

Using Automatic Case Splits and Efficient CNF Translation to Guide a SAT-Solver When Formally Verifying Out-of-Order Processors

Miroslav N. Velev

mvelev@ece.cmu.edu

<http://www.ece.cmu.edu/~mvelev>

Abstract. The paper integrates automatically generated case-splitting expressions, and an efficient translation to CNF, in order to formally verify an out-of-order superscalar processor having register renaming, as well as Reorder Buffer and Reservation Stations that are completely implemented and instantiated. The processor was defined in the high-level Hardware Description Language AbsHDL, based on the logic of Equality with Uninterpreted Functions and Memories (EUFM), and was formally verified with an extended version of the decision procedure EVC, combined with a SAT-solver. The manual work was limited to definition of necessary invariant constraints. The formal verification was decomposed, based on automatically generated case-splitting expressions—matching Recorder Buffer entries with Reservation Stations to compute the results for those Reorder Buffer entries, and allowing for orders of magnitude speedup if many CPUs are available for parallel runs of EVC. Efficient translation from EUFM to CNF—by producing more ITEs, and merging ITE-trees with 2 levels of their leaves—resulted in additional 32× speedup.

1. Introduction

This paper presents a method for automatic formal verification of out-of-order superscalar processors with register renaming [38][81], and studies the potential for additional speedup if extensive computing resources are available. Currently companies use compute-farms with thousands of CPUs to extensively test new microprocessors by binary simulation [36], without guarantee for complete correctness. In our earlier work on applying the logic of Equality with Uninterpreted Functions and Memories (EUFM) [20]—see Sect. 2.1—to formal verification of pipelined and superscalar processors with in-order execution, we imposed some simple restrictions [92][93] on the high-level description style, resulting in correctness formulas where most of the word-level values appear only in positive equality comparisons. That allowed us to exploit a property that such word-level values can be treated as distinct constants, thus significantly pruning the solution space, and achieving orders of magnitude speedup, while still performing complete formal verification; we call this property *Positive Equality*. Those restrictions, together with techniques to model multicycle functional units, exceptions, and branch prediction [94], allowed our tool flow [96] to be used at Motorola [53] to formally verify a model of the M•CORE processor, and detect 3 bugs, as well as corner cases that were not fully implemented. However, the automatic formal verification of out-of-order processors with register renaming, as well as Reorder Buffer and Reservation Stations that are completely implemented and instantiated, remains a challenge.

Previous methods for formal verification of out-of-order processors require extensive manual work to abstract all hardware structures; to reduce the complexity of the proof—by setting up an inductive argument over the number of Reorder Buffer entries [5][40][47][48][55], or by applying symmetry reductions to decrease the number of Reorder Buffer entries, Reservation Stations, and registers considered in the proof [44][66][67], or by deriving rewriting rules that are based on the structure of the correctness formulas and are used to simplify those formulas [40][97]; to define an *abstraction function*—mapping an implementation state to an equivalent specification state—by extending the Reorder Buffer with auxiliary state used to simplify the expressions produced by the abstraction function [4][40][44][54][97], or by defining an intermediate abstraction to bridge the gap between the implementation and the specification [11][40][41][47][48][77][78]; and to define and prove many lemmas and theorems [40][77][78].

In this paper, the formal verification is done with an extended version of our tool flow [96] that was applied at Motorola, and consists of: 1) the term-level symbolic simulator TLSim, used to symbolically simulate the high-level implementation and specification processors, and produce an EUFM correctness formula; 2) an improved version of the decision procedure EVC that exploits Positive Equality and other optimizations to translate the EUFM correctness formula to a satisfiability-equivalent Boolean formula; and 3) an efficient SAT-solver. The tool flow was also used in an advanced computer architecture course [99], where students designed and formally verified single-issue pipelined processors, as well as extensions with exceptions and branch prediction, and dual-issue superscalar implementations.

The contributions of this paper include: 1) the definition of invariant constraints that are necessary for formal verification of out-of-order superscalar processors with completely implemented Reorder Buffer and Reservation Stations; 2) the use of automatically generated case splits, based on the invariant constraints, to decompose the proof into many simpler proofs that can be discharged efficiently by exploiting Positive Equality, and result in orders of magni-

tude speedup, compared to monolithic formal verification; and 3) a new translation from propositional logic to Conjunctive Normal Form (CNF) [45], uniquely targeted to the structure of the Boolean correctness formulas from such processors, and resulting in additional 32× speedup. These contributions make possible a formal verification method that contrasts previous approaches, where the Reorder Buffer and Reservation Stations are manually abstracted, and the user has to set up an inductive proof over the number of Reorder Buffer entries, and possibly manually apply symmetry reductions to decrease the number of Reservation Stations. Furthermore, and also in contrast to previous approaches, the Reorder Buffer is not extended with auxiliary state in order to simplify the expressions resulting from the abstraction function; significant additional speedup can be expected with such approaches, at the cost of extra manual work.

2. Background

2.1 Translation from EUFM to Propositional Logic

The syntax of EUFM [20] 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* [20][95] 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.

To check an EUFM formula for validity, we can use a specialized decision procedure [30][84], or can translate the EUFM formula to an equivalent Boolean formula that has to be a tautology in order for the EUFM formula to be valid. The second approach allows us to benefit from the recent tremendous advances in SAT-solvers—e.g., [34][68][74] (see [59][98] for comparative studies, and [12][50][107] for surveys)—and is used in the current paper. Restrictions on the style for describing high-level processors [92][93] 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 [92][93] 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 [18] for correctness proof). However, equations between g-term variables can be either *true* or *false*, and can be encoded with Boolean variables [33][72][100], by accounting for the transitivity property of equality [19].

When translating an EUFM formula to an equivalent Boolean formula, applications of the same UF or UP can be eliminated with nested ITEs [93]. For example, if $f(a_1, b_1)$, $f(a_2, b_2)$, and $f(a_3, b_3)$ are three applications of UF f , 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 term 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 term 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 term variable. That is, the second, third, and any subsequent applications of the UF are eliminated with *ITE-chains* that enforce functional consistency. UPs are eliminated in the same way, but using new Boolean variables instead of new term variables. This method for eliminating UFs and UPs by enforcing functional consistency is used in [54][55][79][96]. Alternatively, functional consistency can be enforced with Ackermann constraints [1]—the three applications of the UF will be replaced with the new term variables c_1, c_2 , and c_3 ; then, the functional consistency of the second application of the UF 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 UF. This method for enforcing functional consistency is used in [8][46][60][72][88][106], but does not result in ITE-chains, and so will not benefit from the CNF translation in Sect. 2.3. A general case of ITE-chains are *ITE-trees*, where the then-expressions can be ITEs as well. We will call *leaves of an ITE-tree* its inputs that appear as then- or else-inputs of the lowest level of ITEs in the tree. ITE-trees also result after eliminating a *read* from a sequence of *writes* by accounting for the forwarding property of the memory semantics, and from modeling conditional instruction flow when instructions are not stalled by the control logic.

After the UFs are eliminated, the terms consist of only ITE operators and term variables. In earlier EUFM decision procedures that exploit Positive Equality [54][55][79][96], equations between nested-ITE terms are eliminated by pushing the equations to the ITE leaves, and replacing the original equation with a disjunction of conjunctions 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 $c_1 \wedge c_2 \wedge (a_1 = b_1) \vee c_1 \wedge \neg c_2 \wedge (a_1 = b_2) \vee \neg c_1 \wedge c_2 \wedge (a_2 = b_1) \vee \neg c_1 \wedge \neg c_2 \wedge (a_2 = b_2)$. However, as observed in [102], we can preserve the ITE-tree structure of equation arguments, and replace the equation with $ITE(c_1, ITE(c_2, a_1 = b_1, a_1 = b_2), ITE(c_2, a_2 = b_1, a_2 = b_2))$. Furthermore, we can translate the resulting ITE-tree to CNF by merging the constraints for ITEs inside the tree, and representing it with a single set of clauses without intermediate variables for outputs of ITEs inside the tree (see Sect. 2.3). This resulted in up to 420× speedup for processors with in-order execution [102].

