Efficient Translation of Boolean Formulas to CNF in Formal Verification of Microprocessors

Miroslav N. Velev

Department of Electrical and Computer Engineering
Carnegie Mellon University, Pittsburgh, PA 15213, U.S.A.
http://www.ece.cmu.edu/~mvelev
e-mail: mvelev@ece.cmu.edu

Abstract—The paper presents a method for translating Boolean formulas to CNF by identifying gates with fanout count of 1, and merging them with their fanout gate to generate a single set of equivalent CNF clauses. This eliminates the intermediate CNF variable for the output of the first gate, and reduces the number of CNF clauses, compared to the conventional translation to CNF, where each gate is assigned an output variable and is represented with a separate set of CNF clauses. Chains of nested ITE operators, where each ITE is used only as else-argument of the next ITE, are similarly merged and represented with a single set of clauses without intermediate variables. This method was applied to Boolean formulas from formal verification of microprocessors. The formulas require up to hundreds of thousands of variables and millions of clauses, when translated to CNF with the conventional approach. The best translation reduced the CNF variables by up to 2×; the SAT-solver decisions by up to 5×; the SAT-solver conflicts by up to 6×; and accelerated the SAT checking by up to 7.6× for unsatisfiable formulas, and 136× for satisfiable ones.

I. Introduction

The speed of SAT-solvers improved by orders of magnitude in the last couple of years [13][29][32]—see [25][43] for comparative studies. A hurdle to further improvements is the operation-intensive Boolean Constraint Propagation (BCP)—reflecting a CNF variable's assignment on all the clauses that contain that variable or its negation—where SAT-solvers spend up to 90% of their execution time [29]. Furthermore, big formulas increase the L2-cache misses of SAT-solvers [46], and require many expensive accesses to main memory.

The most common input format of SAT-solvers is Conjunctive Normal Form (CNF) [17]. In conventional translation of Boolean formulas to CNF [35] (see also [24][30]), a new CNF variable is introduced for the output of every logic gate, and a separate set of CNF clauses correlates that variable with the CNF variables for the inputs of the gate, given that gate's function. This paper studies translation to CNF by merging logic gates with *fanout count* of 1 (i.e., whose output is used only once) with the *fanout gate* that they drive, and expressing the combined function of those gates with a single set of CNF clauses, thus eliminating the CNF variable for the output of the driving gate, and saving CNF clauses. The idea is similar to that in technology mapping—an NP-complete problem [19] that can be solved efficiently by heuristics minimizing a cost function. The heuristics used in the current paper reduce the number of CNF variables and clauses, and thus reduce both the required BCP and the resulting cache misses. The heuristics were implemented in the decision procedure EVC [42] for the logic of Equality with Uninterpreted Functions and Memories (EUFM) [8], and were evaluated on formulas from formal verification of microprocessors. EVC was previously used at Motorola [21] to formally verify a model of the M•CORE processor, and detected bugs.

II. Conventional Translation to CNF

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 (see Table I), we get an equivalent CNF representation [35]. Both the size of the resulting CNF and the complexity of the translation procedure are linear in the size of the original Boolean formula. In Table I, the ITE operator functions like a multiplexor, i.e., ITE(i, t, e) is equivalent to $i \wedge t \vee \neg i \wedge e$.

TABLE I
CONVENTIONAL TRANSLATION TO CNF

Logic Gate	Equivalent Constraints	CNF Clauses
$o \leftarrow \text{AND}(i_1, i_2, ..., i_n)$	$\neg i_1 \Rightarrow \neg o$ $\neg i_2 \Rightarrow \neg o$ ... $\neg i_n \Rightarrow \neg o$ $i_1 \wedge i_2 \wedge ... \wedge i_n \Rightarrow o$	$(i_1 \vee \neg o) \wedge$ $(i_2 \vee \neg o) \wedge$ ... $(i_n \vee \neg o) \wedge$ $(\neg i_1 \vee \neg i_2 \vee ... \vee \neg i_n \vee o)$
$o \leftarrow \text{OR}(i_1, i_2, ..., i_n)$	$i_1 \Rightarrow o$ $i_2 \Rightarrow o$ ... $i_n \Rightarrow o$ $\neg i_1 \wedge \neg i_2 \wedge ... \wedge \neg i_n \Rightarrow \neg o$	$(\neg i_1 \vee o) \wedge$ $(\neg i_2 \vee o) \wedge$ ... $(\neg i_n \vee o) \wedge$ $(i_1 \vee i_2 \vee ... \vee i_n \vee \neg o)$
$o \leftarrow \text{ITE}(i, t, e)$	$i \wedge t \Rightarrow o$ $i \wedge \neg t \Rightarrow \neg o$ $\neg i \wedge e \Rightarrow o$ $\neg i \wedge \neg e \Rightarrow \neg o$	$(\neg i \vee \neg t \vee o) \wedge$ $(\neg i \vee t \vee \neg o) \wedge$ $(i \vee \neg e \vee o) \wedge$ $(i \vee e \vee \neg o)$
$o \leftarrow \text{NOT}(i)$	$i \Rightarrow \neg o$ $\neg i \Rightarrow o$	$(\neg i \vee \neg o) \wedge$ $(i \vee o)$

Instead of translating the inverters, as shown in the last row of Table I, we can subsume them in their fanout gates, as done in [30] and illustrated in Table II. By replacing all instances of the CNF variable for the inverter output with the negated CNF variable for the inverter input, we eliminate the CNF output variable and the 2 clauses for each inverter.

