Effective Use of Boolean Satisfiability Procedures in the Formal Verification of Superscalar and VLIW Microprocessors*

Miroslav N. Velev[1] and Randal E. Bryant[2]

[1] *School of Electrical and Computer Engineering*
Georgia Institute of Technology, Atlanta, GA 30332, U.S.A.,
mvelev@ece.gatech.edu

[2] *Computer Science Department*
Carnegie Mellon University, Pittsburgh, PA 15213, U.S.A.
Randy.Bryant@cs.cmu.edu

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. The microprocessors are described in a high-level hardware description language, based on the logic of Equality with Uninterpreted Functions and Memories (EUFM). The formal verification is done with Burch and Dill's correctness criterion, using flushing to map the state of the implementation processor to the state of the specification. The EUFM correctness formula is translated to an equivalent Boolean formula by exploiting the property of Positive Equality, and using the automatic tool `EVC`. We identify the SAT-checkers `Chaff` and `BerkMin` as significantly outperforming the rest of the SAT tools when evaluating the Boolean correctness formulas. We examine ways to enhance the performance of `Chaff` and `BerkMin` by variations when generating the Boolean formulas. We reassess optimizations we developed earlier to speed up the formal verification.

1. Introduction

In the past few years, SAT-checkers made dramatic improvements in both their speed and capacity. We compare 31 of them with decision diagrams—BDDs (Bryant, 1986, 1992) and BEDs (Williams, 2000)—as well as with ATPG tools (Hamzaoglu and Patel, 1999)(Tafertshofer, *et al.*, 2000), when used as Boolean

*This research was supported by the Semiconductor Research Corporation under contract 00-DC-684. A preliminary version of the paper was published as (Velev and Bryant, 2001a).

Satisfiability (SAT) procedures in the formal verification of microprocessors. The comparison is based on two benchmark suites, each consisting 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., ISCAS85 (Brglez and Fujiwara, 1985) and ISCAS89 (Brglez, *et al.*, 1989), which are collections of circuits that have nothing in common, each of our suites is based on a single design and hence provides 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 (Burch and Dill, 1994). That allows it either to be checked directly with a customized decision procedure, such as `SVC`[1], or to be translated to an equivalent Boolean formula (Velev and Bryant, 1999) 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` (Moskewicz, *et al.*, 2001; Zhang, *et al.*, 2001) and `BerkMin` (Goldberg and Novikov, 2002) as the most efficient SAT-checkers for the second verification strategy. `Chaff` and `BerkMin` significantly outperform BDDs and the SAT-checker `DLM-3` (Shang and Wah, 1998), the previous most efficient SAT procedures for, respectively, correct and buggy processors. We reevaluate optimizations we developed earlier to enhance the performance of BDDs and `DLM-3`, and conclude that many of them are no longer crucial on the same benchmark suites. This study allows us to eliminate conservative approximations that can lead to false negative results—a source of annoyance for users.

Our initial research was on developing an Efficient Memory Model (EMM) for abstracting memory arrays in symbolic ternary simulation at the bit level (Velev and Bryant, 1998a). The ternary value X, representing a don't-care condition and encoded symbolically, allows us to dramatically reduce the number of symbolic vectors that need to be simulated. Additionally, it gives us a way to express ambiguity in signal values—a property that we exploited to model violations in the setup and hold time requirements for memory inputs, as well as to represent the uncertainty of memory output delays that can range between a minimum and a maximum value.

The EMM dynamically introduces consistent initial state for accessed symbolic addresses, and that allowed us to use read-only EMMs to abstract bit-level combinational functional units (Velev and Bryant, 1998b). However, the presence of feedback loops in pipelined processors (e.g., as introduced by the forwarding logic or the Register File) resulted in impossible to satisfy variable-ordering constraints when BDDs were used to evaluate the Boolean correctness formulas. By restricting the style for defining processors, while still able to model the same features, we obtained correctness formulas where most of the word-level values appear only in positive (not negated) equality comparisons. This structure of the formulas allowed us to treat such word-level values as distinct constants,

[1]`SVC` (Stanford Validity Checker) is available from: `http://sprout.Stanford.EDU/SVC`.

thus dramatically pruning the solution space, while still performing exhaustive formal proofs. We called this property Positive Equality (Bryant, German, and Velev, 2001), and showed that it results in orders of magnitude speedup (Velev and Bryant, 1999).

Next, we demonstrated that the same modeling techniques can be used to define and formally verify single-issue pipelined, and dual-issue superscalar processors that implement exceptions, branch prediction, and multicycle functional units (Velev and Bryant, 2000). A more complex VLIW processor—imitating the Intel® Itanium® (Sharangpani and Arora, 2000) in features such as predicated execution, advanced loads, and speculative register remapping—was then formally verified (Velev, 2000a). A method to automatically abstract memory arrays, whose correct operation is enforced by the interaction of forwarding and stalling logic, resulted in an order of magnitude speedup of the BDD-based evaluation of the correctness formula (Velev, 2001). A significant breakthrough occurred with the development of the SAT-checker `Chaff`, as reported in our earlier study (Velev and Bryant, 2001a). The current paper gives more details of that work, and presents additional experimental results.

