Effective Use of Boolean Satisfiability Procedures in the Formal Verification of Superscalar and VLIW Microprocessors[1]

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

Randal E. Bryant*
randy.bryant@cs.cmu.edu
http://www.cs.cmu.edu/~bryant

*Department of Electrical and Computer Engineering
School of Computer Science
Carnegie Mellon University, Pittsburgh, PA 15213, U.S.A.

Abstract

We compare SAT-checkers and decision diagrams on the evaluation of Boolean formulas produced in the formal verification of both correct and buggy versions of superscalar and VLIW microprocessors. We identify one SAT-checker that significantly outperforms the rest. We evaluate ways to enhance its performance by variations in the generation of the Boolean correctness formulas. We reassess optimizations previously used to speed up the formal verification and probe future challenges.

1 Introduction

In the past few years, SAT-checkers have made a dramatic improvement in both their speed and capacity. We compare 28 of them with decision diagrams—BDDs [7] and BEDs [61]—as well as with ATPG tools [21][52] when used as Boolean Satisfiability (SAT) procedures in the formal verification of microprocessors. The comparison is based on two benchmark suites, each of 101 Boolean formulas generated in the verification of 1 correct and 100 buggy versions of the same design—a superscalar and a VLIW microprocessor, respectively. Unlike existing benchmark suites, e.g., ISCAS 85 [5] and ISCAS 89 [6], which are collections of circuits that have nothing in common, our suites are based on the same correct design and hence provide a point for consistent comparison of different evaluation methods.

The correctness condition that we use is expressed in a decidable subset of First-Order Logic [10]. That allows it either to be checked directly with a customized decision procedure [51] or to be translated to an equivalent Boolean formula [55] that can be evaluated with SAT engines for either proving correctness or finding a counterexample. The latter approach can directly benefit from improvements in the SAT tools.

We identify Chaff [38] as the most efficient SAT-checker for the second verification strategy when applied to both correct and buggy designs. Chaff significantly outperforms BDDs [7] and the SAT-checker DLM-2 [48], the previous most efficient SAT procedures for, respectively, correct and buggy processors. We reevaluate optimizations used to enhance the performance of BDDs and DLM-2 and conclude that many of them are no longer crucial on the same benchmark suites. Our study allows us to eliminate conservative approximations that might result in false negatives and thus consume precious user time for analysis. We also prioritize the optimizations that are still useful with Chaff in the order of their impact on the efficiency of the formal verification.

1. This research was supported by the SRC under contract 00-DC-684.

DAC 2001, June 18-22, 2001, Las Vegas, Nevada, USA.

2 Background

The formal verification is done by correspondence checking—comparison of the superscalar/VLIW Implementation against a non-pipelined Specification, based on the Burch and Dill flushing technique [10]. The correctness criterion is expressed as a formula in the logic of Equality with Uninterpreted Functions and Memories (EUFM) [10] and states that all user-visible state elements in the processor should be updated in sync by either 0, or 1, or up to k instructions after each clock cycle, where k is the issue width of the design. The correctness formula is then translated to a Boolean formula by an automatic tool [55] that exploits the properties of Positive Equality [8], the e_{ij} encoding [18], and a number of conservative approximations. The resulting Boolean formula should be a tautology in order for the processor to be correct and can be evaluated by any SAT procedure.

The syntax of EUFM [10] includes terms and formulas. Terms are used in order to abstract word-level values of data, register identifiers, memory addresses, as well as the entire states of memory arrays. A term can be an Uninterpreted Function (UF) applied on a list of argument terms, a domain variable, or an *ITE* operator selecting between two argument terms based on a controlling formula, such that *ITE*(*formula*, *term*1, *term*2) will evaluate to *term*1 when *formula* = **true** and to *term*2 when *formula* = **false**. The syntax for terms can be extended to model memories by means of the functions *read* and *write* [10][59]. Formulas are used in order to model the control path of a microprocessor, as well as to express the correctness condition. A formula can be an Uninterpreted Predicate (UP) applied on a list of argument terms, a propositional 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 connected by Boolean connectives. We will refer to both terms and formulas as expressions.

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

Two possible ways to impose the property of functional consistency of UFs and UPs are Ackermann constraints [1] and nested *ITE*s [3][4][21]. The Ackermann scheme replaces each UF (UP) application in the EUFM formula F with a new domain variable (propositional variable) and then adds external consistency constraints. For example, the UF application $f(a_1, b_1)$ will be replaced by a new domain variable c_1, another application of the same UF, $f(a_2, b_2)$, will be replaced by a new domain variable c_2. Then, the resulting EUFM formula F' will be extended as $[(a_1 = a_2) \wedge (b_1 = b_2) \Rightarrow (c_1 = c_2)] \Rightarrow F'$. In the nested *ITE*s

scheme, the first application of the UF above will still be replaced by a new domain variable c_1. However, the second one will be replaced by $ITE((a_2 = a_1) \wedge (b_2 = b_1), c_1, c_2)$, where c_2 is a new domain variable. A third one, $f(a_3, b_3)$, will be replaced by $ITE((a_3 = a_1) \wedge (b_3 = b_1), c_1, ITE((a_3 = a_2) \wedge (b_3 = b_2), c_2, c_3))$, where c_3 is a new domain variable, and so on. Similarly for UPs.

