency can be enforced by a modified version of the Ackermann scheme for functional consistency [1], but that results in worse performance than the above modified nested-*ITE* scheme. Transitivity is enforced by triangulating the top-level equality-comparison graph.

2.2 Conventional Boolean-to-CNF Translation

A *primary CNF variable* is one representing the value of a primary input, i.e., input of the original Boolean circuit. An *auxil-*

iary CNF variable is one representing the value of a gate output. In general, the translation of Boolean formulas to CNF is exponential. However, by introducing a new CNF variable for the output of every logic gate, and imposing constraints that preserve the function of that gate [25], we get a satisfiability-equivalent CNF. Both the size of the CNF and the complexity of the translation are linear in the size of the original Boolean formula.

Instead of explicitly translating the inverters (NOT gates), we can subsume them in their fanout gates [19], by replacing all instances of the CNF variable for the inverter output with the negated CNF variable for the inverter input, thus eliminating the output variable and the 2 clauses for each inverter.

2.3 Translation from Propositional Logic to CNF by Merging ITE-Trees and Other Gate Groups

We can preserve the *ITE*-tree structure of equation arguments when pushing the equations to the leaves [35], producing Boolean formulas with many *ITE*-trees. That is, the equation $ITE(c_1, a_1, a_2) = ITE(c_2, b_1, b_2)$ will be replaced with the formula $ITE(c_1, ITE(c_2, a_1 = b_1, a_1 = b_2), ITE(c_2, a_2 = b_1, a_2 = b_2))$. Then, an *ITE*-tree can be translated to CNF with a unified set of clauses [35], without intermediate variables for outputs of *ITE*s inside the tree. For every path from a non-controlling input of the tree, we introduce 2 clauses: the first expressing the condition that if the input is *true* and is selected to appear at the tree output, *o*, by a corresponding assignment to controlling formulas of ITEs that are ahead in the tree, then the tree output should be *true*; the second expressing the condition that if the input is *false* and selected, then the tree output, *o*, should be *false*.

ITE-trees can be further merged with 1 or more levels of their AND/OR leaves that have fanout count of 1. We can also merge other gate groups [34], e.g., AND/OR→ITE (an ITE with an AND or OR as its then- or else-input, or a different AND/OR gate at each of these inputs), AND/ITE→OR, and OR/ITE→AND, but that results in minor additional improvements if ITE-trees are merged [35]. Note that a driven gate may have many input gates with fanout count of 1. Then, we can choose which one to merge by using a variant of the FANIN heuristic [17] for BDD-variable ordering—selecting the input gate with highest topological level. The motivation is to shorten the longest path for BCP from a primary input to the output of the driven gate. Thus, if the heuristic is applied to many groups, we could significantly shorten many paths for BCP from primary inputs to the output of the circuit.

We will call a *logic block* any group of connected gates with only one output signal leaving the block. The functionality of every logic block can be expressed with a set of implications. A *positive* (*negative*) *implication* is one that implies a value of *true* (*false*) for the block output.

3. Encoding the Unobservability of Logic Blocks

3.1 Encoding Local Unobservability of Logic Blocks by Merging Them with Adjacent Gates

The *controlling (input) value* of an AND or OR gate uniquely determines the value of the gate, regardless of the values of the other inputs, i.e., the controlling value of an AND is 0 (*false*) and of an OR is 1 (*true*). The non-controlling value of an AND (OR) is the negation of that gate's controlling value, i.e., 1 (0).