TABLE II
TRANSLATION TO CNF BY SUBSUMING INVERTERS

Example Logic Gate	Equivalent Constraints	CNF Clauses
$o \leftarrow \text{AND}(\text{NOT}(i_1), i_2, ..., i_n)$	$i_1 \Rightarrow \neg o$ $\neg i_2 \Rightarrow \neg o$ ... $\neg i_n \Rightarrow \neg o$ $\neg i_1 \wedge i_2 \wedge ... \wedge i_n \Rightarrow o$	$(\neg i_1 \vee \neg o) \wedge$ $(i_2 \vee \neg o) \wedge$ ... $(i_n \vee \neg o) \wedge$ $(i_1 \vee \neg i_2 \vee ... \vee \neg i_n \vee o)$
$o \leftarrow \text{OR}(\text{NOT}(i_1), i_2, ..., i_n)$	$\neg i_1 \Rightarrow o$ $i_2 \Rightarrow o$ ... $i_n \Rightarrow o$ $i_1 \wedge \neg i_2 \wedge ... \wedge \neg i_n \Rightarrow \neg o$	$(i_1 \vee o) \wedge$ $(\neg i_2 \vee o) \wedge$ ... $(\neg i_n \vee o) \wedge$ $(\neg i_1 \vee i_2 \vee ... \vee i_n \vee \neg o)$
$o \leftarrow \text{ITE}(\text{NOT}(i), t, \text{NOT}(e))$	$\neg i \wedge t \Rightarrow o$ $\neg i \wedge \neg t \Rightarrow \neg o$ $i \wedge \neg e \Rightarrow o$ $i \wedge e \Rightarrow \neg o$	$(i \vee \neg t \vee o) \wedge$ $(i \vee t \vee \neg o) \wedge$ $(\neg i \vee e \vee o) \wedge$ $(\neg i \vee \neg e \vee \neg o)$

III. 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][41] that satisfy the *forwarding property* of the memory semantics—that a *read* gets the data written by the most recent *write* to an address term equal to the address of the *read*, 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 the implementation details of 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.

Restrictions on the style for describing high-level processors [37][38] reduced the number of terms that appear in both positive and negated equations—and are so called *g-terms* (for general terms)—and increased the number of terms that appear only in positive equations—and are so called *p-terms* (for positive terms). The property of Positive Equality [37][38] allows us to treat syntactically different p-terms as not equal when evaluating the validity of an EUFM formula, thus achieving significant simplifications and orders of magnitude speedup (see [6] for a correctness proof). However, equations between g-term variables can evaluate to either *true* or *false*, and can be encoded with Boolean variables [12][31][44], by accounting for the property of transitivity of equality [7].

In the decision procedure EVC [42], applications of the same UF or UP are eliminated with nested *ITE*s [38]. 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 enforcing functional consistency is used in [22][23][33]. 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][31][36], but does not result in *ITE*-chains, and so will benefit less from the CNF translation described in the next section.

In spite of the tremendous improvements in SAT-solvers, the formal verification of complex pipelined processors does not complete in 24 hours without Positive Equality, but takes a few seconds or a few minutes with the property [43].

IV. Translation to CNF by Merging Gates

A *primary CNF variable* represents the value of a primary input, i.e., corresponds to a Boolean variable in the original Boolean formula. An *auxiliary CNF variable* represents the value of a gate output. A *literal* is an occurrence of a CNF variable or its negation in a clause.

In EVC, the final Boolean formula consists of AND, OR, NOT, and ITE gates. A hashing scheme [38] ensures that: there are no duplicate gates; represents cases of an AND gate driving another AND gate with a single AND gate that has all the inputs of the two gates, except for the output of the driving gate; and similarly represents cases of an OR gate driving another OR gate with a single OR gate. That is, the final Boolean formula has neither AND gates that directly drive other AND gates, nor OR gates that directly drive other OR gates. The hashing scheme also eliminates duplicate inputs to AND and OR gates, and replaces an AND or an OR with a constant if the gate has complemented inputs.

The goal of the following translation to CNF is to reduce the number of CNF variables and clauses, thus to reduce the required BCP that takes up to 90% of the SAT-solving time [29] and generates many non-sequential memory accesses (prone to L2-cache misses) in order to reflect an assignment to a CNF variable on all the clauses that contain the variable. Currently, the L2-cache miss penalty is hundreds of cycles, and is increasing [16]. Furthermore, BCP requires data-dependent branches that are hard to predict, and so frequently incur the branch misprediction penalty—at least 19 cycles, and up to 125 instructions in the Intel Pentium 4 [16]. The choice to implement the following optimizations is based on profiling the benchmarks to identify frequent patterns. Other optimizations can be implemented similarly.

A. Merging ITE-Chains

This translation to CNF is illustrated in row 1 of Table III, and is applied to a chain of n nested ITEs, where the else-expression of each ITE is another ITE with a fanout count of 1, and the innermost ITE has an else-expression e_n that is either not an ITE or has fanout count greater than 1, i.e., is also used elsewhere. Such ITE-chains result after eliminating UPs with the nested-ITE scheme (see Section III), and after eliminating a bit-level *read* from a sequence of *write*s by accounting for the forwarding property of the memory semantics.

An ITE-chain is translated to CNF by means of $2n + 2$ clauses—2 for each of the $n + 1$ chain inputs, t_j ($j = 1, ..., n$) and e_n. For each such input, the first clause expresses the condition that if the input is *true* and is selected by a corresponding assignment to controlling formulas, i_j, of ITE operators that are ahead in the chain, then the chain output, o, should be *true*. The second clause expresses the condition that if the input is *false* and selected, then the output, o, should be *false*.

This CNF translation of ITE-chains can be similarly extended for ITE-trees, where the then-expressions have fanout counts of 1. However, the benchmarks used in this paper contain very few or no ITE-trees.

Negations in an ITE-chain can be reflected by counting the number of negations between the chain output, o, and each of the chain inputs, t_j ($j = 1, ..., n$) and e_n. An odd number of negations between one of these chain inputs and the output results in complementing the output variable, o, in the 2 clauses for that input; an even number of negations leaves the 2 clauses for the input unmodified.

From Table I, the conventional CNF translation of a chain of n nested ITEs will introduce n variables and $4n$ clauses, while the presented translation introduces 1 variable and $2n + 2$ clauses, thus saving $n - 1$ variables and $2n - 2$ clauses.