Positive Equality allows the identification of two types of terms in the structure of an EUFM formula—those which appear in only positive equations and are called *p-terms* (for positive terms), and those which appear in both positive and negative equations and are called *g-terms* (for general terms). A negative equation is one which appears under an odd number of negations or as part of the controlling formula for an *ITE* operator. The efficiency from exploiting Positive Equality is due to the observation that the truth of an EUFM formula under a maximally diverse interpretation of the p-terms implies the truth of the formula under any interpretation. A maximally diverse interpretation is one where the equality comparison of a domain variable with itself evaluates to **true**, that of a p-term domain variable with a syntactically distinct domain variable evaluates to **false**, and that of a g-term domain variable with a syntactically distinct g-term domain variable (a g-equation) could evaluate to either **true** or **false** and can be encoded with Boolean variables [18][40].

3 Microprocessor Benchmarks

We base our comparison of SAT procedures on a set of high-level microprocessors, ranging from a single-issue 5-stage pipelined DLX [23], 1×DLX-C, to a dual-issue superscalar DLX with multicycle functional units, exceptions, and branch prediction, 2×DLX-CC-MC-EX-BP [56], to a 9-wide VLIW architecture, 9VLIW-MC-BP [57], that imitates the Intel Itanium [25] [49] in speculative features such as predicated execution, speculative register remapping, advanced loads, and branch prediction.

The VLIW design is far more complex than any other that has been formally verified previously in an automatic way. It has a fetch engine that supplies the execution engine with a packet of 9 instructions, with no internal data dependencies. Each of these instructions is already matched with one of 9 execution pipelines of 4 stages: 4 integer pipelines, two of which can perform both integer and floating-point memory accesses; 2 floating-point pipelines; and 3 branch-address computation pipelines. Every instruction is predicated with a qualifying predicate identifier, such that the result of that instruction affects user-visible state only when the predicate evaluates to 1. Data values are stored in 4 register files: integer, floating-point, predicate, and branch-address. The two floating-point ALUs, as well as the Instruction and Data Memories, can each take multiple cycles for computing a result or completing a fetch, respectively. There can be up to 42 instructions in flight. An extended version, 9VLIW-MC-BP-EX, also implements exceptions.

We created 100 incorrect versions of both 2×DLX-CC-MC-EX-BP and 9VLIW-MC-BP. The bugs were variants of actual errors made in the design of the correct versions and also coincided with the types of bugs that Van Campenhout, *et al.* [54] analyzed to be among the most frequent design errors. The injected bugs included omitting inputs to logic gates, e.g., an instruction is not squashed when a preceding branch is taken or a stalling condition for the load interlock does not fully account for the cases when the dependent data operand will be used. Other types of bugs were due to using incorrect inputs to logic gates, functional units, or memories, e.g., an input with the same name but a different index. Finally, lack of mechanisms to correct a speculative update of a user-visible state element when the speculation is incorrect. Hence, the variations introduced were not completely random, as done in other efforts to generate benchmark suites [22][26][27][36]. The bugs were spread over the entire designs and occurred either as single or multiple errors.

4 Comparison of SAT Procedures

We evaluated 28 SAT-checkers: SATO.3.2.1 [44][63]; GRASP [17][32] [33], used both with a single strategy and with restarts, randomization, and recursive learning [2]; CGRASP [12][34], a version of GRASP that exploits structural information; DLM-2 and DLM-3 [48], as well as DLM-2000 [62], all incomplete SAT-checkers (i.e., they cannot prove unsatisfiability) based on global random search and discrete Lagrangian Multipliers as a mechanism to not only get the search out of local minima, but also steer it in the direction towards a global minimum—a satisfying assignment; satz [30][45], satz.v213 [30][45], satz-rand.v4.6 [19] [45], eqsatz.v20 [31]; GSAT.v41 [45][47], WalkSAT.v37 [45] [46]; posit [16][45]; ntab [13][45]; rel_sat.1.0 and rel_sat.2.1 [3][45]; rel_sat_rand1.0 [19][45]; ASAT and C-SAT [15]; CLS [41]; QSAT [39] and QBF [42], two SAT-checkers for quantified Boolean formulas; ZRes [11], a SAT-checker combining Zero-Supressed BDDs (ZBDDs) with the original Davis-Putnam procedure; BSAT and IS-USAT, both based on BDDs and exploiting the properties of unate Boolean functions [29]; Prover, a commercial SAT-checker based on Stålmarck's method [50]; Heer-Hugo [20], also based on the same method; and Chaff [38], a complete SAT-checker exploiting lazy Boolean constraint propagation, non-chronological backtracking, restarts, randomization, and many optimizations.

Additionally, we experimented with 2 of the fastest (and publicly available) ATPG tools—ATOM [21] and TIP [52]—used in a mode that tests the output of a benchmark for being stuck-at-0, which triggers the justification of value 1 at the output, turning the ATPG tool into a SAT-checker. We also used Binary Decision Diagrams (BDDs) [7] and Boolean Expression Diagrams (BEDs) [61]—the latter not being a canonical representation of Boolean functions, but shown to be extremely efficient when formally verifying multipliers [60].

The translation to the CNF format [28], used as input to most SAT-checkers, was done after inserting a negation at the top of the Boolean correctness formula that has to be a tautology in order for the processor to be correct. If the formula is indeed a tautology, its negation will be **false**, so that a complete SAT-checker will be able to prove unsatisfiability. Else, a satisfying assignment for the negation will be a counterexample.

In translating to CNF, we introduced a new *auxiliary Boolean variable* for the output of every *AND*, *OR*, or *ITE* gate in the Boolean correctness formula and then imposed disjunctive constraints (clauses) that the value of a variable at the output of a gate be consistent with the values of the variables at the inputs, given the function of the gate. Inverters were subsumed in the clauses for the driven gates. All clauses were conjuncted together, including a constraint that the only primary output (the negation of the Boolean correctness formula) is **true**. The variables in the support of the Boolean correctness formula before its translation to CNF will be called *primary Boolean variables*.