We can account for the local unobservability context of a logic block by merging it with adjacent gates that are on the only path from the block output to the circuit output [37]. Then, if one of those gates has a controlling value on an input that is not along this path, all clauses for the logic block will get satisfied, thus allowing a conventional CNF-based SAT-solver to exploit unobservability. Note that when a logic block is merged with an adjacent AND gate, the other inputs to the AND will affect only the positive implications of the new block, while the negative implications will be the same as in the original block, since the negative implications of the original block determine conditions for a controlling value of 0 at the AND input driven by the original block. Similarly, if a logic block is merged with an adjacent OR gate, the other inputs to the OR will affect only the negative implications of the new block. If a logic block is merged with an adjacent ITE, where the block output drives either the then-input or the else-input, then the ITE controlling input will affect both the positive and the negative implications of the new block. Each of the other inputs of a merged AND (OR) gate will determine an implication for the new block's output for cases when the input has a controlling value and thus uniquely determines the output value of that AND (OR) gate, i.e., the output value of the new block. For a merged ITE, the other non-controlling input, which is not along the merged path, will result in 2 implications—for the cases when the input is *true* (*false*) and its value is selected to propagate to the output of the new block. By merging a logic block with an adjacent gate, we will reduce the variables by 1 (eliminating the variable for the output of the original logic block), and the clauses by 2 (eliminating the 2 clauses for the gate output as a function of the logic block's output value). The new block can be similarly merged with more gates until reaching a fanout point. Further merging the block with gates along each of the fanout paths will require replication of the constraints for the block's functionality for each of those paths, and so will increase the number of clauses and slow down the SAT-solving.

3.2 Using Unobservability Variables to Encode the Local and Global Unobservability of Logic Blocks

An alternative way to encode the local unobservability of a logic block is to introduce a *CNF unobservability variable* [37], representing the conditions when the block's output is unobservable at the primary output. Such conditions depend on values of inputs to nearby gates situated on a fanout-free partial path from the block output toward the primary output. A CNF logic variable is still used to represent the logic value of the block output. The unobservability variable for a logic block is disjuncted to each clause for that block, so that when the unobservability variable is 1—meaning that the logic block *is not* observable at the primary output—all clauses for that block will be satisfied and a conventional CNF-based SAT-solver will have more freedom in assigning values to the variables in these clauses, possibly leaving some of those variables unassigned. A value of 0 assigned to a CNF unobservability variable means that the block's output value *may be* allowed to propagate to the end of the fanout-free partial path from the block output toward the primary output, and so may be observable at the primary output. In this paper, a CNF unobservability variable is introduced only for ITE-trees with fanout count of 1.

In order to more fully exploit the unobservability of a logic block, we can account for its global unobservability [36]. An *unobservability check-point* is a signal that has an associated CNF unobservability variable. The *nearest cutset of unobservability check-points* for a block consists of the unobservability check-points that are each situated on a different path from the block output to the primary output, covering all such paths, such that each of these check-points is closest to the block compared to other unobservability check-points on the same path from the block output to the primary output.

A new constraint for unobservability of each block is added, in order to express the condition that if all of the nearest unobservability check-points are unobservable, then the block is unobservable, since each path from the block output to the primary output will go through one of those unobservability check-points. We also need to extend the constraint for local observability of each block at the end of its fanout-free partial path toward the primary output, by accounting for the observability of each of the nearest unobservability check-points. If the block is observable at the end of its partial path toward the primary output, and one of the nearest unobservability check-points is observable, then the block is considered observable.

4. Exploiting the Polarity of Logic Blocks to Reduce the Number of Their Clauses

A logic block appears in only positive (negative) polarity if all the paths from its output to the Boolean circuit output have an even number of negations or no negations (an odd number of negations), and in dual polarity if some paths have an even number of negations or no negations, while others have an odd number of negations. The following theorem [37] is a generalization of a theorem by Plaisted and Greenbaum [19] for single gates:

THEOREM. *Keeping only the clauses from positive (negative) implications for logic blocks that appear in only negative (positive) polarity results in a satisfiability-equivalent CNF formula.*

5. Comparison of Translation Strategies on EUFM Formulas from Correct Processors

The goal of this section is to compare the three encodings on g-equations—e_{ij}, small-domain, and the special interpreted predicate *abs_equality*()—on EUFM formulas from correct processors, given the CNF translations from Sect. 2.3, 3, and 4. Also, since the small-domain encoding replaces each g-term variable with an ITE-tree selecting a constant from the set assigned to that variable, studied is the benefit from preserving the ITE-tree structure of the expressions replacing g-term variables.