TABLE III
TRANSLATION TO CNF BY MERGING GATES

#	Group Name	Logic Gates	Equivalent Constraints in the New Translation	CNF Clauses in the New Translation	Statistics from Conventional Translation	Statistics from the New Translation
1	ITE-Chain	$o \leftarrow \text{ITE}(i_1, t_1, e_1)$ $e_1 \leftarrow \text{ITE}(i_2, t_2, e_2)$ $e_2 \leftarrow \text{ITE}(i_3, t_3, e_3)$ . . . $e_{n\text{-}1} \leftarrow \text{ITE}(i_n, t_n, e_n)$ fanout_count(e_1) = 1 fanout_count(e_2) = 1 . . . fanout_count($e_{n\text{-}1}$) = 1 ((fanout_count(e_n) > 1) $\vee$ (gate_type(e_n) $\neq$ ITE))	$i_1 \wedge t_1 \Rightarrow o$ $i_1 \wedge \neg t_1 \Rightarrow \neg o$ $\neg i_1 \wedge i_2 \wedge t_2 \Rightarrow o$ $\neg i_1 \wedge i_2 \wedge \neg t_2 \Rightarrow \neg o$ . . . $\neg i_1 \wedge \neg i_2 \wedge ... \wedge \neg i_{n\text{-}1} \wedge i_n \wedge t_n \Rightarrow o$ $\neg i_1 \wedge \neg i_2 \wedge ... \wedge \neg i_{n\text{-}1} \wedge i_n \wedge \neg t_n \Rightarrow \neg o$ $\neg i_1 \wedge \neg i_2 \wedge ... \wedge \neg i_{n\text{-}1} \wedge \neg i_n \wedge e_n \Rightarrow o$ $\neg i_1 \wedge \neg i_2 \wedge ... \wedge \neg i_{n\text{-}1} \wedge \neg i_n \wedge \neg e_n \Rightarrow \neg o$	$(\neg i_1 \vee \neg t_1 \vee o) \wedge$ $(\neg i_1 \vee t_1 \vee \neg o) \wedge$ $(i_1 \vee \neg i_2 \vee \neg t_2 \vee o) \wedge$ $(i_1 \vee \neg i_2 \vee t_2 \vee \neg o) \wedge$ . . . $(i_1 \vee i_2 \vee ... \vee i_{n\text{-}1} \vee \neg i_n \vee \neg t_n \vee o) \wedge$ $(i_1 \vee i_2 \vee ... \vee i_{n\text{-}1} \vee \neg i_n \vee t_n \vee \neg o) \wedge$ $(i_1 \vee i_2 \vee ... \vee i_{n\text{-}1} \vee i_n \vee \neg e_n \vee o) \wedge$ $(i_1 \vee i_2 \vee ... \vee i_{n\text{-}1} \vee i_n \vee e_n \vee \neg o)$	new CNF variables: n new CNF clauses: $4n$ new CNF literals: $12n$	new CNF variables: 1 new CNF clauses: $2n + 2$ new CNF literals: $n^2 + 7n + 4$
2	AND→ITE	$o \leftarrow \text{ITE}(i, t, e)$ $t \leftarrow \text{AND}(a_1, ..., a_n)$ fanout_count(t) = 1	$i \wedge \neg a_1 \Rightarrow \neg o$ . . . $i \wedge \neg a_n \Rightarrow \neg o$ $i \wedge a_1 \wedge ... \wedge a_n \Rightarrow o$ $\neg i \wedge e \Rightarrow o$ $\neg i \wedge \neg e \Rightarrow \neg o$	$(\neg i \vee a_1 \vee \neg o) \wedge$ . . . $(\neg i \vee a_n \vee \neg o) \wedge$ $(\neg i \vee \neg a_1 \vee ... \vee \neg a_n \vee o) \wedge$ $(i \vee \neg e \vee o) \wedge$ $(i \vee e \vee \neg o)$	new CNF variables: 2 new CNF clauses: $n + 5$ new CNF literals: $3n + 13$	new CNF variables: 1 new CNF clauses: $n + 3$ new CNF literals: $4n + 8$
3	OR→ITE	$o \leftarrow \text{ITE}(i, t, e)$ $t \leftarrow \text{OR}(a_1, ..., a_n)$ fanout_count(t) = 1	$i \wedge a_1 \Rightarrow o$ . . . $i \wedge a_n \Rightarrow o$ $i \wedge \neg a_1 \wedge ... \wedge \neg a_n \Rightarrow \neg o$ $\neg i \wedge e \Rightarrow o$ $\neg i \wedge \neg e \Rightarrow \neg o$	$(\neg i \vee \neg a_1 \vee o) \wedge$ . . . $(\neg i \vee \neg a_n \vee o) \wedge$ $(\neg i \vee a_1 \vee ... \vee a_n \vee \neg o) \wedge$ $(i \vee \neg e \vee o) \wedge$ $(i \vee e \vee \neg o)$	new CNF variables: 2 new CNF clauses: $n + 5$ new CNF literals: $3n + 13$	new CNF variables: 1 new CNF clauses: $n + 3$ new CNF literals: $4n + 8$
4	OR→AND	$o \leftarrow \text{AND}(a_1, a_2, ..., a_n)$ $a_1 \leftarrow \text{OR}(b_1, ..., b_m)$ fanout_count(a_1) = 1	$\neg b_1 \wedge ... \wedge \neg b_m \Rightarrow \neg o$ $\neg a_2 \Rightarrow \neg o$ . . . $\neg a_n \Rightarrow \neg o$ $b_1 \wedge a_2 \wedge ... \wedge a_n \Rightarrow o$ $b_2 \wedge a_2 \wedge ... \wedge a_n \Rightarrow o$ . . . $b_m \wedge a_2 \wedge ... \wedge a_n \Rightarrow o$	$(b_1 \vee ... \vee b_m \vee \neg o) \wedge$ $(a_2 \vee \neg o) \wedge$ . . . $(a_n \vee \neg o) \wedge$ $(\neg b_1 \vee \neg a_2 \vee ... \vee \neg a_n \vee o) \wedge$ $(\neg b_2 \vee \neg a_2 \vee ... \vee \neg a_n \vee o) \wedge$ . . . $(\neg b_m \vee \neg a_2 \vee ... \vee \neg a_n \vee o)$	new CNF variables: 2 new CNF clauses: $n + m + 2$ new CNF literals: $3n + 3m + 2$	new CNF variables: 1 new CNF clauses: $n + m$ new CNF literals: $mn + 2n + 2m - 1$
5	ITE→AND	$o \leftarrow \text{AND}(a_1, a_2, ..., a_n)$ $a_1 \leftarrow \text{ITE}(i, t, e)$ fanout_count(a_1) = 1	$i \wedge \neg t \Rightarrow \neg o$ $\neg i \wedge \neg e \Rightarrow \neg o$ $\neg a_2 \Rightarrow \neg o$ . . . $\neg a_n \Rightarrow \neg o$ $i \wedge t \wedge a_2 \wedge ... \wedge a_n \Rightarrow o$ $\neg i \wedge e \wedge a_2 \wedge ... \wedge a_n \Rightarrow o$	$(\neg i \vee t \vee \neg o) \wedge$ $(i \vee e \vee \neg o) \wedge$ $(a_2 \vee \neg o) \wedge$ . . . $(a_n \vee \neg o) \wedge$ $(\neg i \vee \neg t \vee \neg a_2 \vee ... \vee \neg a_n \vee o) \wedge$ $(i \vee \neg e \vee \neg a_2 \vee ... \vee \neg a_n \vee o)$	new CNF variables: 2 new CNF clauses: $n + 5$ new CNF literals: $3n + 13$	new CNF variables: 1 new CNF clauses: $n + 3$ new CNF literals: $4n + 8$
6	AND→OR	$o \leftarrow \text{OR}(a_1, a_2, ..., a_n)$ $a_1 \leftarrow \text{AND}(b_1, ..., b_m)$ fanout_count(a_1) = 1	$b_1 \wedge ... \wedge b_m \Rightarrow o$ $a_2 \Rightarrow o$ . . . $a_n \Rightarrow o$ $\neg b_1 \wedge \neg a_2 \wedge ... \wedge \neg a_n \Rightarrow \neg o$ $\neg b_2 \wedge \neg a_2 \wedge ... \wedge \neg a_n \Rightarrow \neg o$ . . . $\neg b_m \wedge \neg a_2 \wedge ... \wedge \neg a_n \Rightarrow \neg o$	$(\neg b_1 \vee ... \vee \neg b_m \vee o) \wedge$ $(\neg a_2 \vee o) \wedge$ . . . $(\neg a_n \vee o) \wedge$ $(b_1 \vee a_2 \vee ... \vee a_n \vee \neg o) \wedge$ $(b_2 \vee a_2 \vee ... \vee a_n \vee \neg o) \wedge$ . . . $(b_m \vee a_2 \vee ... \vee a_n \vee \neg o)$	new CNF variables: 2 new CNF clauses: $n + m + 2$ new CNF literals: $3n + 3m + 2$	new CNF variables: 1 new CNF clauses: $n + m$ new CNF literals: $mn + 2n + 2m - 1$
7	ITE→OR	$o \leftarrow \text{OR}(a_1, a_2, ..., a_n)$ $a_1 \leftarrow \text{ITE}(i, t, e)$ fanout_count(a_1) = 1	$i \wedge t \Rightarrow o$ $\neg i \wedge e \Rightarrow o$ $a_2 \Rightarrow o$ . . . $a_n \Rightarrow o$ $i \wedge \neg t \wedge \neg a_2 \wedge ... \wedge \neg a_n \Rightarrow \neg o$ $\neg i \wedge \neg e \wedge \neg a_2 \wedge ... \wedge \neg a_n \Rightarrow \neg o$	$(\neg i \vee \neg t \vee o) \wedge$ $(i \vee \neg e \vee o) \wedge$ $(\neg a_2 \vee o) \wedge$ . . . $(\neg a_n \vee o) \wedge$ $(\neg i \vee t \vee a_2 \vee ... \vee a_n \vee \neg o) \wedge$ $(i \vee e \vee a_2 \vee ... \vee a_n \vee \neg o)$	new CNF variables: 2 new CNF clauses: $n + 5$ new CNF literals: $3n + 13$	new CNF variables: 1 new CNF clauses: $n + 3$ new CNF literals: $4n + 8$