The experiments were performed on a 336 MHz Sun4 with 1.2 GB of memory and 1 GB of swap space. CUDD [14] and the sifting dynamic variable reordering heuristic [43] were used for the BDD-based runs. In the BED evaluations, we experimented with converting the final BED into a BDD with both the `up_one()` and `up_all()` functions [61] by employing 4 different variable ordering heuristics—variants of the depth-first and fanin [37] heuristics—that were the most efficient in the verification of multipliers [60][61].

The SAT procedures that scaled for the 100 buggy variants of 2×DLX-CC-MC-EX-BP are listed in Table 1. The rest of the SAT solvers had trouble even with the single-issue processor, 1×DLX-C, or could not scale for its dual-issue version, 2×DLX-CC (without exceptions, multicycle functional units, and branch prediction). The SAT-checker Chaff had the best performance, finding a satisfying assignment for each benchmark in less than

40 seconds (indeed, less than 37 seconds). We ran the rest of the SAT procedures for 400 and 4,000 seconds—one and two orders of magnitude more, respectively. DLM-2 was the second most efficient SAT-checker for this suite, closely followed by DLM-3. CGRASP was next, solving only half of the benchmarks in 400 seconds, followed by QSAT with 49 of the benchmarks under 400 seconds. The rest of the SAT procedures, including BDDs, performed significantly worse. DLM-2000 is slower than DLM-2 and DLM-3 because of extensive analysis before each decision.

SAT Procedure	% Satisfiable in		
	< 40 sec	< 400 sec	< 4,000 sec
Chaff	100	100	100
DLM-2	61	90	98
DLM-3	58	86	99
CGRASP	46	50	71
QSAT	40	49	52
SATO	22	39	71
rel_sat.1.0	13	20	22
WalkSAT	13	18	32
rel_sat_rand	10	27	34
DLM-2000	9	37	70
GRASP	6	27	48
GRASP + restarts	6	11	18
CLS	5	8	10
rel_sat.2.1	4	71	99
eqsatz	3	4	5
BDDs	2	2	5

Table 1: Comparison of SAT procedures on 100 buggy versions of 2xDLX-CC-MC-EX-BP.

When verifying the correct 2×DLX-CC-MC-EX-BP, Chaff again had the best performance, requiring 40 seconds of CPU time, followed by BDDs with 2,635 seconds [56], and QSAT with 14 hours and 37 minutes. CGRASP, SATO, GRASP, and GRASP with restarts, randomization, and recursive learning could not prove the CNF formula unsatisfiable in 24 hours.

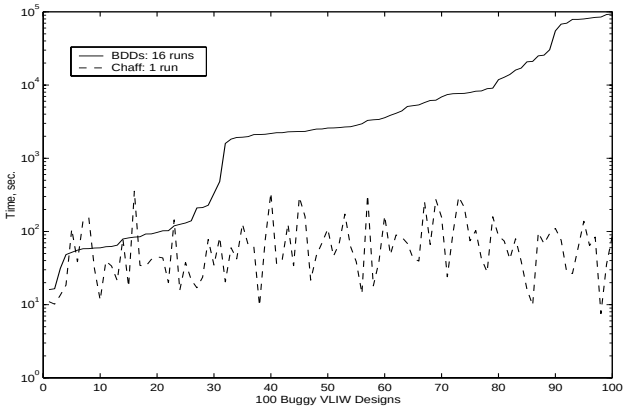

Figure 1: Comparison of Chaff and BDDs on 100 buggy versions of 9VLIW-MC-BP. The benchmarks are sorted in ascending order of their times for the BDD-based experiment.

We then compared Chaff and DLM-2 on the 100 buggy VLIW designs: Chaff was better in 77 cases, with DLM-2 being faster with more than 60 seconds on only 10 benchmarks. However, Chaff took at most 355 seconds, and 79 seconds on average, while DLM-2 did not complete 2 of the benchmarks in 3,600 seconds (we tried 4 different parameter sets). When verifying the correct 9VLIW-MC-BP, Chaff required 1,644 seconds, compared to the 31.5 hours by BDDs [57], using a monolithic correctness criterion in both cases. Figure 1 compares Chaff and BDDs on the 100 buggy VLIW designs, such that Chaff is evaluating only one monolithic correctness criterion, while BDDs evaluate 16 weak (and easier) criteria in parallel [57]. The assumption is that there are enough computing resources to support parallel runs of the tool. As soon as one of these parallel runs comes with a counterexample, we terminate the rest, and consider the minimum time as the verification time. As shown, the difference between BDDs and Chaff is up to 4 orders of magnitude.

Applying the script `simplify` [35] in order to perform algebraic simplifications on the CNF formula for one of the buggy VLIW designs required more than 47,000 seconds, while Chaff took only 14 seconds to find a satisfying assignment without simplifications. This is not surprising, given the CNF formula sizes of up to 450,000 clauses with up to 25,000 variables.

Hence, based on experiments with two suites consisting of 100 buggy designs and their correct counterpart, we identified Chaff as the most efficient SAT procedure—more than 2 orders of magnitude faster than other SAT solvers—for evaluating Boolean formulas generated in the formal verification of complex microprocessors with realistic features. How does this change the frontier of possibilities? The rest of the paper examines ways to increase the productivity in formal verification of microprocessors by using Chaff as the back-end SAT-checker.

5 Impact of Structural Variations in Generating the Boolean Correctness Formulas

Early reduction of p-equations. When eliminating UFs and UPs that take only p-terms as arguments, the translation algorithm introduces equations between argument terms in order to enforce the functional consistency property by nested *ITE*s [8][55]. The argument terms consist of only nested *ITE*s that select one among a set of supporting domain variables. If the terms on both sides of an equation have disjoint supports of p-term domain variables, then the two compared terms will not be equal under a maximally diverse interpretation and their equation can be replaced with **false**. This is already done in the final step of the translation algorithm [55]. However, an early reduction of such equations will result in a different structure of the DAG for the final Boolean formula, i.e., in a different (but equivalent) CNF formula to be evaluated by SAT-checkers.