The computer used for the experiments was a Dell OptiPlex GX260 having a 3.06-GHz Intel Pentium 4 processor with a 512-KB on-chip L2 cache, 2 GB of physical memory, and running Red Hat Linux 9.0. The Boolean correctness formulas produced by exploiting Positive Equality were evaluated with the SAT-solver siege_v4 [21][22]. The experiments were to formally verify safety of the benchmarks: `ooo_engine6`, an out-of-order processor with a 6-entry reorder buffer, 6 reservation stations, register renaming, and register-register ALU instructions; `1dlx_m_iq29`, a single-issue pipelined DLX with multicycle functional units, exceptions, branch prediction, and a 29-entry instruction queue; `9vliw_5_iq4`, a 9-wide, 5-stage VLIW processor that implements predicated execution, register remapping, advanced loads, and a 4-entry instruction queue; `14pipe`, a 14-wide, 5-stage pipelined processor with in-order execution, implementing register-register ALU and load instructions; and `9pipe_ooo`, a 9-wide, 5-stage pipelined processor with out-of-order execution, as well as ALU and load instructions. The number of instruction queue entries in these designs, or their issue widths were chosen so that the experiments for each design could complete for all of the encodings, given the available memory.

Compared were the following Boolean-to-CNF translations:

old translation—without preserving the ITE-tree structure of equation arguments but representing the equations with disjunctions of conjunctions (see Sect. 2.1), followed by conventional translation to CNF by subsuming inverters in the driven gates;

merging of ITE-trees—in a variant without merging the ITE-trees with AND/OR leaves that have fanout count of 1, and by preserving the ITE-tree structure of equation arguments—this strategy had the best performance in [35], where the experiments were based on the e_{ij} encoding and the nested-ITE scheme for UP elimination; two variants were also explored—by merging the ITE-trees with 1 level of AND/OR leaves that have fanout count of 1, and by merging the ITE-trees with 2 levels of AND/OR leaves that have fanout count of 1;

method (1)—the first method for encoding the unobservability of logic blocks (ITE-trees in the experiments) with fanout count of 1—by merging them with adjacent gates on the only path from the block output toward the primary output (see Sect. 3.1), after merging those ITE-trees, and by preserving the ITE-tree structure of equation arguments—this method had the best performance in [37], where the experiments were based on the e_{ij} encoding and the nested-ITE scheme for UP elimination.

The results are summarized in Table 1 for each of the three encodings of g-equations, as well as for the small-domain encoding with preserving the ITE-tree structure of expressions that replace the g-term variables ("small-domain + ITEs"). For two of the benchmarks, `1dlx_m_iq29` and `9vliw_5_iq4`, the e_{ij} encoding was also combined with the Ackermann scheme for eliminating UPs; the other three designs do not have functional units abstracted with UPs. Transitivity constraints were needed only for the out-of-order processors, `ooo_engine6` and `9pipe_ooo`, when the e_{ij} encoding or the interpreted predicate *abs_equality*() was used, such that for the latter only partial transitivity was enforced [33].

From the total verification times in Table 1, the e_{ij} encoding had the best performance on 4 of the 5 benchmarks—`ooo_engine6`, `1dlx_m_iq29`, `14pipe`, and `9pipe_ooo`—while the special interpreted predicate *abs_equality*() was best on the other benchmark—`9vliw_5_iq4`. Method (1) had the best performance on 4 of the 5 benchmarks, except on `ooo_engine6`, where best was the strategy of merging ITE-trees with 2 levels of leaves, resulting in 5× speedup compared to the other strategies of merging the ITE-trees. For benchmark `1dlx_m_iq29`, the e_{ij} encoding, combined with the Ackermann scheme for UP elimination, was marginally better than when combined with the nested-ITE scheme for UP elimination, but that was not the case for versions of the benchmark with longer instruction queues.