B. Merging AND→ITE and OR→ITE Groups

These cases are illustrated in rows 2 and 3 of Table III. An AND→ITE group, where an n-input AND gate with fanout count of 1 is used as the then-expression of an ITE, is translated to CNF by extending each of the $n + 1$ clauses from conventional translation of the AND gate (see Table I) with the condition that the output of the AND is selected by the ITE-controlling formula, i. The 2 clauses for the else-expression of the ITE are the same as in conventional translation. The translation of the OR→ITE group is similar. This translation can also be extended for ITEs where the else-expression is either an AND or an OR, as well as for ITEs where each of the then- and the else-expressions is either an AND or an OR; all of these cases were implemented and used for the experiments in Section V. A negation of the AND (OR) output in an AND→ITE (OR→ITE) group is reflected by negating the ITE output, o, in all clauses for cases where the AND (OR) output is selected by the ITE.

The conventional CNF translation of AND→ITE and OR→ITE groups introduces 2 variables and $n + 5$ clauses, while the presented translation introduces 1 variable and $n + 3$ clauses, saving 1 variable and 2 clauses per group.

C. Merging OR→AND, ITE→AND, AND→OR, and ITE→OR Groups

These cases are illustrated in rows 4–7 of Table III, and also save 1 variable and 2 clauses per group, relative to conventional translation. In every clause from conventional translation of the lower gate—an OR or an ITE (AND or an ITE) with fanout count of 1—the output variable a_1 is replaced with the output variable of the upper AND (OR) gate, o, in the same polarity, and literals are added if necessary to express the condition that the upper gate input values will allow the output value of the lower gate to propagate to the output of the upper gate.

These four groups are similarly translated to CNF when another input is an OR/ITE (AND/ITE) with fanout count of 1. If multiple inputs are candidates to be merged with their fanout AND (OR) gate, we can choose an input heuristically, e.g., as discussed next.

D. Using the FANIN Heuristic to Choose an Input

The FANIN heuristic [27] orders the BDD variables of a Boolean formula by traversing the formula in depth-first manner. For each gate, the inputs are visited in descending topological order, such that the BDD variables are defined in the order that they are reached. The motivation is that a subformula with more topological levels will likely have a BDD with more levels, so that its BDD variables will affect more levels in the final BDD and have to be declared before the BDD variables of subformulas with fewer topological levels.

The motivation for using the FANIN heuristic to choose an input gate to merge with its fanout AND/OR gate in the configurations in Section IV.C—if there are multiple candidate inputs—is to shorten the longest path for BCP from a primary input to the output of the fanout gate. Thus, if the heuristic is applied to many groups, we could significantly shorten the longest path for BCP from a primary input to the output of the Boolean formula. In the experiments, the topological levels of gates were computed based on the original Boolean formula.