In the extended version of the decision procedure EVC [96] that is used in this paper, the final Boolean formula consists of AND, OR, NOT, and ITE gates. Hashing [93] ensures that: there are no duplicate gates; merges an AND having another AND as input into a single AND having as inputs all the inputs of the two gates, except for the output of the merged AND, and similarly for an OR having another OR as input; eliminates duplicate inputs to AND and OR gates; and replaces an AND/OR with a constant if the gate has complemented inputs.

2.2 Conventional Translation from Propositional Logic to CNF

A *primary CNF variable* is one representing the value of a primary input, i.e., input of the original Boolean circuit. (Boolean formula and Boolean circuit are used interchangeably in this paper.) An *auxiliary 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 [87], we get a satisfiability-equivalent [22] CNF representation. Both the size of the resulting CNF and the complexity of the translation procedure are linear in the size of the original Boolean formula. For AND, OR, NOT, and ITE gates, the conventional translation to CNF is as follows (“ \leftarrow ” stands for assignment):

$$\begin{aligned} o &\leftarrow \text{AND}(i_1, i_2, \dots, i_n) : \\ &\quad (i_1 \vee \neg o) \wedge (i_2 \vee \neg o) \wedge \dots \wedge (i_n \vee \neg o) \wedge (\neg i_1 \vee \neg i_2 \vee \dots \vee \neg i_n \vee o) \\ o &\leftarrow \text{OR}(i_1, i_2, \dots, i_n) : \\ &\quad (\neg i_1 \vee o) \wedge (\neg i_2 \vee o) \wedge \dots \wedge (\neg i_n \vee o) \wedge (i_1 \vee i_2 \vee \dots \vee i_n \vee \neg o) \\ o &\leftarrow \text{NOT}(i) : \\ &\quad (\neg i \vee \neg o) \wedge (i \vee o) \\ o &\leftarrow \text{ITE}(i, t, e) : \\ &\quad (\neg i \vee \neg t \vee o) \wedge (\neg i \vee t \vee \neg o) \wedge (i \vee \neg e \vee o) \wedge (i \vee e \vee \neg o) \end{aligned}$$

Instead of explicitly translating the inverters (NOT gates), we can subsume them in their fanout gates [71], 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. The *controlling value* of an AND gate is *false* (0), since it produces a *false* (0) on the gate output, regardless of the values on other inputs. Similarly, the controlling value of an OR gate is *true* (1).

2.3 Translation from Propositional Logic to CNF by Merging ITE-Trees

Preserving the ITE-tree structure of equation arguments (see Sect 2.1) results in Boolean correctness formulas with ITE-trees, where each ITE inside a tree has *fanout count* of 1, i.e., drives only one gate, and that is another ITE inside the same tree. An ITE-tree can be translated to CNF with a unified set of clauses [102], without intermediate variables for outputs of ITEs inside the tree—see Fig. 1.a. Furthermore, ITE-trees can be merged with 1 level of their AND/OR leaves [102], where each leaf has fanout count of 1—see Fig. 1.b.

We can similarly merge other gate groups when translating them to CNF [101]: AND/OR \rightarrow ITE groups, where an ITE has an AND or OR gate as its then-input, or else-input, or a different AND/OR gate at each of these inputs; and OR/ITE \rightarrow AND (AND/ITE \rightarrow OR) groups, where an AND (OR) is driven by an OR (AND), or an ITE. Note that a driven AND (OR) gate may have many OR/ITE (AND/ITE) inputs with fanout count of 1. Then, we can choose which one to merge by using a variant of the FANIN heuristic [64] for BDD-variable ordering—selecting the input gate with highest topological level. The motivation is to shorten the longest path for *Boolean Constraint Propagation*

(BCP) from a primary input to the output of the driven AND (OR) gate. Then, if the heuristic is applied to many gate groups, we could significantly shorten many paths for BCP from primary inputs to the output of the Boolean circuit.

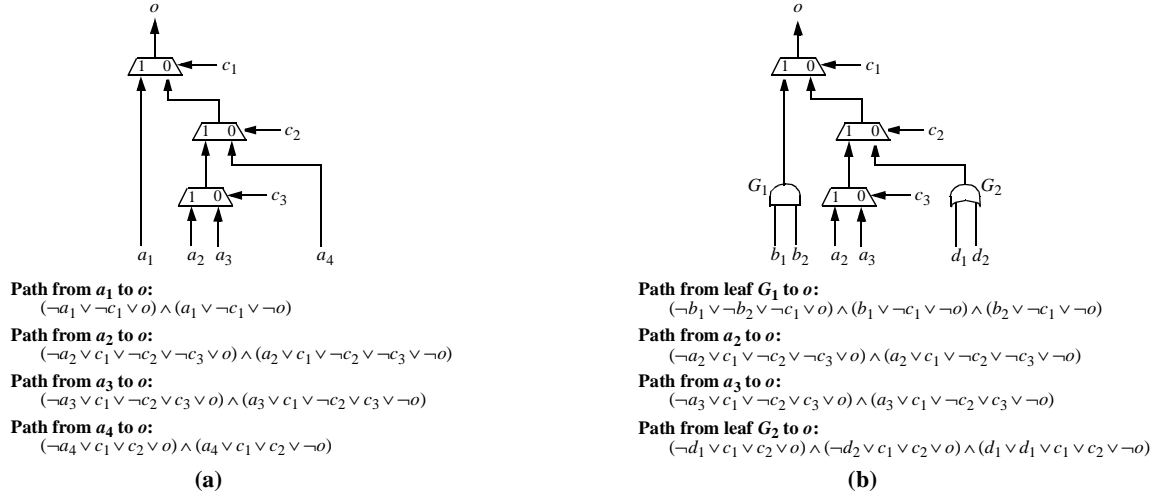


Fig. 1. (a) Example ITE-tree, and its translation to CNF with a unified set of clauses without intermediate variables for outputs of ITEs inside the tree; and (b) Merging an ITE-tree with 1 level of its AND/OR leaves that have fanout count of 1. Each ITE-tree is represented with the conjunction of all clauses for paths from the tree leaves to the tree output. ITEs are shown as multiplexers.

3. Implementation Processor to be Formally Verified

The out-of-order implementation processor to be formally verified is shown in Fig. 2. It is modeled after the PowerPC 750 [42], but can execute only register-register-ALU instructions. A Fetch Engine supplies up to two instructions to the Register Rename & Dispatch unit on every clock cycle. Each instruction has five fields: a *RegWrite* bit, indicating whether the instruction will update the Register File; an *Opcode*; a destination register *Dest*; and two source registers, *Src1* and *Src2*. The Register Rename & Dispatch unit issues in program order up to 2 of the instructions supplied by the Fetch Engine, as long as there are available Reorder Buffer (ROB) entries and Reservation Stations. The Reorder Buffer is a FIFO structure, implemented as a shift register (like in the PowerPC 750) that keeps the original program order of instructions currently in execution, and temporarily stores their results until the instructions are completed in program order. The Register Rename & Dispatch unit renames each destination register to a destination tag, *DestTag*, that is different from those of instructions currently in execution, and is used to identify the result of the instruction. The Register Rename & Dispatch unit also provides the data for each operand of an instruction that is being issued, as long as that operand is already computed and is located either in the Register File or the ROB, or else the destination tag of the most recently issued instruction that is in the ROB and will compute the operand. A Reservation Station is a buffer that temporarily stores instruction information, including the opcode, the two source tags, and operands, until both operands become available and the instruction can be executed by a functional unit. After an instruction is executed, its result is placed on a common data bus, and is forwarded to instructions in other Reservation Stations, to instructions that are being issued by the Register Rename & Dispatch unit, and to the ROB entry waiting for the result. The PowerPC 750 has 6 Reservation Stations, and 6 ROB entries. The Fetch Engine gets instructions from a read-only Instruction Memory.

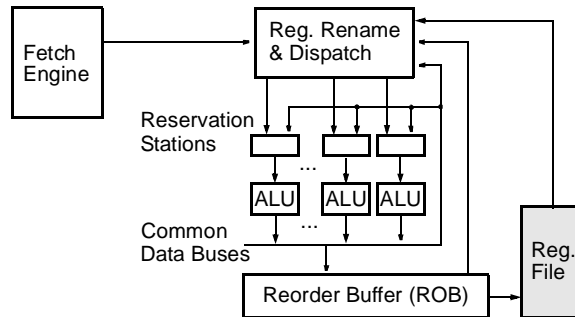


Fig. 2. Block diagram of the implementation processor.