Given the advantage of the interpreted predicate *abs_equality*() on benchmark `9vliw_5_iq4`, compared to the e_{ij} encoding, when both are combined with Boolean-to-CNF translation with method (1)—see Table 1—would that advantage be preserved for more complex versions of the same benchmark with longer instruction queues? On models with between 6 and 10 instruction queue entries, the interpreted predicate *abs_equality*() took between 30% less time to 20% longer, but introduced 50% more CNF variables and more than twice the CNF clauses for the design with 10 instruction queue entries. Furthermore, when using *abs_equality*() for the models with 11, 12, and 13 instruction queue entries, the EUFM decision procedure EVC ran out of memory, while it could complete the translation to CNF when the e_{ij} encoding was used instead.

Therefore, from Table 1 and from the above paragraph, *the most efficient translation for EUFM formulas from correct designs is by using the e_{ij} encoding of g-equations, the nested-ITE scheme for elimination of uninterpreted predicates, preserving the ITE-tree structure of equation arguments, and Boolean-to-CNF translation with method (1).*

Experiments were also run to determine the benefit from preprocessing the CNF formulas generated in the experiments for

Table 1. Comparison of the three g-equation encodings on unsatisfiable Boolean formulas from correct processors—decisions and conflicts by the SAT-solver siege_v4, and total formal verification time with each encoding.

Processor	Encoding of g-equations	Decisions $\times 10^6$ / Conflicts $\times 10^6$					Time (TLSim + EVC + siege_v4) [sec]				
		old trans-lation	merging of ITE-trees				old trans-lation	merging of ITE-trees			
			w/o leaves	+ 1 level of leaves	+ 2 levels of leaves	Method (1)		w/o leaves	+ 1 level of leaves	+ 2 levels of leaves	Method (1)
`ooo_engine6`	e_{ij}	0.4 / 0.3	0.2 / 0.2	0.2 / 0.1	0.1 / 0.1	0.2 / 0.2	646	216	154	**43**	186
	small-domain	0.4 / 0.3	0.3 / 0.2	0.3 / 0.2	0.1 / 0.1	0.4 / 0.2	660	262	286	57	287
	small-domain + ITEs	0.4 / 0.3	0.3 / 0.2	0.3 / 0.2	0.1 / 0.1	0.4 / 0.2	712	239	261	49	277
	abs_equality()	1.1 / 0.9	1.0 / 0.8	0.8 / 0.6	0.1 / 0.1	1.0 / 0.8	1,956	1,248	951	124	1,232
`1dlx_m_iq29`	e_{ij}	4.5 / 0.3	3.7 / 0.2	4.3 / 0.2	4.4 / 0.2	3.3 / 0.1	1,289	248	262	241	232
	e_{ij} + Ackermann UPs	4.1 / 0.2	3.8 / 0.2	3.9 / 0.2	4.2 / 0.2	3.0 / 0.1	992	255	236	255	**222**
	small-domain	5.9 / 0.4	5.0 / 0.3	3.9 / 0.2	4.1 / 0.3	4.1 / 0.2	2,417	749	583	670	523
	small-domain + ITEs	11.8 / 0.3	5.3 / 0.3	5.4 / 0.2	4.4 / 0.2	4.1 / 0.2	2,237	534	469	355	381
	abs_equality()	4.1 / 0.3	4.1 / 0.2	5.3 / 0.2	4.5 / 0.2	3.9 / 0.2	1,767	548	628	562	479
`9vliw_5_iq4`	e_{ij}	46.7 / 1.7	31.2 / 0.9	38.0 / 1.1	36.7 / 1.0	25.4 / 0.8	3,205	784	910	853	598
	e_{ij} + Ackermann UPs	42.5 / 1.5	37.4 / 1.1	37.8 / 1.1	31.7 / 0.9	21.9 / 0.6	2,781	992	853	764	421
	small-domain	37.0 / 1.2	25.3 / 0.8	18.9 / 0.7	21.0 / 0.8	18.3 / 0.7	14,825	7,189	5,764	6,377	5,565
	small-domain + ITEs	103.2 / 0.9	49.3 / 0.7	43.6 / 0.6	47.5 / 0.7	32.1 / 0.5	17,200	2,706	1,955	2,296	1,542
	abs_equality()	32.3 / 1.0	34.9 / 1.1	32.6 / 1.0	35.0 / 1.1	15.1 / 0.5	1,610	920	870	843	**346**
`14pipe`	e_{ij}	52.8 / 1.3	30.6 / 0.8	45.1 / 1.3	34.2 / 0.7	29.9 / 0.8	10,758	744	1,381	731	**719**
	small-domain	52.8 / 5.0	13.7 / 1.3	20.0 / 1.5	20.5 / 1.5	13.5 / 1.2	65,506	5,062	7,218	7,417	4,885
	small-domain + ITEs	79.0 / 3.7	28.5 / 1.2	38.2 / 1.4	40.1 / 1.5	28.5 / 1.3	57,712	3,766	5,223	5,659	3,786
	abs_equality()	24.6 / 0.7	31.2 / 0.7	29.6 / 0.7	36.5 / 0.9	33.0 / 0.8	6,154	4,048	4,071	4,266	4,338
`9pipe_ooo`	e_{ij}	3.1 / 1.0	1.7 / 0.4	1.7 / 0.4	1.7 / 0.4	1.5 / 0.4	1,665	239	245	241	**222**
	small-domain	7.0 / 2.0	6.2 / 1.5	6.5 / 1.5	6.7 / 1.5	6.1 / 1.5	5,542	1,906	1,930	1,948	1,894
	small-domain + ITEs	10.1 / 1.6	7.8 / 1.0	8.2 / 1.0	8.4 / 1.0	8.3 / 1.2	4,321	820	936	955	941
	abs_equality()	3.9 / 0.5	4.1 / 0.6	4.8 / 0.7	4.8 / 0.6	4.8 / 0.7	594	317	344	295	330