The rest of the paper is organized as follows. Section 2 presents the background of high-level modeling and formal verification of microprocessors. Section 3 describes our microprocessor benchmarks used in the experiments. Section 4 lists the compared SAT procedures, explains the translation of the Boolean correctness formulas to CNF format, and presents results showing that only two SAT tools—`Chaff` and `BerkMin`—scale for our complex benchmarks. Then, we explore ways to efficiently use these two SAT-checkers. Section 5 studies the impact of variations when generating and evaluating the Boolean correctness formulas. Section 6 compares two ways to encode word-level equality comparisons in the correctness formulas. Section 7 evaluates the benefits of decomposing the correctness criterion. Section 8 studies the usefulness of conservative approximations and Positive Equality. Section 9 concludes the paper, and prioritizes the optimizations that help `Chaff` and `BerkMin`.

2. Background

The formal verification is done by correspondence checking—comparison of the single-issue pipelined, or superscalar, or VLIW implementation processor against a non-pipelined specification processor, by using Burch and Dill's flushing technique (1994). The correctness criterion is expressed as a formula in the logic of Equality with Uninterpreted Functions and Memories (EUFM), also proposed by Burch and Dill (1994), and states that all architectural state elements in the processor should be updated in synchrony by either 0, or 1, or up to k instructions after each clock cycle, where k is the maximum number of instructions that the design can fetch in a clock cycle. The correctness formula is then translated to an equivalent Boolean formula by the automatic tool `EVC` (Velev and Bryant, 2001b) that exploits the properties of Positive Equality (Bryant, German, and

Velev, 2001), the e_{ij} encoding (Goel, *et al.*, 1998), and a number of conservative approximations. The resulting Boolean correctness formula should be a tautology (or, equivalently, its complement should be unsatisfiable) in order for the processor to be correct, and can be evaluated by any SAT procedure.

The syntax of EUFM (Burch and Dill, 1994) includes terms and formulas—see Figure 1.

$$\begin{array}{rcl}
\mathit{term} & ::= & \mathit{ITE}(\mathit{formula}, \mathit{term}, \mathit{term}) \\
 & & \mid \mathit{uninterpreted\text{-}function}(\mathit{term}, \ldots, \mathit{term}) \\
 & & \mid \mathit{read}(\mathit{term}, \mathit{term}) \\
 & & \mid \mathit{write}(\mathit{term}, \mathit{term}, \mathit{term}) \\
 & & \\
\mathit{formula} & ::= & \textbf{true} \mid \textbf{false} \mid (\mathit{term} = \mathit{term}) \\
 & & \mid (\mathit{formula} \wedge \mathit{formula}) \mid (\mathit{formula} \vee \mathit{formula}) \mid \neg \mathit{formula} \\
 & & \mid \mathit{ITE}(\mathit{formula}, \mathit{formula}, \mathit{formula}) \\
 & & \mid \mathit{uninterpreted\text{-}predicate}(\mathit{term}, \ldots, \mathit{term})
\end{array}$$

Figure 1: Syntax of the logic of Equality with Uninterpreted Functions and Memories.

Terms are used to abstract word-level values of data, register identifiers, memory addresses, as well as the entire states of memory arrays. A term can be an Uninterpreted Function (UF) applied to a list of argument terms; a term variable (that can be viewed as an UF symbol without arguments); or an *ITE* operator selecting between two argument terms based on a controlling formula, such that $\mathit{ITE}(\mathit{formula}, \mathit{term}_1, \mathit{term}_2)$ will evaluate to term_1 when $\mathit{formula} = \textbf{true}$ and to term_2 when $\mathit{formula} = \textbf{false}$. The syntax for terms can be extended to model memories by means of the interpreted functions *read* and *write* (Burch and Dill, 1994)(Velev, 2001). Function *read* takes two terms, serving as memory state and address, respectively, and returns a term for the data at that address. Function *write* takes three terms—memory state, address, and data—and returns a term for the new memory state after the update. The two functions satisfy the forwarding property of the memory semantics—a *read* returns the data written by the last *write*, if their addresses are equal, or the data from the previous memory state otherwise. The initial state of a memory is abstracted with a term variable.

Formulas are used 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 to a list of argument terms; a propositional variable (that can be viewed as an UP symbol without arguments); 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, conjuncted, or disjuncted. 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*—equal combinations of expressions at the inputs of the UF (or UP) produce equal output values. 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 processor. We assume that the functional units and memories are formally verified separately.