Every entry in the ROB has 6 fields: *Valid*, indicating whether the entry contains a valid instruction; *RegWrite*, controlling whether the instruction will write its result to the Register File; *ValidResult* indicating whether the instruction's result has been computed, in which case it is stored in field *Result* of the same ROB entry; *Dest* and *DestTag*, the destination register and its renaming tag, respectively. Instructions are executed out of program order, as soon as both operands of an instruction become available in the Reservation Station allocated for the instruction. Each Reservation Station has 8 fields: *Valid*, indicating whether the Reservation Station contains a valid instruction; *ValidData1* and *ValidData2*, indicating whether the first and second data operands, respectively, are available in fields *Data1* and *Data2*; the source tags for the data operands, *SrcTag1* and *SrcTag2*, respectively; and the destination tag, *DestTag*, that will be placed on a common data bus together with the result, and will be used to forward the result to the ROB entry with the same destination tag, and to waiting Reservation Stations.

Up to 2 of the 2 oldest instructions are retired in program order on every clock cycle, i.e., are removed from the beginning of the ROB and their results are written to the Register File, as long as those ROB entries are valid, their *RegWrite* bits are *true*, and their results have been computed (bits *ValidResult* are *true*).

The user-visible state consists of the PC and Register File. The specification processor is non-pipelined and executes one instruction per clock cycle by fetching the instruction from the read-only Instruction Memory, incrementing the PC, computing the ALU result, and writing it to the instruction's destination register in the Register File if the instruction's *RegWrite* bit is *true*.

The ROB, the Reservation Stations, the Dispatch logic, the logic for renaming source registers of dispatched instructions, and the Retirement logic are completely implemented. The block that renames destination registers of dispatched instructions is abstracted with a generator of arbitrary values [94], producing new term variables. Constraints are imposed that each such term variable, abstracting a destination tag of a dispatched instruction, be different from term variables abstracting other destination tags in the execution engine, as discussed next.

4. Constraints for the Formal Verification

4.1 Necessary Constraints for the Abstracted Block That Renames Destination Registers

Let FE_1 and FE_2 be the first and second instruction supplied by the Fetch Engine, and let $FE_1.DestTag$ and $FE_2.DestTag$ be the renaming tags for the destination registers of these instructions. Let ROB_i be the i^{th} entry in the ROB, and let $ROB_i.Valid$ and $ROB_i.DestTag$ be its *Valid* bit and its destination tag that renames the instruction's destination register. Then, since the block that renames destination registers is abstracted with a generator of arbitrary values, we need to impose the constraints that the destination tag of a newly dispatched instruction be different from the destination tags of valid instructions in the ROB:

$$ROB_i.Valid \Rightarrow \neg(ROB_i.DestTag = FE_1.DestTag), \quad i = 1, \dots, N \quad (1)$$

$$ROB_i.Valid \Rightarrow \neg(ROB_i.DestTag = FE_2.DestTag), \quad i = 1, \dots, N \quad (2)$$

where N is the number of ROB entries. In the case of processors that can dispatch more instructions per clock cycle, such N constraints have to be imposed for each dispatched instruction.

Similarly, the verification required the constraint that the destination tags of both dispatched instructions be different:

$$\neg(FE_1.DestTag = FE_2.DestTag) \quad (3)$$

Note that constraints (1), (2), and (3) do not require an invariant check, i.e., are assumed to be properties of the abstracted block for renaming destination registers. An actual implementation of that block has to be formally verified to satisfy these properties.

4.2 Necessary Invariant Constraints

The ROB was implemented as a shift register (like in the PowerPC 750 [42]), such that instructions are removed only from the beginning of the shift register, the remaining valid entries are shifted to fill the emptied slots, and new instructions are placed from the resulting first free position. Hence, if an entry is free (i.e., not valid), then so will be all subsequent entries, and the required invariant constraints for the ROB were:

$$\neg ROB_i.Valid \Rightarrow \bigwedge_{j=i+1}^N \neg ROB_j.Valid, \quad i = 1, \dots, N-1 \quad (4)$$

Note that constraints (4) imply that if an entry in the ROB is valid, then so are all preceding entries:

$$ROB_i.Valid \Rightarrow \bigwedge_{j=1}^{i-1} ROB_j.Valid, \quad i = 2, \dots, N \quad (4')$$

although such invariant constraints were not used.

Since each destination register is renamed to a unique destination tag at dispatch, then every pair of valid entries in the ROB should have different destination tags:

$$\text{ROB}_i.\text{Valid} \wedge \text{ROB}_j.\text{Valid} \Rightarrow \neg(\text{ROB}_i.\text{DestTag} = \text{ROB}_j.\text{DestTag}), \quad i < j, \quad i, j = 1, \dots, N \quad (5')$$

Based on (4'), if $\text{ROB}_j.\text{Valid}$ is *true* then $\text{ROB}_i.\text{Valid}$ is *true*, for $i < j$, and we can rewrite (5') as:

$$\text{ROB}_j.\text{Valid} \Rightarrow \neg(\text{ROB}_i.\text{DestTag} = \text{ROB}_j.\text{DestTag}), \quad i < j, \quad i, j = 1, \dots, N \quad (5)$$

which are the invariant constraints that were used.

Similarly, since the destination tags are also kept in the Reservation Stations to identify the result when placed on a common data bus, then every pair of Reservation Stations RS_i and RS_j that contain valid information should have different destination tags:

$$\text{RS}_i.\text{Valid} \wedge \text{RS}_j.\text{Valid} \Rightarrow \neg(\text{RS}_i.\text{DestTag} = \text{RS}_j.\text{DestTag}), \quad i < j, \quad i, j = 1, \dots, K \quad (6)$$

where K is the number of Reservation Stations.

Since instructions that write to the Register File, i.e., whose *RegWrite* bit is *true*, are allocated both an ROB entry and a Reservation Station at dispatch, then every valid instruction in the ROB with *RegWrite* bit of *true* and *ValidResult* bit of *false* (i.e., the result is not yet available in the *Result* field of that ROB entry) should have a pending update from a Reservation Station. That Reservation Station should contain valid information and destination tag equal to the destination tag of that ROB entry. Additionally, if the first ROB entry (containing the oldest instruction in execution) does not have its result computed, then the Reservation Station that will produce the result should have both of its data operands ready, as there are no older instructions that could produce those operands; this extra constraint is needed to avoid deadlock. Hence:

$$\begin{aligned} & \text{ROB}_1.\text{Valid} \wedge \text{ROB}_1.\text{RegWrite} \wedge \neg\text{ROB}_1.\text{ValidResult} \\ & \Rightarrow \bigvee_{j=1}^K [\text{RS}_j.\text{Valid} \wedge (\text{RS}_j.\text{DestTag} = \text{ROB}_1.\text{DestTag}) \wedge \text{RS}_j.\text{ValidData1} \wedge \text{RS}_j.\text{ValidData2}] \end{aligned} \quad (7')$$

$$\begin{aligned} & \text{ROB}_i.\text{Valid} \wedge \text{ROB}_i.\text{RegWrite} \wedge \neg\text{ROB}_i.\text{ValidResult} \\ & \Rightarrow \bigvee_{j=1}^K \text{RS}_j.\text{Valid} \wedge (\text{RS}_j.\text{DestTag} = \text{ROB}_i.\text{DestTag}), \quad i = 2, \dots, N \end{aligned} \quad (7)$$