Eliminating UPs with Ackermann constraints. Ackermann constraints [1] result in a negated equation for the outputs of the eliminated UF or UP: $[(a_1 = a_2) \wedge (b_1 = b_2) \Rightarrow (c_1 = c_2)] \Rightarrow F'$, which is equivalent to: $(a_1 = a_2) \wedge (b_1 = b_2) \wedge \neg(c_1 = c_2) \vee F'$. The negated equation for the output values c_1 and c_2 means that they cannot be p-terms—something that we want to avoid in order to exploit the computational efficiency of Positive Equality. Therefore, Ackermann constraints should not be used for eliminating UFs whose results appear only in positive equations. However, they can be used when eliminating UPs—then the negated equations will be over Boolean variables and that is not a problem when using Positive Equality. Hence, Ackermann constraints can be used instead of nested *ITE*s for eliminating UPs.

The data points for 4 runs with structural variations, shown in Fig. 2, are the minimum times among 4 parallel runs: one with no structural variations (the data plotted for 1 run), one for each of the above variations used alone, and one for both variations combined. The data for 4 runs with parameter variations are the minimum among the 1 run with no structural variations and 3 additional runs where some of the input parameters to Chaff were changed. The average time for finding a satisfying assignment when using structural variations is 45.8 seconds, with the maximum being 278 seconds, compared to 45 and 254 seconds, respectively, with parameter variations. Therefore, the effect of structural variations is almost identical with that of parameter variations, as can be seen in the figure. Running them in parallel (7 runs) reduces the average time to 37 seconds and the maximum to 218 seconds. Hence, only a few parallel runs with differ-

ent structural and/or parameter variations can help reduce the time for SAT checking with Chaff. Structural variations also accelerated the verification of correct designs with up to 20%.

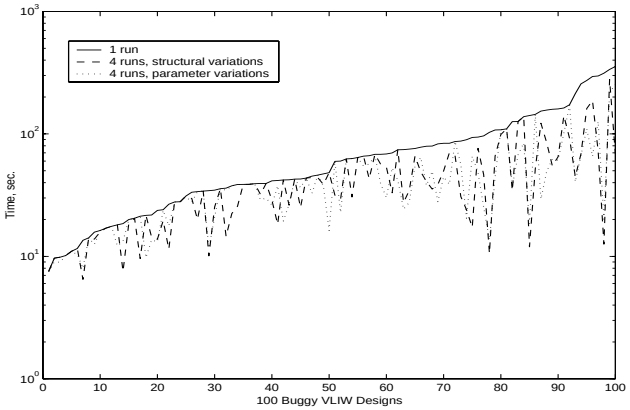

Figure 2: Using structural vs. parameter variations in Chaff. The benchmarks are sorted in ascending order of their times for the experiment with 1 run.

6 Encoding G-Equations

The e_{ij} encoding. The equation $g_i = g_j$, where g_i and g_j are g-term domain variables, is replaced by a unique Boolean variable e_{ij} [18]. Transitivity of equality, $(g_i = g_j) \wedge (g_j = g_k) \Rightarrow (g_i = g_k)$ has to be enforced additionally, e.g., by triangulating the comparison graph of those e_{ij} variables that affect the final Boolean formula and then enforcing transitivity for each of the resulting triangles—sparse transitivity [9]. Although not every correct microprocessor requires transitivity for its correctness proof, that property is needed in order to avoid false negatives for buggy processors or for designs that do need transitivity.

The small domains encoding. Every g-term domain variable is assigned a set of constant values that it can take on in a way that allows it to be either equal to or different from any other g-term domain variable that it can be transitively compared for equality with [40]. If the set of constants for a g-term variable consists of N values, those can be indexed with $\lceil \log_2(N) \rceil$ Boolean variables. Then two g-term domain variables are equal if their indexing Boolean variables select simultaneously a common constant. Note that transitivity is automatically enforced in this encoding. Depending on the structure of the g-term variable comparison graphs, the small domains encoding might introduce fewer primary Boolean variables than the e_{ij} encoding. That would mean a smaller search space. However, now the equality comparison of two g-term domain variables gets replaced with a Boolean formula—a disjunction of conjuncts, each consisting of many Boolean variables or their complements and encoding the possibility that the two g-term domain variables evaluate to the same common constant—instead of just a single Boolean variable.

The two encodings are compared on the 100 buggy VLIW designs in Fig. 3. In a single run of the small domains encoding, the maximum CPU time for detecting a bug is 3,633 seconds and the average is 394 seconds, compared to 355 and 79 seconds, respectively, for the e_{ij} encoding (which was used for the experiments before this section). Constraints for transitivity of equality were included when using the e_{ij} encoding. Structural variations with 4 runs reduced the maximum time with the small domains encoding to 1,240 seconds, and the average to 154 seconds, compared to 154 and 46 seconds, respectively, for the e_{ij} encoding.

When verifying the correct 9VLIW-MC-BP, the small domains encoding resulted in 1,152 primary Boolean variables, with 890 of them being indexing variables, and required 6,008 seconds of CPU time. On the other hand, the e_{ij} encoding resulted in 2,615 primary Boolean variables, with 2,353 of them being e_{ij} variables, and required 1,644 seconds of CPU time. Since this design does not need transitivity of equality for its correctness proof, such constraints were not included in the formula generated with the e_{ij} encoding. Adding these constraints resulted in 705 extra e_{ij} variables due to triangulating the g-term comparison graph, and in 2,680 seconds of CPU time—an increase of over 1,000 seconds. Hence, including transitivity constraints for a design that does not need them for its correctness proof might result in an increase of the verification time.