2.1. Example High-Level Pipelined Microprocessor

The above abstraction techniques are illustrated with the 3-stage pipelined processor shown in Figure 2. The 3 stages are Instruction Fetch and Decode (IFD), Execute (EX), and Write-Back (WB). For illustration purposes, the processor can execute register-register instructions only. Uninterpreted functions ALU, and +4 are used to abstract, respectively, the ALU, and the adder for incrementing the Program Counter (PC). The Register File is abstracted with functions *read* and *write*, such that signal WB_RegWrite is used as the condition for performing *write*s. In other words, if *wb_regwrite* is a symbolic expression for the value of that signal, then the new Register File state will be *ITE*(*wb_regwrite*, *write*(*prev_state*, *wb_destreg*, *wb_result*), *prev_state*), where *prev_state* is an expression for the previous Register File state; *wb_destreg* and *wb_result* are expressions for the values of signals WB_DestReg and WB_Result, respectively, and serve as the address and data arguments of the *write* operation. All the word-level values—register identifiers, opcodes, data operands, ALU result, and PC—are modeled as terms. The opcode, Op, specifies the operation to be performed by the ALU. We will assume that the processor does not execute self-modifying code, which allows us to represent the (read-only) Instruction Memory, InstrMemory, as a collection of UFs and UPs that take the PC as argument and abstract the fetching and decoding of the corresponding field of a new instruction. The processor has forwarding logic, situated in EX, only for the second data operand. Data hazards for the first data operand are avoided by the stalling logic in IFD, so that the dependent instruction is delayed in IFD until the result it needs is written back to the Register File. The Register File is assumed to be write-before-read, which is modeled by synchronizing its updates with a phase clock that precedes the phase clock controlling the updates of pipeline latch IFD_EX. These modeling details are expressed in a high-level hardware description language that is accepted by the term-level symbolic simulator `TLSim` (Velev and Bryant, 2001b).

The correct behavior is defined by a non-pipelined specification processor that is built from the same UFs, UPs, and architectural state elements (PC and Register File in the example) as the pipelined implementation—see Figure 3. This design is the Instruction Set Architecture (ISA) put together in a single-

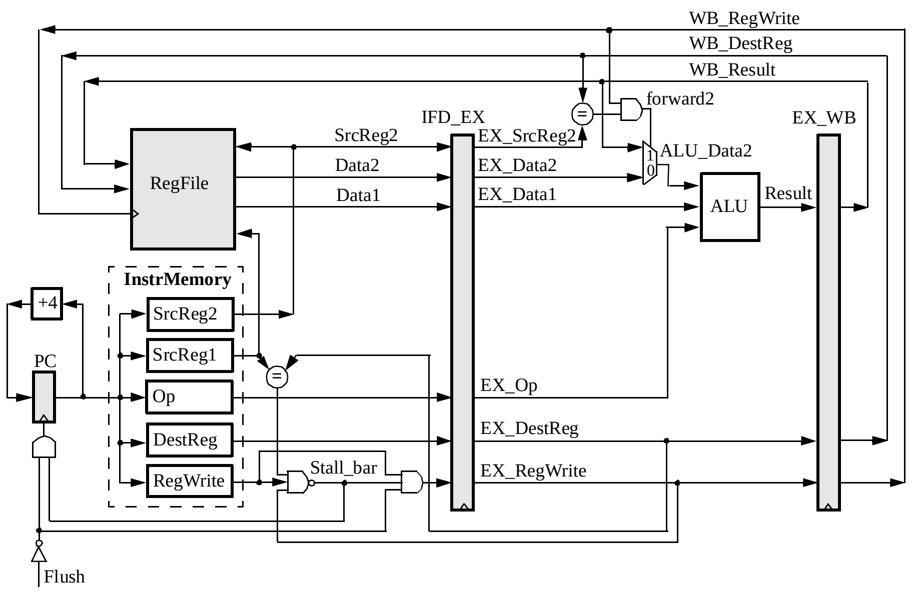


Figure 2: Block diagram of a 3-stage pipelined processor.

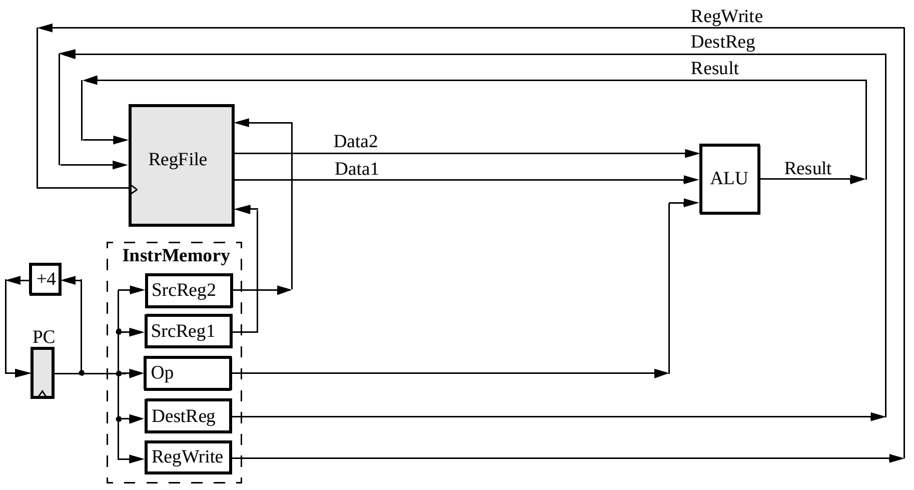


Figure 3: Block diagram of the non-pipelined specification processor.

cycle model. The processor fetches, executes, and completes one new instruction on every clock cycle. Because of its simplicity, it is easy to define correctly. Furthermore, it will be extremely easy to formally verify—the action of every instruction type can be checked directly against its expected behavior, as defined in the ISA.