The next invariant constraints avoid circular data dependencies between Reservation Stations, e.g., Reservation Station k waiting for an operand to be produced by Reservation Station l , which is waiting for an operand to be produced by Reservation Station k —a scenario leading to deadlock, as also noted by McMillan [66]. Hence, the invariant constraints that if a Reservation Station is waiting for an operand, it will be produced by a Reservation Station corresponding to an earlier ROB entry:

$$\begin{aligned} & \text{RS}_i.\text{Valid} \wedge \neg\text{RS}_i.\text{ValidData1} \\ & \Rightarrow \bigvee_{j=2}^N [\text{ROB}_j.\text{Valid} \wedge \text{ROB}_j.\text{RegWrite} \wedge \neg\text{ROB}_j.\text{ValidResult} \\ & \quad \wedge (\text{RS}_i.\text{DestTag} = \text{ROB}_j.\text{DestTag}) \wedge \text{RS}_i.\text{data1_from_before_ROB}_j], \quad i = 1, \dots, K \end{aligned} \quad (8)$$

where $\text{RS}_i.\text{data1_from_before_ROB}_j$ is the condition that ROB_j is preceded by an ROB entry that has an earlier index and is waiting for the same result as the first data operand of Reservation Station i , i.e., for result with destination tag equal to $\text{RS}_i.\text{SrcTag1}$:

$$\begin{aligned} \text{RS}_i.\text{data1_from_before_ROB}_j \leftarrow & \bigvee_{k=1}^{j-1} [\text{ROB}_k.\text{Valid} \wedge \text{ROB}_k.\text{RegWrite} \wedge \neg\text{ROB}_k.\text{ValidResult} \\ & \wedge (\text{RS}_i.\text{SrcTag1} = \text{ROB}_k.\text{DestTag})], \quad i = 1, \dots, K, \quad j = 2, \dots, N \end{aligned} \quad (9)$$

where “ \leftarrow ” stands for assignment. In constraints (8), index j starts from 2 since constraint (7') guarantees that if a Reservation Station will produce the result for ROB_1 , then that Reservation Station has both of its data operands ready. Constraints like (8) and (9) were also imposed for the second data operand of each Reservation Station.

The next invariant constraints state that if a Reservation Station contains valid information, then there is an ROB entry waiting for the result from that Reservation Station:

$$\begin{aligned} & \text{RS}_i.\text{Valid} \\ & \Rightarrow \bigvee_{j=1}^N [\text{ROB}_j.\text{Valid} \wedge \text{ROB}_j.\text{RegWrite} \wedge \neg\text{ROB}_j.\text{ValidResult} \\ & \quad \wedge (\text{RS}_i.\text{DestTag} = \text{ROB}_j.\text{DestTag})], \quad i = 1, \dots, K \end{aligned} \quad (10)$$

5. Automatic Case Splits

One of the main reasons for the complexity of automatic formal verification of out-of-order processors with a Reorder Buffer, register renaming, and Reservation Stations is the large number of matchings between ROB entries, waiting for results, and Reservation Stations that could produce those results. If the number of ROB entries, N , is greater than or equal to the number of Reservation Stations, K , then the maximum number of ROB entries that can be waiting for results is K , and the number of matchings will be $K!$, where every matching includes the cases that each of those K ROB entries is either invalid (bit *Valid* is *false*), or will not write to the Register File (bit *RegWrite* is *false*), or its result has been computed (bit *ValidResult* is *true*), or there is a specific Reservation Station that has valid information and will produce the result for that ROB entry. Hence, a possible way to automatically case-split the formal verification is to spawn multiple runs of the tool flow, each handling one of the $K!$ cases, assuming enough computing resources are available, as can be expected for an industrial environment.

To explain the above decomposition in terms of the constraints from Sect. 4, we will be using case-splitting expressions that are variants of constraints (7') and (7) with just one of the K disjuncts to the right of the implication. That is, a variant of (7') will be used to restrict ROB_1 to be either invalid, or have a *RegWrite* bit of *false*, or have its result computed, or map to a specific one of the K Reservation Stations. Similarly, a variant of (7) will be used to restrict each of ROB_2 through ROB_N to be either invalid, or have a *RegWrite* bit of *false*, or have its result computed, or map to a specific Reservation Station that is different from those assigned to previous ROB entries. Furthermore, for each matching between ROB entries and Reservation Stations, we can simplify each of the K constraints (8) and each of the K constraints (10) to a variant with only one of the disjuncts to the right of the implication, i.e., the disjunct corresponding to the ROB entry whose destination tag equals that of the given Reservation Station. Each of the other disjuncts in an instance of (8) or (10) will simplify to *false*, since valid ROB entries have different destination tags, and so there can be only one ROB entry whose destination tag equals that of a Reservation Station. Note that all possible matchings between ROB entries and Reservation Stations can be enumerated automatically, and if an automatic tool flow can prove correctness for each of the matchings, then that would imply correctness over the entire solution space.

6. Merging ITE-Trees with 2 Levels of Leaves

ITE-trees can be further merged with 2 levels of their leaves—see Fig. 3—such that the merged leaves have fanout count of 1.

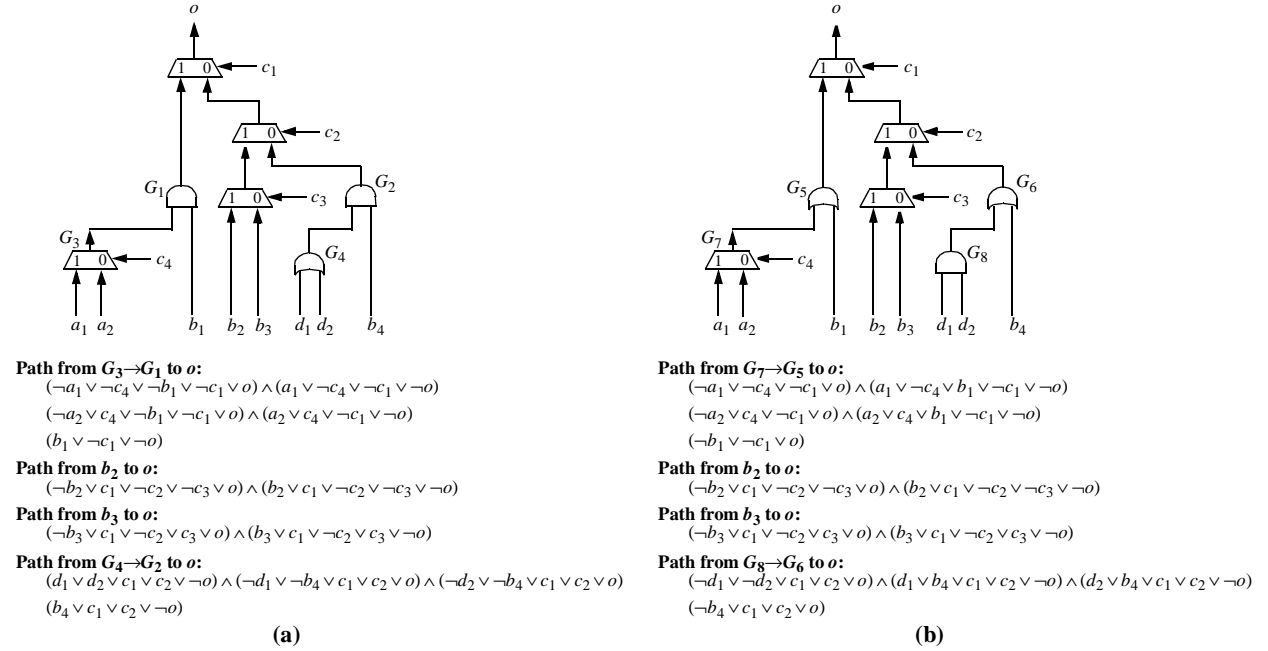


Fig. 3. Merging an ITE-tree with 2 levels of its leaves, where each merged leaf has fanout count of 1: (a) ITE→AND and OR→AND groups as the 2 levels of merged leaves; and (b) ITE→OR and AND→OR groups as the 2 levels of merged leaves.