Table 2. Comparison of translation strategies on satisfiable Boolean formulas from 10 buggy variants of processor `9vliw_5_iq6`.

Strategy for Translation from EUFM to CNF	CPU Time [sec]		
	Min.	Average	Max.
Method (0) + e_{ij}	70	1,220	2,321
Method (0a) + e_{ij}	24	803	1,920
Method (0a) + e_{ij} + Ackermann UPs	18	671	1,890
Method (0a) + *abs_equality*()	14	640	1,283
Method (1) + e_{ij}	76	1,357	2,909
Method (1a) + e_{ij}	14	654	1,498
Method (1a) + e_{ij} + Ackermann UPs	**25**	**530**	**1,030**
Method (1a) + *abs_equality*()	20	637	1,655
Method (2) + e_{ij}	132	1,216	3,043
Method (2a) + e_{ij}	35	579	1,149
Method (2a) + e_{ij} + Ackermann UPs	3	503	1,130
Method (2a) + *abs_equality*()	29	605	1,442
Method (2g) + e_{ij}	31	793	1,982
Method (2ga) + e_{ij}	3	665	1,357
Method (2ga) + e_{ij} + Ackermann UPs	25	580	1,149
Method (2ga) + *abs_equality*()	28	571	1,653
Running all of the above strategies in parallel and stopping when one solution found	3	310	708

`9vliw_5_iq4` and more complex variants with longer instruction queues and with more pipeline stages, by using the recently developed preprocessor NiVER [24] in a mode that allows it to increase the total literal count by 10. That number was used in experiments in [24], and led to speedups of up to 2–3× for SAT-solving of preprocessed versions of simpler CNF formulas from our earlier work. However, preprocessing the CNF formulas from the variants of `9vliw_5_iq4`—both the versions generated with the old translation, and the versions generated with method (1)—resulted in comparable or longer times, compared to SAT-solving the original CNF formulas without preprocessing.