Note that by applying all of the abstractions, we get a much more general pipelined processor than the original, such that the functional units are only functionally consistent, but do not satisfy any other properties of their original implementations. However, proving the correctness of such abstract processors is much easier. If the pipeline is correct, it will work properly for any functionally consistent implementation of the logic that is abstracted with UFs or UPs.

2.2. Overview of Translating the EUFM Correctness Formula to Equivalent Boolean Formula

In order to translate the EUFM correctness formula to an equivalent Boolean formula, we need to eliminate the UFs and UPs in a way that their property of functional consistency is enforced, as well as to encode the term-level equality comparisons with Boolean formulas such that the property of transitivity of equality is satisfied.

Two possible ways to eliminate UFs and UPs, while enforcing their property of functional consistency, are Ackermann constraints (Ackermann, 1954), and nested *ITE*s (Velev and Bryant, 1999)(Bryant, German, and Velev, 2001). The Ackermann scheme replaces each UF (UP) application in the EUFM formula F with a new term variable (propositional variable), and then adds external constraints for functional consistency. For example, the UF application $f(a_1, b_1)$ will be replaced by a new term variable c_1, another application of the same UF, $f(a_2, b_2)$, will be replaced by a new term variable c_2. Then, the resulting EUFM formula F' will be extended as

$$[(a_1 = a_2) \wedge (b_1 = b_2) \Rightarrow (c_1 = c_2)] \;\Rightarrow\; F'.$$

In the nested-*ITE* scheme, the first application of the UF above will still be replaced by a new term variable c_1. However, the second will be replaced by

$$ITE((a_2 = a_1) \wedge (b_2 = b_1), c_1, c_2),$$

where c_2 is a new term variable. A third, $f(a_3, b_3)$, 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 term variable, and so on. UPs are eliminated similarly by using new Boolean variables, instead of new term variables.

Positive Equality allows the identification of two types of terms in the structure of an EUFM formula—those that appear in only positive equations (*p-equations*) and are so called *p-terms* (for positive terms), and those that appear in both positive and negative equations and are so called *g-terms* (for general terms). A negative equation is one that 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 term variable with itself evaluates to **true**, that of a p-term variable with a syntactically distinct term variable evaluates to **false**, and that of a g-term variable with a syntactically distinct g-term variable could be either **true** or **false** and can be encoded either with a Boolean variable (Goel, *et al.*, 1998) or with a Boolean function (Pnueli, *et al.*, 1999)—details of these encodings are presented in Section 6. We call the equality comparison of two syntactically distinct g-term variables a *g-equation*. Evaluating the EUFM correctness formula under a maximally diverse interpretation results in dramatic simplifications, and thus in orders of magnitude speedup.

As a result, the EUFM correctness formula is translated to an equivalent Boolean formula that has to be a tautology in order for the EUFM correctness formula to be valid. The Boolean formula can be evaluated with any SAT procedure—see Section 4.

3. Microprocessor Benchmarks

We base our comparison of SAT procedures on a set of high-level microprocessors:

- 1×DLX-C (Velev and Bryant, 1999): a single-issue 5-stage pipelined DLX, as described by Hennessy and Patterson (2002);
- 2×DLX-CC (Velev and Bryant, 1999): a dual-issue superscalar DLX, which is an extended version of a processor verified by Burch (1996);
- 2×DLX-CC-MC-EX-BP (Velev and Bryant, 2000): a version of 2×DLX-CC with multicycle functional units, exceptions, and branch prediction;
- 9VLIW-MC-BP (Velev, 2000a): a 9-wide VLIW processor that imitates the Intel® Itanium® (Intel, 1999)(Sharangpani and Arora, 2000) in speculative features such as predicated execution, speculative register remapping, advanced loads, and branch prediction.

The single-issue pipelined processor, 1×DLX-C, has five stages: Fetch, Decode, Execute, Memory, and Write-Back. The design can execute seven instruction types: register-register ALU instructions, register-immediate ALU instructions, loads, stores, branches, jumps, and nops. A nop only increments the Program Counter (PC), but does not modify other architectural state elements. Branches do not have delay slots, i.e., an instruction that immediately follows a branch is completed only if the branch is not taken. The processor is biased for branch-not-taken, and continues to fetch instructions that sequentially follow a branch (i.e., instructions from the path when the branch is not taken) until the branch is resolved in the Execute stage. Then, if the branch is taken, as can be checked in the Memory stage, the three speculatively fetched sequential instructions that are in the Fetch, Decode, and Execute stages are squashed (canceled), and the PC is updated with the target of the branch. Read-After-Write hazards—due to pending updates of the Register File by instructions that are in the Memory and Write-Back stages, when a dependent instruction is in the Execute stage—are resolved by forwarding of the data values from the Memory and Write-Back stages to the inputs of the functional units in the Execute sage. However, the processor does not have a forwarding path from the output of the Data Memory in the Memory stage to the Execute stage in order to satisfy a data dependency when a load gets data from memory and that value is used in the Execute stage by the instruction immediately following the load. Although such a forwarding path is feasible, it will likely lengthen the clock cycle in order to allow a signal to propagate through the Data Memory, the forwarding logic, and then the ALU in the Execute stage, thus slowing all instructions only to satisfy a data dependency in the infrequent case of a load immediately followed by a dependent instruction. Instead, commercial pipelined processors and our design adopt an alternative solution that avoids such data hazards by stalling the dependent instruction in the Decode stage, when the load providing a data operand is in the Execute stage. That means that the dependent instruction stays in the Decode stage during the next clock cycle, while the load is allowed to advance to the Memory stage, and a bubble (a combination of control bits that will not modify any architectural state element) is inserted in the Execute stage. A cycle later, the dependent instruction is allowed to advance to the Execute stage, while the load will have gotten the data from the Memory stage and will be in the Write-Back stage, so that the data dependency can be satisfied by the forwarding path from the Write-Back to the Execute stage. This mechanism, preventing data hazards in the case of a load immediately followed by a dependent instruction, is called a *load interlock* (Hennessy and Patterson, 2002). A pipelined processor should be able to handle any combination of hazards that might occur between instructions in the pipeline stages. We assume that the processor does not execute self-modifying code, which allows us to model the (read-only) Instruction Memory in the Fetch stage and the Data Memory in the Memory stage as separate memories. Otherwise, the processor has to be extended with a mechanism to re-execute instructions that get modified by store instructions still in later pipeline stages; this mechanism is similar to the one for correcting branch mispredictions, e.g., as implemented in 2×DLX-CC-MC-EX-BP.