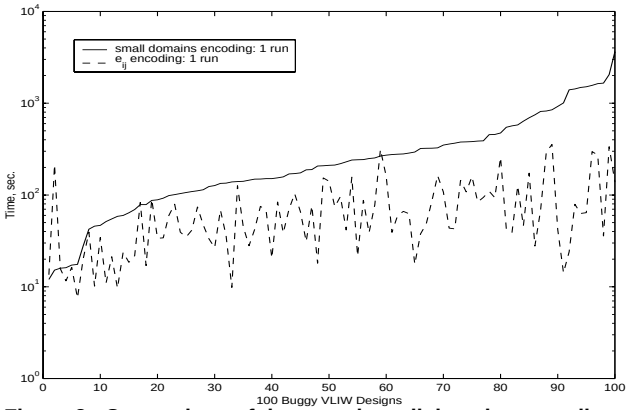

Figure 3: Comparison of the e_{ij} and small domains encodings on 100 buggy versions of 9VLIW-MC-BP, using Chaff. The benchmarks are sorted in ascending order of their times for the experiment with the small domains encoding.

We also compared the two encodings on correct designs that do require transitivity of equality for their correctness proofs—superscalar processors with out-of-order execution that can execute register-register and load instructions. Because instructions are dispatched when they do not have Write-After-Write (in addition to Write-After-Read and Read-After-Write) dependencies [23] on instructions that are earlier in the program order but are stalled due to data dependencies, transitivity of equality is required in proving the equality of the final states of the Register File reached after the Implementation and the Specification sides of the commutative correctness diagram.

Issue Width	G-Equation Encoding			
	e_{ij}		small domains	
	Primary Boolean Variables	CPU Time [sec]	Primary Boolean Variables	CPU Time [sec]
2	95	3.5	81	3.7
3	201	54	127	64
4	346	810	194	2,358
5	530	2,500	249	3,804

Table 2: Comparison of the e_{ij} and small domains encodings on correct out-of-order superscalar microprocessors that do require transitivity of equality for their correctness proofs.

While the small domains encoding introduced fewer Boolean variables—less than half of those required by the e_{ij} encoding for the 5-wide design—it resulted in longer CPU times. Chaff could not prove the unsatisfiability of the CNF formula for the 6-wide superscalar processor with either encoding in less than 24 hours of CPU time—a direction for future work.

The efficiency of the e_{ij} encoding can be explained by the impact of g-equations on the instruction flow, and hence on the correctness formula. Such equations determine forwarding and stalling conditions, based on equality comparisons of register identifiers, as well as instruction squashing conditions for correcting branch mispredictions, based on equality comparisons of actual and predicted branch targets. Therefore, g-equations affect

the execution of many instructions. A single Boolean variable, introduced in the e_{ij} encoding, naturally fits the purpose of accounting for both cases—that the equality comparison is either **true** or **false**. Transitivity of equality is never violated—as soon as two e_{ij} variables in a triangle become **true** then the third e_{ij} variable in that triangle immediately becomes **true**, due to the imposed transitivity constraints and the effect of the unit clause rule in SAT-checkers, and this is immediately extended to any cycle of e_{ij} variables [9]—which avoids wasteful exploration of infeasible portions of the search space.

On the other hand, the small domains encoding enumerates all mappings of g-term domain variables to a sufficient set of distinct constants, thus introducing more information than actually required to solve the problem. Now, an auxiliary Boolean variable f_{ij} is introduced in place of each primary e_{ij} Boolean variable from the previous encoding, such that f_{ij} represents the value of a Boolean formula enumerating the cases when g-terms i and j will evaluate to the same common constant. Therefore, f_{ij} depends on the indexing Boolean variables x_{il} that encode the mapping of g-term i to its set of possible constants, and on the indexing Boolean variables x_{jk} that encode the mapping of g-term j to its set of possible constants. Note that the indexing variables x_{il} will affect the value of each f_{im} auxiliary variable that encodes the equality between g-term i and some g-term m. If a SAT-checker assigns values to f_{ij} variables before all their supporting indexing variables, then the f_{ij} values might violate transitivity of equality. Furthermore, it might take a while before enough indexing variables get assigned in order to detect the violation and to correct it by backtracking. The work done in the meantime will be wasted. On the other hand, if all supporting indexing variables get assigned before the f_{ij} variables that they affect, then those f_{ij} variables will flip every time when a single indexing variable in their support flips. Note that each legal assignment to e_{ij} variables is a legal assignment to f_{ij} variables, except that now it can be justified with many possible assignments to the indexing variables. Hence, multiple branches in the formula will be revisited for what will be just one visit with the e_{ij} encoding. As a result, the small domains encoding is less efficient than the e_{ij} encoding.

In a different application—encoding constraint satisfaction problems as SAT instances—Hoos [24] similarly found that better performance is achieved with an encoding that introduces more variables but results in conceptually simpler search spaces.

7 Benefits of Conservative Approximations and Positive Equality

Conservative approximations, such as manually inserted translation boxes (dummy UFs or UPs with one input) [56] or automatically abstracted memories [57] have the potential to speed up the verification of correct designs, but might result in false negatives that will require manual user intervention and analysis. Not exploiting such optimizations in the verification of 9VLIW-MC-BP-EX resulted in CPU time of 2,542 seconds with monolithic evaluation of the correctness criterion and the e_{ij} encoding, compared to 1,513 seconds with the optimizations. However, exploiting structural variations in only one run—combining early reductions of p-equations and Ackermann constraints for eliminating UPs—resulted in CPU time of 1,964 seconds. This is a negligible overhead, compared with the burden of manual analysis necessary to identify potential false negatives that might result when using these optimizations.