V. Results

The Boolean formulas used in the experiments are from formal verification of safety of: ooo_engine6, an out-of-order processor with a 6-entry reorder buffer, 6 reservation stations, register renaming, and register-register ALU instructions; 1dlx_c_iq40, a single-issue pipelined DLX [16], modeled as described in [38], and extended with a 40-entry instruction queue placed between the instruction fetch stage and the execution pipeline; 1dlx_c_iq50, a variant with a 50-entry instruction queue; 1dlx_c_mc_ex_bp_iq40, an extension of 1dlx_c_iq40 with multicycle functional units, exceptions, and branch prediction, modeled as in [39]; 1dlx_c_mc_ex_bp_iq50, a variant with a 50-entry instruction queue; 9vliw_iq2, a 9-wide VLIW processor that implements predicated execution, register remapping, and advanced loads (see [40]), and has a 2-entry instruction queue; 9vliw_iq6, a variant with a 6-entry instruction queue; 15pipe, a 15-wide, 5-stage pipelined processor with in-order execution, implementing register-register ALU and load instructions [43]; and 10pipe_ooo, a 10-wide, 5-stage pipelined processor with out-of-order execution, also implementing register-register ALU and load instructions [43].

Table IV summarizes the results. The abstraction function [8] for the verification was computed by controlled flushing [9], where the user provides a flushing schedule that eliminates the triggering of stalling conditions, thus simplifying the correctness formula. Equations between g-term variables were encoded with a unique new Boolean variable [12] for the first seven benchmarks, since the method from [12] outperformed the methods from [31][44] on these models, but with a special predicate [44] for the last two benchmarks, since that method was best on those two models. The SAT-solver Siege_v3 [32]—one of the winners in the SAT'03 competition [25]—was used for the experiments. The computer was a Dell OptiPlex GX260 with a 3.06-GHz Intel Pentium 4, having a 512-KB on-chip L2-cache, 2 GB of memory, and running Red Hat Linux 9.

As Table IV shows, the formulas had between 335 and 21,330 Boolean variables before translation to CNF. With the conventional translation, the CNF variables were between 47,440 and 416,535; the CNF clauses between 551,775 and 10,117,902; Siege_v3 made between $0.89{\times}10^6$ and $1{,}454.97{\times}10^6$ decisions, resolved between $0.31{\times}10^6$ and $52.88{\times}10^6$ conflicts, and took between 1,553 and 141,857 seconds to prove the CNF formulas unsatisfiable.

Five other strategies were also explored: Strategy 1, subsuming the inverters (see Table II); Strategy 2, also merging ITE-chains (see Section IV.A); Strategy 3, also merging AND→ITE and OR→ITE groups (see Section IV.B); Strategy 4, also merging OR→AND, ITE→AND, AND→OR and ITE→OR groups (see Section IV.C), by choosing the first input that is an OR (AND) with fanout count of 1 and has fewer inputs than a parameter `INPUT_LIMIT`, such that if no such input is found, then choosing the first input that is an ITE with fanout count of 1; and Strategy 5 that differs from Strategy 4 in choosing the input that is an OR (AND) with fanout count of 1 and has fewer inputs than parameter `INPUT_LIMIT`, or an input that is an ITE with fanout count of 1, such that the chosen input has the highest topological level, as in the FANIN heuristic (see Section IV.D), and if multiple candidate inputs have the same topological level, then ties are broken by choosing an ITE, or otherwise a gate with fewer inputs, or otherwise the first found input. Parameter `INPUT_LIMIT` was experimentally determined to be 4, i.e., the input OR (AND) gate could have either 2 or 3 inputs. With each of these strategies, the time for translation to CNF was almost identical to that with conventional translation.

From Table IV, Strategy 3 had the best performance on 5 of the 9 benchmarks (1dlx_c_iq50, 1dlx_c_mc_ex_bp_iq50, 9vliw_iq2, 9vliw_iq6, and 15pipe), and was within 1% of the best strategy for 2 other benchmarks (ooo_engine6, and 10pipe_ooo). Strategy 5