Does Positive Equality still matter? Without Positive Equality—using the e_{ij} encoding for all equations, including p-equations, as originally done by Goel et al. [10], and applying method (1), i.e., the most efficient translation for EUFM formulas from correct designs—the formal verification of a correct dual-issue superscalar DLX processor with exceptions, multicycle functional units and branch prediction [29] did not complete in 110,000 seconds, compared to finishing in 1.1 second with Positive Equality. Thus, Positive Equality results in at least 5 orders of magnitude speedup for complex designs, even with state-of-the-art SAT-solvers and the best translation from EUFM to CNF.

6. Comparison of Translation Strategies on EUFM Formulas from Buggy Processors

In addition to method (1), compared were:

method (0)—merging of ITE-trees with one level of AND/OR leaves that have fanout count of 1;

method (2)—encoding only the local unobservability of logic blocks (ITE-trees) by using unobservability variables;

method (2g)—extension of method (2) by also encoding the global unobservability of logic blocks.

Each of these strategies was also implemented in variant **(a)** where the ITE-trees were merged with one level of their AND/OR leaves that have fanout count of 1, and the polarity of gates and logic blocks was exploited to reduce the number of their clauses, such that this optimization was applied to ITE-trees, ITE-trees merged with AND/OR leaves or with adjacent gates on the only path to the primary output, as well as to AND/OR→ITE, ITE→AND, and ITE→OR groups. Also evaluated were two modifications of variants (a)—by using the Ackermann scheme to eliminate the UPs instead of the nested-ITE scheme, and by using the special interpreted predicate *abs_equality*() to encode the g-equations instead of the e_{ij} encoding. Table 2 summarizes the results from 10 buggy models of processor `9vliw_5_iq6` (a variant of `9vliw_5_iq4` with a 6-entry instruction queue).

Best was method (1a), combined with the e_{ij} encoding of g-equations and the Ackermann scheme for UP elimination, since that strategy reduced the most the maximum CPU time. If sufficient CPUs are available for parallel runs of the tool flow with each of these 16 strategies, stopping the rest of the runs as soon as one returns a solution, we can reduce the maximum CPU time to about 70% of its value with method (1a), combined with the Ackermann scheme for UP elimination, as shown in the last row of Table 2, i.e., the speedup from parallel runs with different strategies is insignificant.

7. Related Work

In previous research [32], the e_{ij} encoding was found to outperform the small-domain encoding when formally verifying simpler microprocessors than those used here. A comparison of the e_{ij}, small-domain, and a pseudo-Boolean encoding of g-equations, based on a different set of benchmarks, is presented in [6]. A hybrid encoding of g-equations, combining the strengths of the e_{ij} and small-domain encodings, is presented in [6][23]. However, all these studies were done with conventional translation to CNF, and with less efficient SAT-solvers.

In the EUFM decision procedure used in this paper, the translation to CNF is done in a single step, by including all transitivity constraints if the e_{ij} encoding or the interpreted predicate *abs_equality*() are applied, i.e., the translation is *eager*, as is also the case in [6][23]. In *lazy translation* to SAT [2][3][4][9]—constraints are added incrementally to prevent recurrence of false counterexamples—but this significantly degrades the performance when deciding complex EUFM formulas [23].

8. Conclusions

Compared were different methods for translation from EUFM to propositional logic, given recently developed efficient Boolean-to-CNF translations. For formulas from correct microprocessors, the best translation was by using the e_{ij} encoding of g-equations, the nested-ITE scheme for UP elimination, preserving the ITE-tree structure of equation arguments, and Boolean-to-CNF translation with method (1)—by merging of ITE-trees and other gate groups, as well as merging of ITE-trees having fanout count of 1 with adjacent gates on the only path to the primary output. This strategy was most efficient for unsatisfiable Boolean formulas from correct designs, since the strategy exploits effectively the unobservability of logic blocks, and also results in higher decision priority for CNF variables that control ITEs at the top of ITE-trees, thus resulting in optimal learning and pruning of the solution space. For EUFM formulas from buggy microprocessors, the best translation was by using the e_{ij} encoding of g-equations, the Ackermann scheme for UP elimination, preserving the ITE-tree structure of equation arguments, and Boolean-to-CNF translation with method (1a)—extension of method (1) by also merging the ITE-trees with one level of their AND/OR leaves that have fanout count of 1, and exploiting the polarity of gates and logic blocks to reduce the number of their clauses. This strategy was most efficient for satisfiable Boolean formulas from buggy designs, since the strategy results in the greatest reduction in the number of clauses, thus making it easier for a SAT-solver to satisfy the resulting CNF formula.