We then evaluated the benefits of exploiting Positive Equality, given the extremely efficient SAT-checker Chaff. This was implemented by introducing an e_{ij} Boolean variable for the equality comparison of two distinct p-term domain variables as done originally by Goel, *et al.* [18], instead of treating these p-terms as different. We started with a buggy version of 1×DLX-C: the bug was detected in 0.02 seconds with Positive Equality, compared to 20 seconds without. Verifying the correct 1×DLX-C took 0.17 seconds with Positive Equality, compared to 3,111 seconds without. The bug in an erroneous version of 2×DLX-CC-MC-EX-BP was detected in 1.6 seconds with Positive Equality, compared with 661 seconds without. The correct 2×DLX-CC-MC-EX-BP was verified in 40 seconds with Positive Equality, consuming 36 MB of memory, but ran out of memory after 77,668 seconds without exploiting Positive Equality. Finally, a bug in an incorrect version of 9VLIW-MC-BP was detected in 173 seconds using 96 MB, compared to running out of memory after 6,351 seconds without Positive Equality. Therefore, exploiting Positive Equality is still the major reason for our success in formally verifying complex microprocessors.

8 Conclusions

We found the SAT-checker Chaff [38] to be the most efficient means for evaluating Boolean formulas generated in the formal verification of both correct and buggy microprocessors, dramatically outperforming 27 SAT-checkers, 2 ATPG tools, and 2 decision diagrams—BDDs [7] and BEDs [61]. Reassessing various optimizations that can be applied when producing the Boolean formula for the microprocessor correctness, we conclude that the single most important step is exploiting Positive Equality [8]. Without it, Chaff would not have scaled for realistic superscalar and VLIW microprocessors with exceptions, multicycle functional units, branch prediction, and other speculative features.

Exploiting the e_{ij} encoding [18] of g-equations resulted in a speedup of a factor of 4 for our most complex VLIW benchmarks compared to the small domains encoding [40] when verifying correct designs, and consistently performed better on buggy versions. Although the e_{ij} encoding results in more than twice as many primary Boolean variables, its efficiency can be explained with the conceptual simplicity of the resulting search space—with each e_{ij} Boolean variable naturally encoding the equality between a pair of g-term domain variables. Transitivity of equality is never violated, which avoids wasteful exploration of infeasible portions of the search space. In contrast, the small domains encoding enumerates all mappings of g-term domain variables to a sufficient set of distinct constants, thus introducing more information than actually required to solve the problem. This results in revisiting portions of the search space for what would be just one visit with the e_{ij} encoding. Transitivity of equality is not guaranteed to be always satisfied, also allowing wasteful work.

Conservative approximations, such as automatic abstraction of memories [57] and manually-inserted translation boxes [56], are not as essential to the fast verification of correct VLIW and dual-issue superscalar processors when using Chaff as these optimizations were when using BDDs—previously the most efficient SAT procedure for correct designs.

Structural variations in generating the Boolean correctness formulas—early reductions of p-equations and using Ackermann constraints for eliminating uninterpreted predicates—as well as parameter variations for Chaff can help to somewhat accelerate the SAT checking, although no single variation performs best.

Applying algebraic simplifications [35] to the CNF formulas resulting from realistic microprocessors is impractical, due to the large number of clauses—hundreds of thousands.

To conclude, we showed that Chaff can easily handle very hard and big CNF formulas, produced in the formal verification of microprocessors without applying conservative transformations that were previously needed in BDD-based evaluations but have the potential to result in false negatives and to take extensive human effort to analyze. We identified the optimizations that do help increase the performance of Chaff on realistic dual-issue superscalar and VLIW designs—Positive Equality, combined with the e_{ij} encoding, and possibly with structural/parameter variations in multiple parallel runs. Our study will increase the productivity of microprocessor design engineers and shorten the

time-to-market for VLIW and DSP architectures that constitute a significant portion of the microprocessor market [53]. The benchmarks used in this paper are available as [58].

Acknowledgments

We thank M. Moskewicz for providing us with Chaff and for fine-tuning it on our benchmarks.

References

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

[2] L. Baptista, and J.P. Marques-Silva, "Using Randomization and Learning to Solve Hard Real-World Instances of Satisfiability,"[2] *Principles and Practice of Constraint Programming (CP '00)*, September 2000.

[3] R.J. Bayardo, Jr., and R. Schrag, "Using CSP look-back techniques to solve real world SAT instances," *14th National Conference on Artificial Intelligence (AAAI '97)*, July 1997, pp. 203-208.

[4] BED Package[3], version 2.5, October 2000.

[5] F. Brglez, and H. Fujiwara, "A Neutral Netlist of 10 Combinational Benchmark Circuits," *International Symposium on Circuits and Systems (ISCAS '85)*, 1985.

[6] F. Brglez, D. Bryan, and K. Kozminski, "Combinational Profiles of Sequential Benchmark Circuits," *International Symposium on Circuits and Systems (ISCAS '89)*, 1989.

[7] R.E. Bryant, "Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams," ACM Computing Surveys, Vol. 24, No. 3 (September 1992), pp. 293-318.

[8] R.E. Bryant, S. German, and M.N. Velev, "Processor Verification Using Efficient Reductions of the Logic of Uninterpreted Functions to Propositional Logic,"[4] ACM Transactions on Computational Logic (TOCL), Vol. 2, No. 1 (January 2001).