TABLE IV
RESULTS FROM UNSATISFIABLE FORMULAS

Boolean Formula (Boolean Variables)	Strategy for Translation from Propositional Logic to CNF	CNF				SAT-solver Siege_v3			Speedup[a]
		Variables	Clauses	Literals	Average Literals/Clause	Decisions	Conflicts	Time [sec]	
ooo_engine6 (335)	Strategy 0: conventional translation	47,440	614,098	1,760,646	2.867	0.89×10^6	0.71×10^6	1,553	———
	Strategy 1: subsume inverters	45,414	610,046	1,752,542	2.873	0.65×10^6	0.51×10^6	924	1.68×
	Strategy 2: Strategy 1 + merge ITE-chains	42,277	603,772	1,752,996	2.903	0.68×10^6	0.53×10^6	896	1.73×
	Strategy 3: Strategy 2 + AND/OR→ITE	40,239	599,696	1,762,097	2.938	0.62×10^6	0.49×10^6	810	1.92×
	Strategy 4: Strategy 3 + OR/ITE→AND & AND/ITE→OR	38,713	596,644	1,794,470	3.008	0.61×10^6	0.47×10^6	805	**1.93**×
	Strategy 5: Strategy 3 + FANIN for OR/ITE→AND & AND/ITE→OR	38,713	596,644	1,799,855	3.017	0.61×10^6	0.47×10^6	845	1.84×
1dlx_c_iq40 (2,669)	Strategy 0: conventional translation	237,715	3,177,190	9,167,000	2.885	13.87×10^6	0.31×10^6	2,265	———
	Strategy 1: subsume inverters	230,305	3,162,370	9,137,360	2.889	13.43×10^6	0.29×10^6	1,977	1.15×
	Strategy 2: Strategy 1 + merge ITE-chains	187,267	3,076,294	9,865,892	3.207	9.18×10^6	0.29×10^6	1,098	2.06×
	Strategy 3: Strategy 2 + AND/OR→ITE	180,800	3,063,360	9,901,796	3.232	8.38×10^6	0.29×10^6	1,115	2.03×
	Strategy 4: Strategy 3 + OR/ITE→AND & AND/ITE→OR	177,781	3,057,322	9,978,682	3.264	8.25×10^6	0.29×10^6	1,131	2.00×
	Strategy 5: Strategy 3 + FANIN for OR/ITE→AND & AND/ITE→OR	177,781	3,057,322	9,973,811	3.262	7.39×10^6	0.27×10^6	934	**2.43**×
1dlx_c_iq50 (3,819)	Strategy 0: conventional translation	416,535	5,914,135	17,114,365	2.894	36.05×10^6	0.72×10^6	8,240	———
	Strategy 1: subsume inverters	405,540	5,892,145	17,070,385	2.897	37.22×10^6	0.54×10^6	6,867	1.20×
	Strategy 2: Strategy 1 + merge ITE-chains	321,742	5,724,549	18,909,047	3.303	24.38×10^6	0.68×10^6	4,165	1.98×
	Strategy 3: Strategy 2 + AND/OR→ITE	311,910	5,704,885	18,969,171	3.325	21.74×10^6	0.60×10^6	3,829	**2.15**×
	Strategy 4: Strategy 3 + OR/ITE→AND & AND/ITE→OR	307,396	5,695,857	19,109,532	3.355	22.91×10^6	0.69×10^6	4,299	1.92×
	Strategy 5: Strategy 3 + FANIN for OR/ITE→AND & AND/ITE→OR	307,396	5,695,857	19,102,201	3.354	22.24×10^6	0.69×10^6	4,961	1.66×
1dlx_c_mc_ex_bp_iq40 (6,826)	Strategy 0: conventional translation	267,452	3,850,072	11,145,336	2.895	18.71×10^6	0.76×10^6	5,223	———
	Strategy 1: subsume inverters	258,846	3,832,860	11,110,912	2.899	19.15×10^6	0.82×10^6	5,623	0.93×
	Strategy 2: Strategy 1 + merge ITE-chains	214,311	3,743,790	11,873,260	3.171	16.00×10^6	0.84×10^6	3,637	1.44×
	Strategy 3: Strategy 2 + AND/OR→ITE	199,710	3,714,588	11,943,448	3.215	14.82×10^6	0.92×10^6	3,539	1.48×
	Strategy 4: Strategy 3 + OR/ITE→AND & AND/ITE→OR	196,768	3,708,704	12,028,254	3.243	15.33×10^6	0.90×10^6	3,664	1.43×
	Strategy 5: Strategy 3 + FANIN for OR/ITE→AND & AND/ITE→OR	196,768	3,708,704	12,023,546	3.242	14.21×10^6	0.71×10^6	3,058	**1.71**×
1dlx_c_mc_ex_bp_iq50 (9,996)	Strategy 0: conventional translation	459,482	7,089,897	20,583,601	2.903	54.31×10^6	2.27×10^6	22,174	———
	Strategy 1: subsume inverters	447,031	7,064,995	20,533,797	2.906	48.24×10^6	1.62×10^6	15,885	1.40×
	Strategy 2: Strategy 1 + merge ITE-chains	361,121	6,893,175	22,433,747	3.254	42.59×10^6	1.92×10^6	12,823	1.73×
	Strategy 3: Strategy 2 + AND/OR→ITE	339,145	6,849,223	22,559,963	3.294	38.32×10^6	1.64×10^6	10,668	**2.08**×
	Strategy 4: Strategy 3 + OR/ITE→AND & AND/ITE→OR	334,738	6,840,409	22,712,165	3.320	40.12×10^6	1.90×10^6	11,614	1.91×
	Strategy 5: Strategy 3 + FANIN for OR/ITE→AND & AND/ITE→OR	334,738	6,840,409	22,705,056	3.319	44.12×10^6	2.16×10^6	13,358	1.66×
9vliw_iq2 (6,402)	Strategy 0: conventional translation	48,856	551,775	1,573,303	2.851	27.25×10^6	4.72×10^6	3,195	———
	Strategy 1: subsume inverters	44,095	542,253	1,554,259	2.866	12.96×10^6	1.35×10^6	823	3.88×
	Strategy 2: Strategy 1 + merge ITE-chains	43,968	541,999	1,554,029	2.867	17.17×10^6	2.23×10^6	1,373	2.33×
	Strategy 3: Strategy 2 + AND/OR→ITE	42,755	539,573	1,561,013	2.893	9.68×10^6	0.99×10^6	579	**5.52**×
	Strategy 4: Strategy 3 + OR/ITE→AND & AND/ITE→OR	40,172	534,407	1,560,525	2.920	14.06×10^6	1.50×10^6	916	3.49×
	Strategy 5: Strategy 3 + FANIN for OR/ITE→AND & AND/ITE→OR	40,172	534,407	1,560,482	2.920	11.79×10^6	1.25×10^6	747	4.28×
9vliw_iq6 (21,330)	Strategy 0: conventional translation	223,729	3,662,687	10,603,981	2.895	$1,454.97\times10^6$	52.88×10^6	141,857	———
	Strategy 1: subsume inverters	209,724	3,634,677	10,547,961	2.902	$1,123.27\times10^6$	43.12×10^6	107,923	1.31×
	Strategy 2: Strategy 1 + merge ITE-chains	209,211	3,633,651	10,547,775	2.903	847.19×10^6	37.46×10^6	76,321	1.86×
	Strategy 3: Strategy 2 + AND/OR→ITE	199,340	3,613,909	10,595,485	2.932	286.45×10^6	8.74×10^6	18,480	**7.68**×
	Strategy 4: Strategy 3 + OR/ITE→AND & AND/ITE→OR	194,433	3,604,095	10,616,107	2.946	361.71×10^6	9.88×10^6	21,684	6.54×
	Strategy 5: Strategy 3 + FANIN for OR/ITE→AND & AND/ITE→OR	194,433	3,604,095	10,616,095	2.946	475.23×10^6	12.83×10^6	28,825	4.92×
15pipe (5,775)	Strategy 0: conventional translation	284,965	10,117,902	30,083,190	2.973	75.21×10^6	5.07×10^6	16,149	———
	Strategy 1: subsume inverters	277,976	10,103,924	30,055,234	2.975	75.03×10^6	3.92×10^6	13,127	1.23×
	Strategy 2: Strategy 1 + merge ITE-chains	140,870	9,829,712	41,846,616	4.257	76.32×10^6	2.82×10^6	7,296	2.21×
	Strategy 3: Strategy 2 + AND/OR→ITE	140,610	9,829,192	41,864,998	4.259	59.09×10^6	2.40×10^6	6,018	**2.68**×
	Strategy 4: Strategy 3 + OR/ITE→AND & AND/ITE→OR	139,660	9,827,292	41,958,714	4.270	77.18×10^6	2.91×10^6	7,910	2.04×
	Strategy 5: Strategy 3 + FANIN for OR/ITE→AND & AND/ITE→OR	139,660	9,827,292	41,950,733	4.269	76.12×10^6	2.76×10^6	7,747	2.08×
10pipe_ooo (2,735)	Strategy 0: conventional translation	85,292	2,087,475	6,163,839	2.953	66.09×10^6	20.65×10^6	40,749	———
	Strategy 1: subsume inverters	81,932	2,080,755	6,150,399	2.956	59.43×10^6	18.23×10^6	35,723	1.14×
	Strategy 2: Strategy 1 + merge ITE-chains	51,670	2,020,231	7,194,429	3.561	54.18×10^6	16.96×10^6	26,448	1.54×
	Strategy 3: Strategy 2 + AND/OR→ITE	51,500	2,019,891	7,197,291	3.563	52.57×10^6	16.03×10^6	25,213	1.62×
	Strategy 4: Strategy 3 + OR/ITE→AND & AND/ITE→OR	51,000	2,018,891	7,226,317	3.579	49.80×10^6	15.92×10^6	27,212	1.50×
	Strategy 5: Strategy 3 + FANIN for OR/ITE→AND & AND/ITE→OR	51,000	2,018,891	7,220,565	3.577	49.43×10^6	16.20×10^6	24,980	**1.63**×