Even with state-of-the-art SAT-solvers and the most efficient EUFM-to-CNF translation, Positive Equality still results in at least 5 orders of magnitude speedup for correct versions of complex dual-issue superscalar processors. Furthermore, the speedup is increasing with the complexity of the designs.

References

[1] W. Ackermann, *Solvable Cases of the Decision Problem*, North-Holland, Amsterdam, 1954.

[2] G. Audemard, P. Bertoli, A. Cimatti, A. Korniowicz, and R. Sebastiani, "A SAT Based Approach for Solving Formulas over Boolean and Linear Mathematical Propositions," *11th International Conference on Automated Deduction (CADE '02)*, LNCS 2392, Springer-Verlag, July 2002, pp. 195–210.

[3] C. Barrett, D. Dill, and A. Stump, "Checking Satisfiability of First-Order Formulas by Incremental Translation to SAT," *Computer-Aided Verification (CAV '02)*, LNCS 2404, July 2002.

[4] C. Barrett, and S. Berezin, "CVC Lite: A New Implementation of the Cooperating Validity Checker," *Computer-Aided Verification (CAV '04)*, LNCS, Springer-Verlag, July 2004.

[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, S.K. Lahiri, and S.A. Seshia, "Deciding CLU Logic Formulas via Boolean and Pseudo-Boolean Encodings," *Workshop on Constraints in Formal Verification (CFV '02)*, September 2002.

[7] R.E. Bryant, and M.N. Velev, "Boolean Satisfiability with Transitivity Constraints," ACM Transactions on Computational Logic (TOCL), Vol. 3, No. 4 (October 2002), pp. 604–627.

[8] J.R. Burch, and D.L. Dill, "Automated Verification of Pipelined Microprocessor Control," *Computer-Aided Verification (CAV '94)*, LNCS 818, Springer-Verlag, June 1994.

[9] L. de Moura, H. Rueß, and M. Sorea, "Lazy Theorem Proving for Bounded Model Checking over Infinite Domains," *11th International Conference on Automated Deduction (CADE '02)*, LNCS 2392, July 2002.

[10] 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)*, Springer-Verlag, June 1998.

[11] E. Goldberg, and Y. Novikov, "BerkMin: A Fast and Robust Sat-Solver," *Design, Automation, and Test in Europe (DATE '02)*, March 2002, pp. 142–149.

[12] R.B. Jones, D.L. Dill, and J.R. Burch, "Efficient Validity Checking for Processor Verification," *International Conference on Computer-Aided Design (ICCAD '95)*, 1995.

[13] S. Lahiri, C. Pixley, and K. Albin, "Experience with Term Level Modeling and Verification of the M•CORE™ Microprocessor Core," *High Level Design, Validation and Test (HLDVT '01)*, November 2001.

[14] S.K. Lahiri, S.A. Seshia, and R.E. Bryant, "Modeling and Verification of Out-of-Order Microprocessors in UCLID," *Formal Methods in Computer-Aided Design (FMCAD '02)*, LNCS 2517, November 2002.

[15] S.K. Lahiri, and R.E. Bryant, "Deductive Verification of Advanced Out-of-Order Microprocessors," *Computer-Aided Verification (CAV '03)*, LNCS, Springer-Verlag, July 2003.

[16] D. Le Berre, and L. Simon, "Results from the SAT'04 Solver Competition," *Theory and Applications of Satisfiability Testing (SAT '04)*, 2004.