The dual-issue superscalar implementation, 2×DLX-CC, consists of two 1×DLX-C pipelines, and can fetch up to two sequential instructions per clock cycle. Now there are two load interlock conditions per instruction in the Decode stage, since each of these two instructions has to be checked for data dependencies on the two possible loads in the Execute stage. If the first instruction in Decode gets stalled, the second is also stalled. Additionally, the second instruction is stalled if it has a data dependency on the first. 2×DLX-CC-MC-EX-BP extends 2×DLX-CC with multicycle functional units, exceptions, and branch prediction. Each of the Instruction Memory, the two ALUs in the Execute stage, and the Data Memory can take multiple cycles to produce a result, and can raise an exception. The Fetch stage has an abstraction of a branch predictor that predicts both the direction (taken or not-taken) and the target of a newly fetched branch, but only the target of a newly fetched jump, since jumps are always taken. Based on these predictions, the PC is speculatively updated, such that when the actual branch/jump outcome is known in the Memory stage, special logic corrects mispredictions by squashing the speculatively fetched instructions.

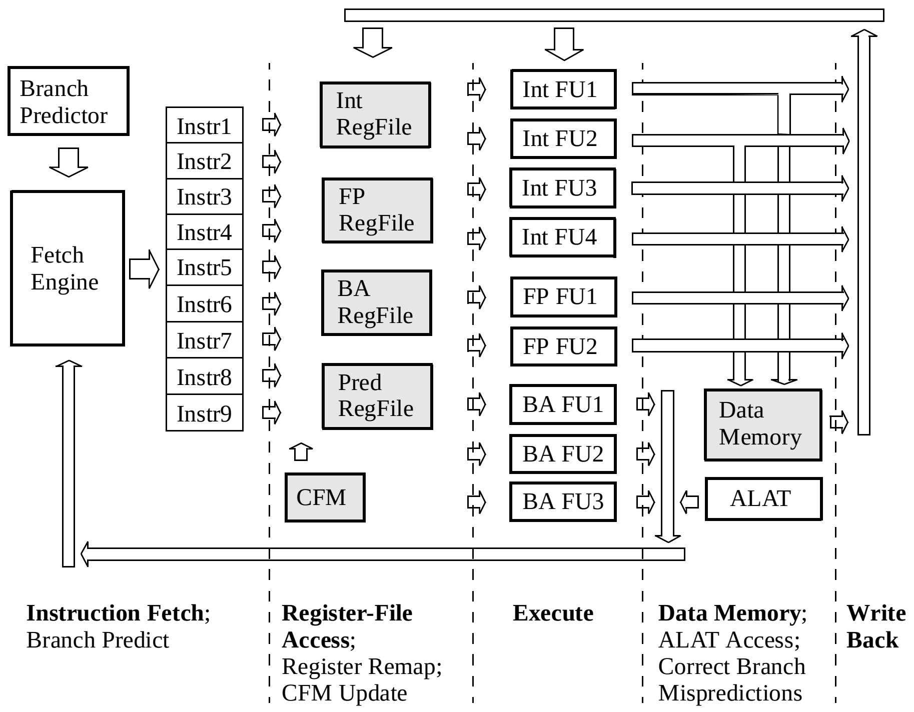


Figure 4: Block diagram of the VLIW architecture that was formally verified.

The VLIW benchmark, 9VLIW-MC-BP (see Figure 4), is far more complex than any other processor 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 Read-After-Write dependencies between any two of them. 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. Data values are stored in 4 register files: integer, floating-point, predicate, and branch-address. In addition to these 4 register files, the architectural state consists of a PC, a Data Memory, as well as two state elements from Intel's 64-bit architecture, IA-64 (Intel, 1999)(Sharangpani and Arora, 2000)—a Current Frame Marker (CFM) that is used for register remapping, and an Advanced Load Address Table (ALAT) that is used to implement advanced loads. Every instruction is predicated with a qualifying predicate register identifier, such that the result of that instruction affects architectural state only when the qualifying predicate register has value 1. 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 that also implements exceptions, was later designed as described in Section 7.