Any merged first-level leaves are either AND or OR gates (if ITEs with fanout count of 1, they would have been part of the ITE-tree). Due to the hashing scheme for formulas in the decision procedure EVC (see Sect. 2.1), any merged second-level leaf is either an ITE, or an OR driving an AND ($G_4 \rightarrow G_2$ in Fig. 3.a), or an AND driving an OR ($G_8 \rightarrow G_6$ in Fig. 3.b). If a first-level AND/OR is driven by many gates with fanout count of 1, merged was the one with highest topological level (see Sect. 2.3). Note that if G_3 supplies the AND gate G_1 with that gate’s controlling value of *false* in Fig. 3.a—i.e., either a_1 is *false* and is selected to appear at the output of G_3 by c_4 being *true*, or a_2 is *false* and is selected to appear at the output of G_3 by c_4 being *false*—then the value of b_1 does not affect the value of G_1 , and so $\neg b_1$ does not appear in the clauses $(a_1 \vee \neg c_4 \vee \neg c_1 \vee \neg o)$ and $(a_2 \vee c_4 \vee \neg c_1 \vee \neg o)$. Similarly for $G_4 \rightarrow G_2$ and b_4 in Fig. 3.a; for $G_7 \rightarrow G_5$ and b_1 in Fig. 3.b; and for $G_8 \rightarrow G_6$ and b_4 in Fig. 3.b.

7. Results

The experiments were conducted on a Dell OptiPlex GX260 having a 3.06-GHz Intel Pentium 4 with a 512-KB on-chip L2-cache, 2 GB of physical memory, and running Red Hat Linux 9.0. The tool flow, consisting of the term-level symbolic simulator TlSim [96] and an extended version of the decision procedure EVC [96], was combined with the SAT-solver *siege_v4* [74]—an improved version of *siege_v0*, one of the top performers in the 2003 SAT competition [59]. In EVC, equations between term variables were encoded with new e_{ij} Boolean variables [33], while transitivity of equality was enforced as described in [19]. The abstraction function—mapping an implementation state to an equivalent specification state—was computed by using the ideas in controlled flushing [21]: the instructions in the ROB were not allowed to advance, thus reducing the ambiguity of the instruction flow; the ALUs were made to deterministically compute the results of Reservation Stations with ready operands; and the i^{th} ROB entry was allowed to write its result to the Register File in the i^{th} cycle of the abstraction function, i.e., in the latest cycle when that result could be computed. The processor used in the experiments had 6 ROB entries, 6 Reservation Stations, 2 data operands per instruction, and could issue and retire up to 2 instructions per cycle—the same numbers as in the PowerPC 750 [42]. The invariance check of all constraints took 12 seconds, if they were verified in sequence, one constraint at a time, but much longer if verified monolithically. Most of the bugs, made when designing the correct processor, were detected by just checking invariance of the constraints. The discussion next is about the formal verification of safety of the correct model—that 1 step of this dual-issue model corresponds to 0, or 1, or 2 steps of its specification. Safety was checked with the commutative correctness diagram used in [20][93][94][98].

Table 1 shows the results from formal verification with the necessary invariant constraints. The “old” translation to CNF is without preserving the ITE-tree structure of equation arguments when eliminating equations, but using a disjunction of conjunctions. “# Runs” is the number of matchings when some or all of the ROB entries are matched with Reservation Stations to produce the results for those ROB entries—e.g., 720 runs (6!) when each of the 6 ROB entries is matched with a different one of the 6 Reservation Stations. With the old translation to CNF, and without case splits to enumerate matchings, *siege_v4* did not finish in 168 hours (1 week). Using the old translation, and matching all 6 ROB entries with reservation stations for a total of 720 runs, resulted in average SAT time of 644 seconds per run. Merging only ITE-trees [102] reduced the average SAT time to 181 seconds per run. Merging ITE-trees with one level of their leaves [102] further reduced the average SAT time to 102 seconds per run. And merging ITE-trees with 2 levels of leaves (see Sect. 6) resulted in average SAT time of 20 seconds per run, i.e., speedup of 5× relative to translation by merging ITE-trees with 1 level of leaves, 9× relative to translation by merging only ITE-trees, and 32× relative to the old translation from EUFM to CNF. Furthermore, if 720 CPUs are available for parallel runs of the tool flow, we can complete all the runs in 55 seconds (the maximum time for a run), resulting in 2 orders of magnitude speedup relative to the sequential execution time of 14,400 seconds for all runs on 1 CPU. The CNF formulas obtained with the old translation are available as [103]

Additional experiments were conducted to determine the optimum number of ROB entries to match with Reservation Stations in order to minimize the SAT time with sequential runs. That number was found to be 4, requiring 360 runs, and resulting in average time per run of 39 seconds, total sequential time of 14,198 seconds, and maximum time per run of 104 seconds. (The table of results is not shown for lack of space.) Further experiments were used to explore the benefit from additional invariant constraints (also not presented for lack of space) that are not necessary for the formal verification, but reduce the solution space. They did not make a difference, if the formal verification was decomposed sufficiently by matching 4, 5, or 6 ROB entries with Reservation Stations, although resulting in an order of magnitude speedup if fewer ROB entries were matched with Reservation Stations. Without matching any of the ROB entries with Reservation Stations, and so executing a single monolithic run, the new translation to CNF—merging ITE-trees with 2 levels of leaves—required more than 24 hours, regardless of the use of extra invariant constraints.

Translation to CNF	# ROB Entries Matched with Reservation Stations	# Runs	Average Formula Statistics					Average SAT Time [sec]	Total SAT Time [sec]	
			Boolean Variables	CNF Variables	CNF Clauses	CNF Literals	Average Literals per Clause		With Sequential Runs	With Parallel Runs
old	0	1	323	64,627	871,053	2,497,863	2.868	> 168 h	> 168 h	> 168 h
old	6	720	323	47,136	612,784	1,757,288	2.868	644	463,680	872
merge ITE-trees	6	720	323	15,909	185,572	961,451	5.181	181	130,320	272
merge ITE-trees with 1 level of leaves	6	720	323	12,543	178,840	1,138,421	6.366	102	73,440 (20.4 h)	223
merge ITE-trees with 2 levels of leaves	6	720	323	12,410	178,574	1,141,377	6.392	20	14,400 (4 h)	55

Table 1. Results from formal verification of safety with the necessary invariant constraints.

The speedup from the new translation to CNF is due to the structure of the given class of Boolean formulas. When formally verifying models with in-order execution, the Boolean formulas do not have ITE-trees where many first- and second-level leaves have fanout count of 1, and so the new translation could not be applied many times.

Merging ITE-trees with their first-level leaves that have fanout count greater than 1, slowed the SAT-solving by up to $40\times$ [102]. Merging ITE-trees with 3 and more levels of leaves, each with fanout count of 1, resulted in worse performance compared to merging ITE-trees with 2 levels of leaves, indicating that the optimal performance for the given class of Boolean formulas is achieved by merging ITE-trees with 2 levels of their leaves.

8. Discussion

The benefits from merging ITE-trees with their leaves include: **1) Reduced variables and clauses**—relative to conventional CNF translation where each ITE is translated separately—but possibly increased literals. **2) Reduced solution space**—fewer CNF variables allow a SAT-solver to make fewer decisions when evaluating a formula; removing the unimportant variables—representing signals inside ITE-trees—improves the efficiency of the search, allowing a SAT-solver to make decisions only based on important variables that control the branching in ITE-trees, or are leaves of ITE-trees. **3) Reduced BCP**—eliminating intermediate variables for outputs of ITEs inside a tree, allows a SAT-solver to quickly propagate the value of an ITE-tree leaf to the tree output; if the literals increase, the BCP will also increase, but the benefits from reduced variables and clauses more than offset this in the experiments; BCP takes up to 90% of the SAT time [68]. **4) Automatic use of signal unobservability**—the clauses, introduced for each path in an ITE-tree, become satisfied as soon as an ITE-controlling signal selects another path, allowing a SAT-solver more freedom in assigning values to unobservable [25][76] signals. **5) Reduced L2-cache misses**—the fewer variables and clauses result in smaller CNF file sizes, and more succinctly represent the solution space, allowing the clauses for the active portion of the search to better fit in the L2 cache; also, the average number of literals per clause increases, and since the literals for a clause are situated in contiguous memory locations, SAT-solvers can better exploit the spacial locality of memory accesses [38]; big CNFs result in high L2-cache misses, and thus decrease the performance of SAT-solvers [108]. **6) Guiding the SAT-solver branching**—each path, passing through an ITE-tree and its leaves, is due to a different symbolic-execution trace, so that by representing each path as clauses without intermediate CNF variables, we point the SAT-solver toward processing one symbolic-execution trace at a time, and make it easier for the SAT-solver to prune infeasible paths whose clauses contain CNF variables with complemented values.