[9] R.E. Bryant, and M.N. Velev, "Boolean Satisfiability with Transitivity Constraints,"[4] *Computer-Aided Verification (CAV '00)*, E.A. Emerson and A.P. Sistla, *eds.*, LNCS 1855, Springer-Verlag, July 2000, pp. 86-98.

[10] 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.

[11] P. Chatalic, and L. Simon, "Multi-Resolution on Compressed Sets of Clauses," *12th International Conference on Tools with Artificial Intelligence (ICTAI '00)*, November 2000, pp. 2-10.

[12] CGRASP, http://vinci.inesc.pt/~lgs/cgrasp.

[13] J.M. Crawford, and L.D. Auton, "Experimental Results on the Crossover Point in Random 3SAT," Frontiers in Problem Solving: Phase Transitions and Complexity, Artificial Intelligence, T. Hogg, B. A. Huberman and C. Williams, *eds.*, Vol. 81, Nos. 1-2 (March 1996), pp. 31-57.

[14] CUDD-2.3.0, http://vlsi.colorado.edu/~fabio.

[15] O. Dubois, "Can a Very Simple Algorithm be Efficient for SAT?", ftp://dimacs.rutgers.edu/pub/challenge/satisfiability/contributed/dubois/.

[16] J.W. Freeman, "Improvements to Propositional Satisfiability Search Algorithms," Ph.D. thesis, Department of Computer and Information Science, University of Pennsylvania, 1995.

[17] GRASP, http://vinci.inesc.pt/~jpms/grasp.

[18] A. Goel, K. Sajid, H. Zhou, A. Aziz, and V. Singhal, "BDD Based Procedures for a Theory of Equality with Uninterpreted Functions," *Computer-Aided Verification (CAV '98)*, A.J. Hu and M.Y. Vardi, *eds.*, LNCS 1427, Springer-Verlag, June 1998, pp. 244-255.

[19] C.P. Gomes, B. Selman, N. Crator, and H.A. Kautz, "Heavy-Tailed Phenomena in Satisfiability and Constraint Satisfaction Problems", Journal of Automated Reasoning, Vol. 24, Nos. 1-2 (February 2000), pp. 67-100.

[20] J.F. Groote, and J.P. Warners, "The propositional formula checker Heer-Hugo," J. of Automated Reasoning, Vol. 24, Nos. 1-2 (February 2000).

[21] I. Hamzaoglu, and J.H. Patel, "New Techniques for Deterministic Test Pattern Generation," Journal of Electronic Testing: Theory and Applications, Vol. 15, Nos. 1-2 (August 1999), pp. 63-73.

[22] J.E. Harlow III, and F. Brglez, "Design of Experiments for Evaluation of BDD Packages Using Controlled Circuit Mutations," *Formal Methods in Computer-Aided Design (FMCAD '98)*, G. Gopalakrishnan and P. Windley, *eds.*, LNCS 1522, Springer-Verlag, November 1998, pp. 64-81.

[23] J.L. Hennessy, and D.A. Patterson, *Computer Architecture: A Quantitative Approach*, 2nd edition, Morgan Kaufmann Publishers, San Francisco, CA, 1996.

[24] H.H. Hoos, "SAT-Encodings, Search Space Structure, and Local Search Performance," *International Joint Conference on Artificial Intelligence (IJCAI '99)*, August 1999, pp. 296-302.

[25] *IA-64 Application Developer's Architecture Guide*,[5] Intel Corporation, May 1999.

[26] K. Iwama, H. Abeta, and E. Miyano, "Random Generation of Satisfiable and Unsatisfiable CNF Predicates," Information Processing 92, Vol. 1: Algorithms, Software, Architecture, J. Van Leeuwen, *ed.*, Elsevier Science Publishers B.V., 1992, pp. 322-328.

[27] K. Iwama, and K. Hino, "Random Generation of Test Instances for Logic Optimizers," *31st Design Automation Conference (DAC '94)*, June 1994.

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

[29] P. Kalla, Z. Zeng, M.J. Ciesielski, and C. Huang, "A BDD-Based Satisfiability Infrastructure Using the Unate Recursive Paradigm," *Design, Automation and Test in Europe (DATE '00)*, March 2000, pp. 232 -236.

[30] C.M. Li, and Anbulagan, "Heurisitics Based on Unit Propoagation for Satisfiability Problems," *International Joint Conference on Artificial Intelligence (IJCAI '97)*, August 1997, pp. 366-371.

[31] C.M. Li, "Integrating Equivalency Reasoning into Davis-Putnam Procedure," *17th National Conference on Artificial Intelligence (AAAI '00)*, July – August 2000, pp. 291-296.

[32] J.P. Marques-Silva, and K.A. Sakallah, "GRASP: A Search Algorithm for Propositional Satisfiability," IEEE Transactions on Computers, Vol. 48, No. 5 (May 1999), pp. 506-521.

[33] J.P. Marques-Silva, "The Impact of Branching Heuristics in Propositional Satisfiability Algorithms,"[2] *9th Portuguese Conference on Artificial Intelligence (EPIA)*, September 1999.

[34] J.P. Marques-Silva, and L.G. e Silva, "Algorithms for Satisfiability in Combinational Circuits Based on Backtrack Search and Recursive Learning,"[2] *12th Symposium on Integrated Circuits and Systems Design (SBCCI '99)*, September – October 1999, pp. 192-195.

[35] J.P. Marques-Silva, "Algebraic Simplification Techniques for Propositional Satisfiability,"[2] *Principles and Practice of Constraint Programming (CP '00)*, September 2000, pp. 537-542.

[36] D. Mitchell, B. Selman, and H. Levesque, "Hard and Easy Distributions of SAT Problems," *10th National Conference on Artificial Intelligence (AAAI '92)*, July 1992, pp. 459-465.