For both 2×DLX-CC-MC-EX-BP and 9VLIW-MC-BP, we created 100 incorrect versions by injecting bugs into the designs. The bugs were variants of actual errors made in the design of the correct versions, as well as variants of errors detected in Intel microprocessors (Bentley, 2001)(Jones, 2002), and in academic processors (Van Campenhout, *et al.*, 1998, 2000). The injected bugs included:

- Omitted inputs to logic gates. For example, a speculatively fetched instruction is not squashed when a preceding branch is mispredicted, or a load interlock does not account for all cases when a value loaded from memory will be used by a dependent instruction.
- Incorrect inputs to logic gates, functional units, or memories. For example, an input with the same name but a different index. Such bugs were detected in the formal verification of an IA-32 instruction-length decoder in an actual Intel processor, as discussed by Jones (2002, pages 85–86). Bentley (2001) similarly lists typos and cut-and-paste errors in a category of "Goof" bugs, detected in the Intel® Pentium® 4 microprocessor.
- Incorrect types of logic gates. For example, an AND gate instead of an OR gate, as was the case in an actual Intel processor bug described by Jones (2002, page 85).
- Lack of a mechanism to correct a speculative update of an architectural state element, if the speculation turns out to be wrong. For example, the PC is updated speculatively, based on a prediction for the direction and target of a newly fetched branch, but there is no logic to update the PC with the correct branch target if the prediction happens to be wrong. Similarly, when designing 9VLIW-MC-BP, a bug was inadvertently made in that the CFM could be updated speculatively by instructions along the predicted path after a branch, but there was no mechanism to restore the correct CFM state if the branch was mispredicted.

Hence, the variations introduced were not completely random, as done in other efforts to generate benchmark suites (Harlow and Brglez, 2001)(Iwama, *et al.*, 1992, 1994)(Mitchell, *et al.*, 1992), but reflected realistic scenarios for errors that can be made when designing high-level microprocessors. The bugs were spread over the entire designs, and occurred either as single or multiple errors.

4. Comparison of SAT Procedures

We evaluated 31 SAT-checkers. Nine of them were complete (i.e., could prove unsatisfiability), were based on the Davis-Putnam-Logemann-Loveland (DPLL) algorithm (Davis, *et al.*, 1962), and implemented learning:

- `SATO.3.2.1` (Zhang, 1997);
- `GRASP` (Marques-Silva and Sakallah, 1999)(Marques-Silva, 1999)[2], implementing nonchronological backtracking, was used both with a single strategy and in a mode with restarts, randomization, and recursive learning (Baptista and Marques-Silva, 2000);

[2]`GRASP` is available from: `http://vinci.inesc.pt/~jpms/grasp`.

- `CGRASP` (Marques-Silva and e Silva, 1999)[3], a version of GRASP that exploits structural information;
- `rel_sat.1.0`, and `rel_sat.2.1` (Bayardo and Schrag, 1997)[4];
- `rel_sat_rand.1.0` (Gomes, *et al.*, 2000)[4];
- `Chaff` (Moskewicz, *et al.*, 2001; Zhang, *et al.*, 2001)[5], implementing lazy Boolean constraint propagation, conflict-based relevance-limited learning, restarts, randomization, and decision heuristics guided by recent conflict clauses;
- `SIMO.2.0` (Copty, *et al.*, 2001)[6]; and
- `BerkMin` version 62 (Goldberg and Novikov, 2002), extending the ideas from `Chaff` with decision heuristics and database management procedures that attempt to satisfy the most recently deduced conflict clauses.

Eight SAT-checkers were also complete and based on the DPLL algorithm, but did not have learning:

- `satz`, and `satz.v213` (Li and Anbulagan, 1997)[4];
- `satz-rand.v4.6` (Gomes, *et al.*, 2000)[4];
- `eqsatz.v20` (Li, 2000);
- `posit` (Freeman, 1995)[4];
- `ntab` (Crawford and Auton, 1996)[4]; and
- `ASAT`, and `C-SAT` (Dubois, *et al.*, 1993).

Seven SAT-checkers were incomplete (i.e., could not prove unsatisfiability):

- `DLM-2`, and `DLM-3` (Shang and Wah, 1998), as well as `DLM-2000` (Wu and Wah, 1999), all 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 toward a global minimum, i.e., toward a satisfying assignment;
- `GSAT.v41` (Selman and Kautz, 1993)[4];
- `WalkSAT.v37` (Selman, *et al.*, 1996)[4];
- `CLS` (Prestwich, 2000); and
- `UnitWalk` (Hirsch and Kojevnikov, 2001)[7], based on local search guided by unit clause elimination.

[3]`CGRASP` is available from: `http://vinci.inesc.pt/~lgs/cgrasp`.

[4]The SAT-checker is available from: `http://www.satlib.org/solvers.html`.

[5]We used the version `mChaff` with parameter file `cherry_032301` (Moskewicz, 2001).

[6]`SIMO.2.0` is available from: `http://frege.mrg.dist.unige.it/star/sim/home.html`.