Sheeran and Stålmarck [82] found that optimal performance of Stålmarck’s method [83] on many formulas from formal verification is achieved when the *dilemma rule*—exploring all possible assignments to a set of Boolean variables, and deducing constraints for the solution space—is applied to either 0 or 1 Boolean variables at a time. The corresponding conclusion from Sect. 7 is that best was the strategy to merge ITE-trees with 2 levels of leaves. The two conclusions indicate that different classes of Boolean formulas require different levels of SAT-solver branching to achieve optimal learning and optimal SAT-solver performance.

Additional speedup can be expected if `siege_v4` is extended into an incremental SAT-solver, such as [27][28][105], where constraints (in the current paper case splits matching ROB entries with Reservation Stations) are added to the CNF formula incrementally, one constraint at a time. For each constraint, if the formula is proved unsatisfiable, the constraint is removed from the clause database, and so are any conflict clauses triggered by that constraint; however, conflict clauses that apply to the solution space without the constraint are kept, so that the effort for deriving them is amortized across the processing of many constraints.

9. Related Work on SAT and EUFM Decision Procedures

Gupta et al. [35] were first to implement a circuit-based SAT-solver that uses structural information to identify gates with unobservable outputs and remove the clauses for those gates, as long as the gates remain unobservable. Novikov [69] exploited signal observability when deriving relations between CNF variables, by branching on up to 5 CNF variables, recording the resulting binary values for other variables, but keeping don't-cares for variables that do not get a binary value, and then extracting compact relations that hold in all assignments not leading to a conflict. Other circuit-based SAT-solvers identify signals with equal or complemented values in order to prune the solution space [52][62][70], or use a hybrid representation of Boolean circuits [32][70]—gate-level for the circuit, and CNF for constraints and learnt clauses. Franco et al. [31] present a circuit-based SAT-solver that uses BDDs [17] in its decision heuristic, and was faster than Chaff [68] on formulas from Bounded Model Checking, but slower on formulas from EUFM-based formal verification of processors with in-order execution. Theoretical results about circuit-based SAT algorithms are presented in [2][16]. Hong et al. [39] used don't-cares to minimize BDDs.

Kautz et al. [10][75] used depth-first traversal of a pebbling graph in order to generate a partial branching sequence, defining CNF-variable assignments to be made by a SAT-solver when beginning to process the CNF formula for that graph, thus guiding the search towards a solution. Reda et al. [73] used BDD-variable ordering heuristics to derive a CNF-variable decision order for SAT-solving CNFs of circuits. To reduce the cost of BCP, Bingham and Hu [13] compiled Boolean formulas to programs, and simulated the resulting code with random vectors. Additional code was generated to identify input patterns that will produce the same output value as the current vector, thus pruning the solution space.

Variations of Tseitin's transformations [87] were used in [9][14][26][29][49][71], and Larrabee [57] was first to apply them to testing of circuits, but none of these authors used transformations for ITE-trees. Kuehlmann et al. [52] represented Boolean circuits in terms of 2-input AND gates and inverters, and transformed groups of 3 connected AND gates, one driven by the other two, into a canonical form by accounting for inverters at gate inputs.

Algebraic simplifications for CNF [7][15][24][58][61][63][65] require long processing times for big formulas. Three of those methods [15][63][65], as well as techniques for CNF variable ordering by minimizing the cut-width [3][104] (see also [23]), were applied to simpler formulas from formal verification of processors [98], but took long time, and did not accelerate the SAT-solving. However, deriving the direct and indirect implications, as well as the extended backward implications [109], between pairs of signals, and adding those implications as 2-literal clauses to the CNFs from conventional translation, resulted in orders of magnitude speedup [6]. Using multiple parallel runs of a SAT-solver with different decision heuristics [80], or with different translations from EUFM to propositional logic [98], and stopping when a run finds a solution, reduced the SAT time.

Burch [21] extended SVC [46], a decision procedure for EUFM, with simplifications that are based on observability don't-cares, but his method depended on manually provided case-splitting expressions, and was computationally expensive even for simple benchmarks. Jones et al. [46], and Levitt and Olukotun [60] devised heuristics that sped up SVC, but did not scale for complex formulas and were not flexible, as observed by Barrett et al. [8].

Automatic methods for deriving invariant constraints in high-level microprocessors have been proposed [43][56][85][86], but have not been applied to out-of-order designs with register renaming, as well as Reorder Buffer and Reservation Stations that are completely implemented and instantiated, and are likely to run into scaling problems due to the large state spaces of such designs.

Nested ITEs were first used to eliminate uninterpreted functions and uninterpreted predicates in [91], where bit-level functional units were abstracted with read-only instances of an Efficient Memory Model (EMM) [89][90] for behavioral abstraction of memories in symbolic simulation. The EMM has been adopted in verification tools by InnoLogic Systems [37], and Synopsys [51].

10. Conclusions

The paper studied the potential for automatic decomposition and speedup in formal verification of out-of-order superscalar processors, having register renaming, as well as a Reorder Buffer and Reservation Stations that are completely implemented and instantiated. This is in contrast to previous approaches, where these hardware structures are manually abstracted, and the user has to set up an inductive proof over the number of Reorder Buffer entries, and possibly manually apply symmetry reductions to decrease the number of Reservation Stations. Furthermore, and also in contrast to previous approaches, the Reorder Buffer was not extended with auxiliary state in order to simplify the expressions resulting from the abstraction function; significant additional speedup can be expected with such approaches, at the cost of extra manual work. The formal verification was possible due to automatically generated case-splitting

expressions—matching Reorder Buffer entries with Reservation Stations that will compute the data operands for those Reorder Buffer entries, and resulting in orders of magnitude speedup if many CPUs are available for parallel runs of the tool flow. An efficient translation from the logic of EUFM to CNF—by producing more ITEs, and merging ITE-trees with 2 levels of their leaves—resulted in additional 32× speedup. Future work will examine more complex out-of-order processors, and will fine-tune the translation to CNF.