a. Speedup is the SAT time with conventional translation, divided by the new SAT time

was best on 3 of the 9 benchmarks (1dlx_c_iq40, 1dlx_c_mc_ex_bp_iq40, and 10pipe_ooo). However, Strategy 3 was better than Strategy 5 on 1dlx_c_iq50 and 1dlx_c_mc_ex_bp_iq50, the more complex variants of 1dlx_c_iq40 and 1dlx_c_mc_ex_bp_iq40, respectively, and significantly better on the most complex formula, 9vliw_iq6, reducing the SAT-solver decisions for that benchmark from $1{,}454.97\times10^6$ (with conventional translation) to 286.45×10^6 (i.e., 5.08×), reducing the SAT-solver conflicts from 52.88×10^6 to 8.74×10^6 (i.e., 6.05×), and speeding up the SAT-checking by 7.68×.

As Table IV indicates, the number of CNF variables, clauses, and literals is not proportional to the SAT time. For example, 15pipe has more CNF variables, and approximately 3× more CNF clauses and literals than 9vliw_iq6 with conventional translation, but was proved unsatisfiable 8.78× faster. The greatest reduction in CNF variables was for 15pipe, where Strategies 2–5 more than halved the CNF variables. The reduction in CNF clauses was significantly smaller for all of the benchmarks—between 1.6% and 3.85%. Note that Strategies 4 and 5 result in the same number of CNF variables and clauses, but depending on the benchmark, require different number of CNF literals, and produce different clauses.

Strategies 3 and 5 also had best performance on satisfiable CNFs from 10 buggy implementations of 9vliw_iq6. Strategy 3 was faster than Strategy 0 in 6 cases, with maximum speedup of 136×; Strategy 5 was faster than Strategy 0 in 7 cases, with maximum speedup of 17.8×. However, Strategy 3 was faster than Strategy 5 on only 5 (half) of the cases, i.e., those strategies had comparable performance. If Strategies 3 and 5 were run in parallel, stopping as soon as one of them finds a solution, they were faster than Strategy 0 in 8 cases.

ITEs model conditional advances of instructions in the pipelines, when stalling conditions are false. Additionally, EVC uses ITE-chains to eliminate UPs that control the flow of instructions. Hence, faster BCP for ITE-chains, as well as for AND→ITE and OR→ITE groups, as achieved with Strategy 3, means faster processing of case-splitting conditions, and results in significant speedup. Merging OR→AND and AND→OR groups, as in Strategies 4 and 5, usually increases the CNF literals, thus slowing the BCP for some formulas.

Strategies 1–5 resulted in similar speedups, for both satisfiable and unsatisfiable formulas, with another efficient SAT-solver, BerkMin621 [14], a new version of BerkMin62 [13].

VI. Related Work

To reduce the cost of Boolean Constraint Propagation, Bingham and Hu [4] compiled Boolean formulas to programs, and simulated the resulting code with random unexplored vectors. Additional code was generated to identify input patterns that will produce the same output value as the current input pattern, thus pruning the solution space. Kuehlmann et al. [20] converted Boolean formulas to a representation with only 2-input AND gates and inverters, then transformed groups of 3 connected AND gates—one driven by the other two—into a canonical form by accounting for inverters at gate inputs. Circuit-based SAT-solvers exploit the circuit structure to identify points with identical or complemented values, as well as to prune the solution space [11][15][20][26]. Algebraic simplifications [5][28], as well as methods for CNF variable ordering by minimizing the cut-width [2][10][45], required long processing time, and did not accelerate the SAT checking [43]. Using multiple parallel runs of a SAT-solver with either different decision heuristics [34], or with different translations from EUFM to propositional logic [43], and stopping as soon as one of the runs finds an answer, reduced the SAT time. Translation of Boolean formulas to tableau is explored in [18].

VII. Conclusions

The paper studied translation of Boolean formulas to CNF by merging adjacent gates, and representing them with a single set of CNF clauses, thus eliminating CNF variables and clauses, and speeding up the BCP and the SAT-solving. Best was Strategy 3—subsuming inverters, merging ITE-chains, and merging AND$\rightarrow$ITE and OR$\rightarrow$ITE groups—resulting in 7.68$\times$ speedup for the most complex unsatisfiable formula, and 136$\times$ speedup for one of the satisfiable formulas from buggy implementations. Second was Strategy 5, additionally merging OR$\rightarrow$AND, ITE$\rightarrow$AND, AND$\rightarrow$OR, and ITE$\rightarrow$OR groups, where the input gate to be merged was chosen by a variant of the FANIN heuristic for BDD variable ordering. ITEs play a critical role in controlling the instruction flow, thus merging ITEs with adjacent gates results in significant speedup. Future work will fine-tune the presented heuristics.