[7]`UnitWalk` is available from: `http://logic.pdmi.ras.ru/~arist/UnitWalk`.

And seven other SAT-checkers, based on different methods:

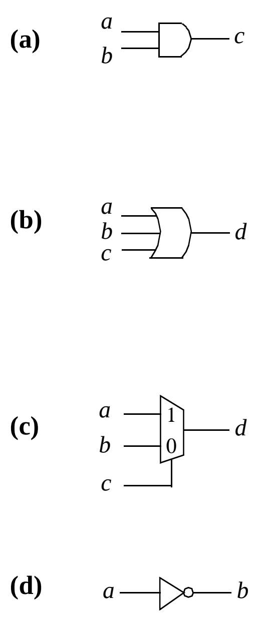
- `QSAT` (Plaisted, *et al.*, 2002), and `QBF` (Rintanen, 1999), both targeted to quantified Boolean formulas;
- `ZRes` (Chatalic and Simon, 2000), combining Zero-Suppressed BDDs (Minato, 1996, 2001) with the original Davis and Putnam (1960) procedure;
- `BSAT`, and `IS-USAT`, both using BDDs and exploiting the properties of unate Boolean functions (Kalla, *et al.*, 2000);
- `Prover`, a commercial SAT-checker using Stålmarck's method (Stålmarck, 1989); and
- `HeerHugo` (Groote and Warners, 2000), a publicly available SAT-checker that also uses Stålmarck's method.

Additionally, we experimented with 2 ATPG tools—`ATOM` (Hamzaoglu and Patel, 1999), and `TIP` (Tafertshofer, *et al.*, 2000)—used to test the output of a benchmark for being stuck-at-0, thus triggering the justification of value 1 at the output, and turning the ATPG tool into a SAT-checker.

We also used Binary Decision Diagrams (BDDs) (Bryant, 1986, 1992), and Boolean Expression Diagrams (BEDs) (Williams, 2000). The latter is not a canonical representation of Boolean functions, but was shown by Williams, *et al.* (2000) to be extremely efficient when formally verifying multipliers.

The translation to the CNF format (Johnson and Trick), 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, then its negation will be constantly **false**, and a complete SAT-checker will be able to prove unsatisfiability. Otherwise, a satisfying assignment for the negation will be a counterexample.

In translating to CNF, we introduced a new *auxiliary Boolean variable* for every $\wedge$, $\vee$, or *ITE* operator in the Boolean correctness formula, and then imposed disjunctive constraints (clauses) that the value of a variable for an operator must be consistent with the values of the variables for the operands of that operator, given its semantics—see Figure 5.a–c. Negations ($\neg$) do not generally require introducing new variables and clauses. Instead, we can represent the value of a negation by using the complement of the variable for its argument. For example, rather than introducing variables a' and y' in Figure 6.a, we use negated versions of variables a and y respectively in Figure 6.b, thus reducing the number of variables and clauses in the CNF formula. Only the negation inserted at the top of the original Boolean correctness formula was explicitly represented with clauses, e.g., those restrictig variable w to be the negation of variable z in Figure 6. All clauses were conjuncted together, including a constraint that the top-level formula (the negation of the original Boolean correctness formula)—represented by variable w in Figure 6—must be **true**. The same translation



(a)

$$\neg a \Rightarrow \neg c \qquad \neg b \Rightarrow \neg c \qquad a \wedge b \Rightarrow c \qquad \Leftrightarrow \quad (a \vee \neg c) \wedge (b \vee \neg c) \wedge (\neg a \vee \neg b \vee c)$$

(b)

$$a \Rightarrow d \qquad b \Rightarrow d \qquad c \Rightarrow d \qquad \neg a \wedge \neg b \wedge \neg c \Rightarrow \neg d \qquad \Leftrightarrow \quad (\neg a \vee d) \wedge (\neg b \vee d) \wedge (\neg c \vee d) \wedge (a \vee b \vee c \vee \neg d)$$

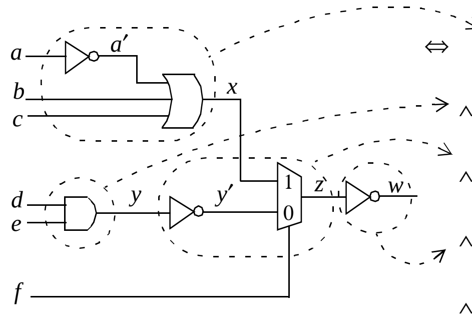
(c)

$$a \wedge c \Rightarrow d \qquad \neg a \wedge c \Rightarrow \neg d \qquad b \wedge \neg c \Rightarrow d \qquad \neg b \wedge \neg c \Rightarrow \neg d \qquad \Leftrightarrow \quad (\neg a \vee \neg c \vee d) \wedge (a \vee \neg c \vee \neg d) \wedge (\neg b \vee c \vee d) \wedge (b \vee c \vee \neg d)$$

(d)

$$a \Rightarrow \neg b \qquad \neg a \Rightarrow b \qquad \Leftrightarrow \quad (\neg a \vee \neg b) \wedge (a \vee b)$$

Figure 5: Translation of Boolean operators to CNF: **(a)** $\wedge$; **(b)** $\vee$; **(c)** *ITE*; and **(d)** $\neg$.