[37] 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.

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

[39] D.A. Plaisted, A. Biere, and Y. Zhu, "A Satisfiability Procedure for Quantified Boolean Formulae," submitted for publication, 2000.

[40] A. Pnueli, Y. Rodeh, O. Shtrichman, and M. Siegel, "Deciding Equality Formulas by Small-Domain Instantiations," *Computer-Aided Verification (CAV '99)*, N. Halbwachs and D. Peled, *eds.*, LNCS 1633, Springer-Verlag, June 1999, pp. 455-469.

[41] S.D. Prestwich, "Stochastic Local Search in Constrained Spaces," *Practical Application of Constraint Technology and Logic Programming (PACLP '00)*, April 2000, pp. 27-39.

[42] J. Rintanen, "Improvements to the Evaluation of Quantified Boolean Formulae," *International Joint Conference on Artificial Intelligence (IJCAI '99)*, August 1999, pp. 1192-1197.

[43] R. Rudell, "Dynamic Variable Ordering for Ordered Binary Decision Diagrams," *International Conference on Computer-Aided Design (ICCAD '93)*, November 1993, pp. 42-47.

[44] SATO.3.2.1, http://www.cs.uiowa.edu/~hzhang/sato.

[45] SATLIB—Solvers, http://www.satlib.org/solvers.html.

[46] B. Selman, H. Kautz, B. Cohen, "Local Search Strategies for Satisfiability Testing," DIMACS Series in Discrete Mathematics and Theoretical Computer Science, Vol. 26 (1996), pp. 521-532.

[47] B. Selman, and H. Kautz, "Domain-Independent Extensions to GSAT: Solving Large Structured Satisfiability Problems," *International Joint Conference on Artificial Intelligence (IJCAI '93)*, August – September 1993, pp. 290-295.

[48] Y. Shang, and B.W. Wah, "A Descrete Lagrangian-Based Global-Search Method for Solving Satisfiability Problems," Journal of Global Optimization, Vol. 12, No. 1, (January 1998), pp. 61-99.

[49] H. Sharangpani, "Intel Itanium Processor Michroarchitecture Overview,"[5] *Microprocessor Forum*, October 1999.

[50] G. Stålmarck, "A System for Determining Propositional Logic Theorems by Applying Values and Rules to Triplets that are Generated from a Formula," Swedish Patent No. 467 076 (approved 1992), U.S. Patent No. 5 276 897 (1994), European Patent No. 0403 454 (1995), 1989.

[51] Stanford Validity Checker (SVC), http://sprout.Stanford.EDU/SVC.

[52] P. Tafertshofer, A. Ganz, and K.J. Antreich, "GRAINE—An Implication GRaph-bAsed engINE for Fast Implication, Justification, and Propagation," IEEE Transactions on CAD, Vol. 19, No. 8 (August 2000).

[53] D. Tennenhouse, "Proactive Computing," Communications of the ACM, Vol. 43, No. 5 (May 2000), pp. 43-50.

[54] D. Van Campenhout, T. Mudge, and J.P. Hayes, "Collection and Analysis of Microprocessor Design Errors," IEEE Design & Test of Computers, Vol. 17, No. 4 (October – December 2000), pp. 51-60.

[55] M.N. Velev, and R.E. Bryant, "Superscalar Processor Verification Using Efficient Reductions of the Logic of Equality with Uninterpreted Functions to Propositional Logic,"[4] *Correct Hardware Design and Verification Methods (CHARME '99)*, L. Pierre and T. Kropf, eds., LNCS 1703, Springer-Verlag, September 1999, pp. 37-53.

[56] M.N. Velev, and R.E. Bryant, "Formal Verification of Superscalar Microprocessors with Multicycle Functional Units, Exceptions, and Branch Prediction,"[4] *37th Design Automation Conference (DAC '00)*, June 2000.

[57] M.N. Velev, "Formal Verification of VLIW Microprocessors with Speculative Execution,"[4] *Computer-Aided Verification (CAV '00)*, E.A. Emerson and A.P. Sistla, *eds.*, LNCS 1855, Springer-Verlag, July 2000.

[58] M.N. Velev, Benchmark suites[4] SSS-SAT.1.0, VLIW-SAT.1.0, FVP-UNSAT.1.0, and FVP-UNSAT.2.0, October 2000.

[59] M.N. Velev, "Automatic Abstraction of Memories in the Formal Verification of Superscalar Microprocessors,"[4] *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.

[60] P.F. Williams, A. Biere, E.M. Clarke, A. Gupta, "Combining Decision Diagrams and SAT Procedures for Efficient Symbolic Model Checking,"[3] *Computer-Aided Verification (CAV '00)*, E.A. Emerson and A.P. Sistla, *eds.*, LNCS 1855, Springer-Verlag, July 2000, pp. 124-138.

[61] P.F. Williams, "Formal Verification Based on Boolean Expression Diagrams,"[3] Ph.D. thesis, Department of Information Technology, Technical University of Denmark, Lyngby, Denmark, August 2000.

[62] Z. Wu, and B.W. Wah, "Solving Hard Satisfiability Problems: A Unified Algorithm Based on Discrete Lagrange Multipliers," *11th IEEE International Conference on Tools with Artificial Intelligence (ICTAI '99)*, November 1999, pp. 210-217.

[63] H. Zhang, "SATO: An Efficient Propositional Prover," *International Conference on Automated Deduction (CADE '97)*, LNAI 1249, Springer-Verlag, 1997, pp. 272-275.

2. http://vinci.inesc.pt/~jpms
3. http://www.it-c.dk/research/bed
4. http://www.ece.cmu.edu/~mvelev
5. http://developer.intel.com/design/ia-64/architecture.htm