[17] S. Malik, A.R. Wang, R.K. Brayton, and A. Sangiovani-Vincentelli, "Logic Verification Using Binary Decision Diagrams in a Logic Synthesis Environment," *International Conference on Computer-Aided Design*, 1988.

[18] M.W. Moskewicz, C.F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an Efficient SAT Solver," *38th Design Automation Conference (DAC '01)*, June 2001.

[19] D.A. Plaisted, and S. Greenbaum, "A Structure Preserving Clause Form Translation," Journal of Symbolic Computation (JSC), Vol. 2, 1985, pp. 293–304.

[20] 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).

[21] L. Ryan, Siege SAT Solver. http://www.cs.sfu.ca/~loryan/personal/

[22] L. Ryan, "Efficient Algorithms for Clause-Learning SAT Solvers," M.S. Thesis, Simon Fraser University, Canada, February 2004.

[23] S.A. Seshia, S.K. Lahiri, and R.E. Bryant, "A Hybrid SAT-Based Decision Procedure for Separation Logic with Uninterpreted Functions," *Design Automation Conference (DAC '03)*, 2003.

[24] S. Subbarayan, and D.K. Pradhan, "NiVER: Non Increasing Variable Elimination Resolution for Preprocessing SAT Instances," *Theory and Applications of Satisfiability Testing (SAT '04)*, May 2004.

[25] G.S. Tseitin, "On the Complexity of Derivation in Propositional Calculus," in *Studies in Constructive Mathematics and Mathematical Logic*, Part 2, 1968, pp. 115–125. Reprinted in J. Siekmann, and G. Wrightson, *eds., Automation of Reasoning*, Vol. 2, Springer-Verlag, 1983.

[26] O. Tveretina, and H. Zantema, "A Proof System and a Decision Procedure for Equality Logic," Technical Report, Department of Computer Science, Technical University of Eindhoven, 2003.

[27] M.N. Velev, and R.E. Bryant, "Exploiting Positive Equality and Partial Non-Consistency in the Formal Verification of Pipelined Microprocessors," *36th Design Automation Conference (DAC '99)*, June 1999.

[28] 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)*, LNCS 1703, Springer-Verlag, September 1999.

[29] M.N. Velev, and R.E. Bryant, "Formal Verification of Superscalar Microprocessors with Multicycle Functional Units, Exceptions, and Branch Prediction," *Design Automation Conference (DAC '00)*, 2000.

[30] 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, pp. 252–267.

[31] M.N. Velev, and R.E. Bryant, "TLSim and EVC: A Term-Level Symbolic Simulator and an Efficient Decision Procedure for the Logic of Equality with Uninterpreted Functions and Memories," *International Journal of Embedded Systems (IJES)*, 2004.

[32] 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, 2003.

[33] M.N. Velev, "Automatic Abstraction of Equations in a Logic of Equality," *Automated Reasoning with Analytic Tableaux and Related Methods (TABLEAUX '03)*, LNAI 2796, September 2003, pp. 189–206.

[34] M.N. Velev, "Efficient Translation of Boolean Formulas to CNF in Formal Verification of Microprocessors," *Asia and South Pacific Design Automation Conference (ASP-DAC '04)*, January 2004.

[35] M.N. Velev, "Exploiting Signal Unobservability for Efficient Translation to CNF in Formal Verification of Microprocessors," *Design, Automation and Test in Europe (DATE '04)*, February 2004, pp. 266–271.

[36] M.N. Velev, "Encoding Global Unobservability for Efficient Translation to SAT," *7th International Conference on Theory and Applications of Satisfiability Testing (SAT '04)*, May 2004.

[37] M.N. Velev, "Comparison of Schemes for Encoding Unobservability in Translation to SAT," submitted for publication.

[38] H. Zantema, and J.F. Groote, "Transforming Equality Logic to Propositional Logic," *4th International Workshop on First Order Theorem Proving (FTP '03)*, June 2003.