References

- [1] W. Ackermann, *Solvable Cases of the Decision Problem*, North-Holland, Amsterdam, 1954.
- [2] M. Alekhovich, and A.A. Razborov, “Satisfiability, Branch-Width and Tseitin Tautologies,” *Symposium on Foundations of Computer Science (FOCS '02)*, November 2002.
- [3] 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*, 2001.
- [4] T. Arons, and A. Pnueli, “Verifying Tomasulo’s Algorithm by Refinement,” *12th International Conference on VLSI Design (VLSI '99)*, June 1999.
- [5] T.Arons, and A.Pnueli, “A Comparison of Two Verification Methods for Speculative Instruction Execution,” *Tools and Algorithms for the Construction and Analysis of Systems (TACAS '00)*, S. Graf, and M. Schwartzbach, eds., LNCS 1785, Springer-Verlag, March–April 2000, pp. 487–502.
- [6] R. Arora, and M.S. Hsiao, “Enhancing SAT-Based Equivalence Checking with Static Logic Implications,” *High Level Design Validation and Test Workshop (HLDVT '03)*, November 2003.
- [7] F. Bacchus, and J. Winter, “Effective Preprocessing with Hyper-Resolution and Equality Reduction,” *Theory and Applications of Satisfiability Testing (SAT '03)*, 2003.
- [8] C. Barrett, D. Dill, and A. Stump, “Checking Satisfiability of First-Order Formulas by Incremental Translation to SAT,” *Computer-Aided Verification (CAV '02)*, July 2002.
- [9] M. Bauer, D. Brand, M. Fischer, A. Meyer, and M. Paterson, “A Note on Disjunctive Form Tautologies,” *SIGACT News*, Vol. 4 (1973).
- [10] P. Beame, H. Kautz, and A. Sabharwal, “Understanding the Power of Clause Learning,” *International Joint Conference on Artificial Intelligence (IJCAI '03)*, August 2003.
- [11] S. Berezin, E. Clarke, A. Biere, and Y. Zhu, “Verification of Out-Of-Order Processor Designs Using Model Checking and a Light-Weight Completion Function,” *Journal on Formal Methods in System Design (FMSD)*, special issue on Microprocessor Verifications, Vol. 20, No. 2 (March 2002).
- [12] A. Biere, and W. Kunz, “SAT and ATPG: Boolean Engines for Formal Hardware Verification,” *International Conference on Computer Aided Design (ICCAD '02)*, November 2002, pp. 782–785.
- [13] J.D. Bingham, and A.J. Hu, “Semi-Formal Bounded Model Checking,” *Computer-Aided Verification (CAV '02)*, LNCS 2404, Springer-Verlag, July 2002.
- [14] T. Boy de la Tour, “An Optimality Result for Clause Form Translation,” *Journal of Symbolic Computation*, Vol. 14 (1992), pp. 283–301.
- [15] R.I. Brafman, “A Simplifier for Propositional Formulas with Many Binary Clauses,” *Int'l. Joint Conference on Artificial Intelligence (IJCAI '01)*, 2001.
- [16] E. Broering, and S.V. Lokam, “Width-Based Algorithms for SAT and CIRCUIT-SAT,” *Theory and Applications of Satisfiability Testing (SAT '03)*, May 2003.
- [17] R.E. Bryant, “Graph-Based Algorithms for Boolean Function Manipulation,” *IEEE Transactions on Computers*, Vol. C-35, No. 8 (August 1986), pp. 677–691.
- [18] 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.
- [19] R.E. Bryant, and M.N. Velev, “Boolean Satisfiability with Transitivity Constraints,” *ACM Transactions on Computational Logic (TOCL)*, Vol. 3, No. 4 (October 2002).
- [20] J.R. Burch, and D.L. Dill, “Automated Verification of Pipelined Microprocessor Control,” *Computer-Aided Verification (CAV '94)*, LNCS 818, Springer-Verlag, June 1994.
- [21] J.R. Burch, “Techniques for Verifying Superscalar Microprocessors,” *33rd Design Automation Conference (DAC '96)*, June 1996.
- [22] H.K. Büning, and T. Lettmann, *Propositional Logic: Deduction and Algorithms*, Cambridge Tracts in Theoretical Computer Science 48, Cambridge University Press, 1999.
- [23] E.M. Clarke, and O. Strichman, “A Failed Attempt to Optimize Variable Ordering with Tools for Constraints Solving,” *Workshop on Constraints in Formal Verification*, 2002.
- [24] J.M. Crawford, and L.D. Auton, “Experimental Results on the Crossover Point in Satisfiability Problems,” *National Conference on Artificial Intelligence*, 1993.
- [25] M. Damiani, and G. De Micheli, “Observability Don’t Care Sets and Boolean Relations,” *International Conference on Computer-Aided Design (ICCAD '90)*, 1990.
- [26] E. Eder, “An Implementation of a Theorem Prover Based on the Connection Method,” *Artificial Intelligence: Methodology, Systems, Applications (AIMSA '84)*, 1985.
- [27] N. Eén, and N. Sörensson, “An Extensible SAT-solver,” *6th International Conference on Theory and Applications of Satisfiability Testing (SAT '03)*, May 2003.
- [28] N. Eén, and N. Sörensson, “Temporal Induction by Incremental SAT Solving,” *Workshop on Bounded Model Checking (BMC '03)*, ENTCS, Vol. 89, No. 4, 2003.
- [29] U. Egly, and T. Rath, “On the Practical Value of Different Definitional Translations to Normal Form,” *International Conference on Automated Deduction (CADE '96)*, 1996.
- [30] J.-C. Filliâtre, S. Owre, H. Rueß, and N. Shankar, “ICS: Integrated Canonizer and Solver,” *Computer-Aided Verification (CAV '01)*, G. Berry, H. Comon, and A. Finkel, eds., LNCS 2102, Springer-Verlag, July 2001, pp. 246–249.
- [31] J. Franco, M. Kouril, J. Schlipf, J. Ward, S. Weaver, M. Dransfield, and W.M. Vanfleet, “SBSAT: A State-Based, BDD-Based Satisfiability Solver,” *6th International Conference on Theory and Applications of Satisfiability Testing (SAT '03)*, May 2003.
- [32] 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.
- [33] 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, pp. 244–255.
- [34] 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.
- [35] 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, pp. 536–541.
- [36] S. Hangal, and M. O’Connor, “Performance Analysis and Validation of the picoJava Processor,” *IEEE Micro*, May–June 1999, pp. 62–72.
- [37] G. Hasteer, Personal communication, February 1999.

- [38] J.L. Hennessy, and D.A. Patterson, *Computer Architecture: A Quantitative Approach*, 3rd edition, Morgan Kaufmann, San Francisco, 2002.
- [39] Y. Hong, P. Beeler, J. Burch, and K. McMillan, "Safe BDD Minimization Using Don't Cares," *Design Automation Conference (DAC '97)*, June 1997.
- [40] R. Hosabettu, M. Srivas, and G. Gopalakrishnan, "Proof of Correctness of a Processor with Reorder Buffer Using the Completion Functions Approach," *Computer-Aided Verification (CAV '99)*, N. Halbwachs, and D. Peled, eds., LNCS 1633, Springer-Verlag, July 1999.
- [41] R.M. Hosabettu, "Systematic Verification of Pipelined Microprocessors," Ph.D. Thesis, Department of Computer Science, University of Utah, August 2000.
- [42] IBM Corporation, *PowerPC 740TM/PowerPC 750TM: RISC Microprocessor User's Manual*, 1999.
- [43] A.J. Isles, R. Hojati, and R.K. Brayton, "Computing Reachable Control States of Systems Modeled with Uninterpreted Functions and Infinite Memory," *Computer-Aided Verification (CAV '98)*, A.J. Hu, and M.Y. Vardi, eds., LNCS 1427, Springer-Verlag, June 1998.
- [44] R. Jhala, and K.L. McMillan, "Microarchitecture Verification by Compositional Model Checking," *Computer-Aided Verification (CAV '01)*, LNCS 2102, July 2001.
- [45] 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>
- [46] 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.
- [47] R.B. Jones, J.U. Skakkebaek, and D.L. Dill, "Formal Verification of Out-of-Order Execution with Incremental Flushing," *Journal on Formal Methods in System Design (FMSD)*, special issue on Microprocessor Verifications, Vol. 20, No. 2 (March 2002), pp. 139–158.
- [48] R.B. Jones, *Symbolic Simulation Methods for Industrial Formal Verification*, Kluwer Academic Publishers, Boston/Dordrecht/London, 2002.
- [49] T.A. Junttila, and I. Niemelä, "Towards and Efficient Tableau Method for Boolean Circuit Satisfiability Checking," *International Conference on Computational Logic (CL '00)*, LNAI 1861, Springer-Verlag, July 2000.
- [50] H. Kautz, and B. Selman, "Ten Challenges Redux: Recent Progress in Propositional Reasoning and Search," *Principles and Practice of Constraint Programming (CP '03)*, F. Rossi, ed., LNCS 2833, Springer-Verlag, September–October 2003.
- [51] A. Kölbl, J.H. Kukula, K. Antreich, and R.F. Damiano, "Handling Special Constructs in Symbolic Simulation," *39th Design Automation Conference (DAC '02)*, June 2002.
- [52] A. Kuehlmann, M.K. Ganai, and V. Paruthi, "Circuit-Based Boolean Reasoning," *38th Design Automation Conference (DAC '01)*, June 2001.
- [53] S. Lahiri, C. Pixley, and K. Albin, "Experience with Term Level Modeling and Verification of the M•CORETM Microprocessor Core," *International Workshop on High Level Design, Validation and Test (HLDVT '01)*, November 2001.
- [54] 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.
- [55] S.K. Lahiri, and R.E. Bryant, "Deductive Verification of Advanced Out-of-Order Microprocessors," *Computer-Aided Verification (CAV '03)*, LNCS, July 2003.
- [56] S.K. Lahiri, R.E. Bryant, and B. Cook, "A Symbolic Approach to Predicate Abstraction," *Computer-Aided Verification (CAV '03)*, LNCS, July 2003.
- [57] T. Larrabee, "Test Pattern Generation Using Boolean Satisfiability," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 11, No. 1, 1992.
- [58] D. Le Berre, "Exploiting the Real Power of Unit Propagation Lookahead," *Workshop on Theory and Applications of Satisfiability Testing (SAT '01)*, H. Kautz, and B. Selman, eds., Elsevier Science Publishers, Electronic Notes in Discrete Mathematics, Vol. 9, June 2001.
- [59] 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.
- [60] J. Levitt, and K. Olukotun, "Verifying Correct Pipeline Implementation for Microprocessors," *International Conference on Computer-Aided Design (ICCAD '97)*, 1997.
- [61] C.M. Li, and Anbulagan, "Look-Ahead versus Look-Back for Satisfiability Problems," *Principles and Practice of Constraint Programming (CP '97)*, LNCS 1330, 1997.
- [62] 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.
- [63] I. Lynce, and J.P. Marques-Silva, "Probing-Based Preprocessing Techniques for Propositional Satisfiability," *International Conference on Tools with Artificial Intelligence (ICTAI '03)*, November 2003.
- [64] S. Malik, A.R. Wang, R.K. Brayton, and A. Sangiovanni-Vincentelli, "Logic Verification Using Binary Decision Diagrams in a Logic Synthesis Environment," *International Conference on Computer-Aided Design (ICCAD '88)*, November 1988.
- [65] J.P. Marques-Silva, "Algebraic Simplification Techniques for Propositional Satisfiability," *Principles and Practice of Constraint Programming (CP '00)*, September 2000.
- [66] K.L. McMillan, "Circular Compositional Reasoning about Liveness," Technical Report, Cadence Berkeley Labs, Cadence Design Systems, 1999.
- [67] K.L. McMillan, "A Methodology for Hardware Verification Using Compositional Model Checking," *Science of Computer Programming*, Vol. 37, No. 1–3 (May 2000).
- [68] 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.
- [69] Y. Novikov, "Local Search for Boolean Relations on the Basis of Unit Propagation," *Design, Automation and Test in Europe (DATE '03)*, March 2003.
- [70] R. Ostrowski, E. Grégoire, B. Mazure, and L. Saïs, "Recovering and Exploiting Structural Knowledge from CNF Formulas," *Principles and Practice of Constraint Programming (CP '02)*, P. Van Hentenryck, ed., LNCS 2470, Springer-Verlag, September 2002, pp. 185–199.
- [71] D.A. Plaisted, and S. Greenbaum, "A Structure Preserving Clause Form Translation," *Journal of Symbolic Computation (JSC)*, Vol. 2, 1985, pp. 293–304.
- [72] 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, 2002.
- [73] S. Reda, R. Drechsler, and A. Orailoglu, "On the Relation Between SAT and BDDs for Equivalence Checking," *Symposium on Quality of Electronic Design*, 2002.
- [74] L. Ryan, Siege SAT Solver v.4. <http://www.cs.sfu.ca/~loryan/personal/>
- [75] A. Sabharwal, P. Beame, and H. Kautz, "Using Problem Structure for Efficient Clause Learning," *Theory and Applications of Satisfiability Testing (SAT '03)*, 2003.
- [76] H. Savoj, and R.K. Brayton, "The Use of Observability and External Don't Cares for the Simplification of Multi-Level Networks," *Design Automation Conference*, 1990.
- [77] J. Sawada, "Formal Verification of an Advanced Pipelined Machine," Ph.D. Thesis, Department of Computer Sciences, University of Texas at Austin, December 1999.
- [78] J. Sawada, and W.A. Hunt, Jr., "Verification of FM9801: Out-of-Order Processor with Speculative Execution and Exceptions That May Execute Self-Modifying Code," *Journal on Formal Methods in System Design (FMSD)*, special issue on Microprocessor Verifications, Vol. 20, No. 2 (March 2002), pp. 187–222.