References

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

[2] F.A. Aloul, I.L. Markov, and K.A. Sakallah, "Faster SAT and smaller BDDs via common function structure," *International Conference on Computer-Aided Design (ICCAD '01)*, November 2001, pp. 443–448.

[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, Springer-Verlag, July 2002, pp. 236–249.

[4] J.D. Bingham, and A.J. Hu, "Semi-formal bounded model checking," *Computer-Aided Verification (CAV '02)*, LNCS 2404, Springer-Verlag, July 2002, pp. 280–294.

[5] R.I. Brafman, "A simplifier for propositional formulas with many binary clauses," *International Joint Conference on Artificial Intelligence (IJCAI '01)*, 2001, pp. 515–520.

[6] R.E. Bryant, S. German, and M.N. Velev, "Processor verification using efficient reductions of the logic of uninterpreted functions to propositional logic," ACM Transactions on Computational Logic (TOCL), Vol. 2, No. 1 (January 2001), pp. 93–134.

[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)*, D.L. Dill, *ed.*, LNCS 818, Springer-Verlag, June 1994, pp. 68–80.

[9] J.R. Burch, "Techniques for verifying superscalar microprocessors," *33rd Design Automation Conference (DAC '96)*, June 1996, pp. 552–557.

[10] E.M. Clarke, and O. Strichman, "A failed attempt to optimize variable ordering with tools for constraints solving," *Workshop on Constraints in Formal Verification (CFV '02)*, September 2002.

[11] M.K. Ganai, L. Zhang, P. Ashar, A. Gupta, S. Malik, "Combining strengths of circuit-based and CNF-based algorithms for a high-performance SAT solver," *39th Design Automation Conference (DAC '02)*, June 2002.

[12] 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)*, LNCS 1427, Springer-Verlag, June 1998.

[13] E. Goldberg, and Y. Novikov, "BerkMin: a fast and robust SAT-solver," *Design, Automation, and Test in Europe (DATE '02)*, March 2002.

[14] E. Goldberg, and Y. Novikov, Personal communication, June 2003.

[15] A. Gupta, A. Gupta, Z. Yang, and P. Ashar, "Dynamic detection and removal of inactive clauses in SAT with application in image computation," *38th Design Automation Conference (DAC '01)*, June 2001.

[16] J.L. Hennessy, and D.A. Patterson, *Computer Architecture: A Quantitative Approach*, 3rd edition, Morgan Kaufmann, San Francisco, 2002.

[17] D.S. Johnson, and M.A. Trick, *eds.*, *The Second DIMACS Implementation Challenge*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science. http://dimacs.rutgers.edu/challenges

[18] T.A. Junttila, and I. Niemelä, "Towards an efficient tableau method for boolean circuit satisfiability checking," *1st International Conference on Computational Logic (CL '00)*, LNAI 1861, Springer-Verlag, July 2000.

[19] K. Keutzer, "DAGON: technology binding and local optimization by DAG Matching," *24th Design Automation Conference (DAC '87)*, June 1987.

[20] A. Kuehlmann, M.K. Ganai, and V. Paruthi, "Circuit-based boolean reasoning," *38th Design Automation Conference (DAC '01)*, June 2001.

[21] S. Lahiri, C. Pixley, and K. Albin, "Experience with term level modeling and verification of the M•CORE™ microprocessor core," *International Workshop on High Level Design, Validation and Test (HLDVT '01)*, November 2001.

[22] 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, Springer-Verlag, November 2002.

[23] 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.

[24] T. Larrabee, "Test pattern generation using boolean satisfiability," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD), Vol. 11, No. 1 (January 1992), pp. 4–15.

[25] D. Le Berre, and L. Simon, "Results from the SAT'03 solver competition," *6th International Conference on Theory and Applications of Satisfiability Testing (SAT '03)*, 2003. http://www.lri.fr/~simon/contest03/results/

[26] F. Lu, L.-C. Wang, K.-T. Cheng, J. Moondanos, and Z. Hanna, "A signal correlation guided ATPG solver and its applications for Solving Difficult Industrial Cases," *40th Design Automation Conference (DAC '03)*, June 2003.

[27] 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 (ICCAD '88)*, November 1988, pp. 6–9.

[28] J.P. Marques-Silva, "Algebraic simplification techniques for propositional satisfiability," *Principles and Practice of Constraint Programming (CP '00)*, Rina Dechter, *ed.*, LNCS 1894, Springer-Verlag, September 2000.

[29] 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, pp. 530–535.

[30] D.A. Plaisted, and S. Greenbaum, "A structure preserving clause form translation," Journal of Symbolic Computation (JSC), Vol. 2, 1985, pp. 293–304.

[31] 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), pp. 279–293.

[32] L. Ryan, Siege SAT Solver v.3. http://www.cs.sfu.ca/~loryan/personal/

[33] 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.

[34] O. Shacham, and E. Zarpas, "Tuning the VSIDS decision heuristic for bounded model checking," *Microprocessor Test and Verification (MTV '03)*, May 2003.

[35] 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, pp. 466–483.

[36] 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. http://www.win.tue.nl/~hzantema/TZ.pdf

[37] 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, pp. 397–401.

[38] 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, pp. 37–53.

[39] 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.

[40] 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.

[41] 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.

[42] 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)*, LNCS 2102, Springer-Verlag, July 2001, pp. 235–240.

[43] 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.

[44] M.N. Velev, "Automatic abstraction of equations in a logic of equality," *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.

[45] D. Wang, E. Clarke, Y. Zhu, and J. Kukula, "Using cutwidth to improve symbolic simulation and boolean satisfiability," *IEEE International High Level Design Validation and Test Workshop (HLDVT '01)*, 2001.

[46] L. Zhang, and S. Malik, "Cache performance of SAT solvers: a case study for efficient implementation of algorithms," *6th International Conference on Theory and Applications of Satisfiability Testing (SAT '03)*, May 2003.