- [79] 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)*, June 2003.
- [80] O. Shacham, and E. Zarpas, "Tuning the VSIDS Decision Heuristic for Bounded Model Checking," *Microprocessor Test and Verification (MTV '03)*, May 2003.
- [81] J.P. Shen, and M. Lipasti, *Modern Processor Design: Fundamentals of Superscalar Processors*, beta edition, McGraw-Hill, July 2002.
- [82] M. Sheeran, and G. Stålmarck, "A Tutorial on Stålmarck's Proof Procedure for Propositional Logic," *Formal Methods in System Design (FMSD)*, Vol. 16, No. 1, 2000.
- [83] G. Stålmarck, "A System for Determining Propositional Logic Theorems by Applying Values and Rules to Triplets that are Generated from a Formula," 1989. Swedish Patent No. 467 076 (approved 1992), U.S. Patent No. 5 276 897 (1994), European Patent No. 0403 454 (1995).
- [84] Stanford Validity Checker (SVC), <http://sprout.Stanford.EDU/SVC>
- [85] J.X. Su, D.L. Dill, and C. Barrett, "Automatic Generation of Invariants in Processor Verification," *Formal Methods in Computer-Aided Design (FMCAD '96)*, 1996.
- [86] A. Tiwari, H. Rueß, H. Saïdi, and N. Shankar, "A Technique for Invariant Generation," *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. 113–127.
- [87] 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.
- [88] O. Tveretina, and H. Zantema, "A Proof System and a Decision Procedure for Equality Logic," Tech. Report, Computer Science, Technical University of Eindhoven, 2003.
- [89] M.N. Velev, and R.E. Bryant, "Efficient Modeling of Memory Arrays in Symbolic Ternary Simulation," *Tools and Algorithms for the Construction and Analysis of Systems (TACAS '98)*, B. Steffen, ed., LNCS 1384, Springer-Verlag, March–April 1998.
- [90] M.N. Velev, and R.E. Bryant, "Incorporating Timing Constraints in the Efficient Memory Model for Symbolic Ternary Simulation," *International Conference on Computer Design (ICCD '98)*, October 1998, pp. 400–406.
- [91] M.N. Velev, and R.E. Bryant, "Bit-Level Abstraction in the Verification of Pipelined Microprocessors by Correspondence Checking," *Formal Methods in Computer-Aided Design (FMCAD '98)*, LNCS 1522, Springer-Verlag, November 1998, pp. 18–35.
- [92] 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.
- [93] 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.
- [94] 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.
- [95] 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.
- [96] 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.
- [97] M.N. Velev, "Using Rewriting Rules and Positive Equality to Formally Verify Wide-Issue Out-Of-Order Microprocessors with a Reorder Buffer," *Design, Automation and Test in Europe (DATE '02)*, March 2002, pp. 28–35.
- [98] 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.
- [99] M.N. Velev, "Integrating Formal Verification into an Advanced Computer Architecture Course," *ASEE Annual Conference & Exposition*, June 2003.
- [100] M.N. Velev, "Automatic Abstraction of Equations in a Logic of Equality," *Automated Reasoning with Analytic Tableaux and Related Methods (TABLEAUX '03)*, 2003.
- [101] M.N. Velev, "Efficient Translation of Boolean Formulas to CNF in Formal Verification of Microprocessors," *Asia and South Pacific Design Automation Conference*, January 2004.
- [102] 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.
- [103] M.N. Velev, CNF Benchmark Suite ENGINE-UNSAT-1.0, August 2003. <http://www.ece.cmu.edu/~mvelev>
- [104] 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.
- [105] J. Whittemore, J. Kim, and K. Sakallah, "SATIRE: A New Incremental Satisfiability Engine," *38th Design Automation Conference (DAC '01)*, June 2001.
- [106] H. Zantema, and J.F. Groote, "Transforming Equality Logic to Propositional Logic," *Workshop on First Order Theorem Proving (FTP '03)*, June 2003.
- [107] L. Zhang, and S. Malik, "The Quest for Efficient Boolean Satisfiability Solvers," *Computer-Aided Verification (CAV '02)*, E. Brinksma, and K.G. Larsen, eds., LNCS 2404, Springer-Verlag, July 2002, pp. 17–36.
- [108] L. Zhang, and S. Malik, "Cache Performance of SAT Solvers: A Case Study for Efficient Implementation of Algorithms," *Theory and Applications of Satisfiability Testing (SAT '03)*, May 2003.
- [109] J.-K. Zhao, E.M. Rudnick, and J.H. Patel, "Static Logic Implication with Application to Redundancy Identification," *IEEE VLSI Test Symposium (VTS '97)*, April–May 1997.