$$\Leftrightarrow \quad (a \vee x) \wedge (\neg b \vee x) \wedge (\neg c \vee x) \wedge (\neg a \vee b \vee c \vee \neg x)$$
$$\wedge (d \vee \neg y) \wedge (e \vee \neg y) \wedge (\neg d \vee \neg e \vee y)$$
$$\wedge (\neg x \vee \neg f \vee z) \wedge (x \vee \neg f \vee \neg z) \wedge (y \vee f \vee z) \wedge (\neg y \vee f \vee \neg z)$$
$$\wedge (\neg z \vee \neg w) \wedge (z \vee w)$$
$$\wedge\, w$$

(a) **(b)**

Figure 6: Translation of a Boolean formula **(a)** to a CNF formula **(b)** by replacing internal negations with the complements of their arguments, when we want to find a falsifying assignment for the original output z. Variables a, b, c, d, e, and f are primary variables since they represent inputs to the formula. Variables x, y, and z are auxiliary variables and are introduced to represent the values of operators other than negations. The top negation with output w, also an auxiliary variable, and the last constraint "$\wedge\ w$" were added so that a satisfying assignment for the CNF formula (b) will be a falsifying assignment for the original output z.

of Boolean formulas to CNF format was used by Larrabee (1992), except that negations were explicitly represented with clauses (see Figure 5.d). When generating the Boolean correctness formula in EVC (Velev and Bryant, 2001b), we hashed the expressions and kept only one copy of isomorphic operators. This significantly reduced the size of the correctness formula, as well as the number of CNF variables and clauses. The variables in the original Boolean correctness formula will be called *primary Boolean variables.*

We ran the experiments on a 336-MHz Sun4 with 4 GB of physical memory. For the BDD-based runs, we used the BDD package `CUDD` version 2.3.0 (Somenzi, 2001)[8], and the sifting dynamic variable reordering heuristic (Rudell, 1993). In the BED evaluations[9], we experimented with converting the final BED into a BDD with both the `up_one()` and `up_all()` functions (Williams, 2000) by employing 4 different variable ordering heuristics—variants of the depth-first and fanin heuristics (Malik, *et al.*, 1988)—that were the most efficient when verifying multipliers (Williams, *et al.*, 2000).

SAT Procedure	% Satisfiable in		
	< 24 sec	< 240 sec	< 2,400 sec
`Chaff`	100	100	100
`BerkMin`	97	100	100
`DLM-3`	51	82	98
`DLM-2`	50	84	97
`UnitWalk`	45	81	98
`CGRASP`	44	49	68
`QSAT`	33	47	52
`SATO`	22	30	69
`GRASP`	14	21	24
`rel_sat.1.0`	13	17	22
`WalkSAT`	10	16	27
`rel_sat_rand`	10	19	29
`SIMO`	7	14	16
`CLS`	5	8	10
`GRASP` with restarts	4	11	14
`rel_sat.2.1`	3	58	97
`DLM-2000`	2	24	66
BDDs	2	2	3
`eqsatz`	1	5	7

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

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 (whose CNF correctness formula had 776 variables, and 3,725 clauses), or could not scale for its dual-issue version, 2×DLX-CC (1,516 CNF variables, and 12,812 clauses) that does not implement exceptions, multicycle functional units, and branch prediction. For example, `Prover` could not solve the Boolean formula for correctness of 2×DLX-CC within 24 hours, as reported in our earlier work (Velev and Bryant, 1999). The SAT-checker `Chaff` had the best performance, finding a satisfying assignment for each benchmark in less than 24 seconds (indeed, less than 23.24 seconds). It

[8]`CUDD` is available from: `http://vlsi.colorado.edu/~fabio`.

[9]Based on BED package version 2.5, available from: `http://www.it-c.dk/research/bed`.

was closely followed by `BerkMin` that solved 97 instances in less than 24 seconds each, and required less than 29 seconds for each of the other three benchmarks. We ran the rest of the SAT procedures for 240 and 2,400 seconds—one and two orders of magnitude more than `Chaff`. `DLM-3` and `DLM-2` were third and fourth, respectively, but could solve only half the instances within the time limit of 24 seconds. `UnitWalk` and `CGRASP` solved 45 and 44 instances, respectively, in 24 seconds for each, followed by `QSAT` with 33 of the benchmarks under 24 seconds. The rest of the SAT procedures, including BDDs, performed significantly worse. `DLM-2000` is slower than `DLM-3` and `DLM-2` because of extensive analysis before each decision.

The 100 buggy variants of 2×DLX-CC-MC-EX-BP are available as benchmark suite SSS-SAT.1.0 (Velev, 2000b), and have been used for SAT experiments by many researchers. Lynce, *et al.* (2001) present the SAT-checker `Quest0.5`, built on top of GRASP and based on restarts and random backtracking. They report that `Quest0.5` took 292 seconds to solve the 100 buggy variants of 2×DLX-CC-MC-EX-BP on their computer, while `Chaff` required 84 seconds, i.e., was approximately 3.5 times faster. Janssen (2001) describes a pointerless BDD package that required less memory than `CUDD` version 2.3.0 when run on the benchmarks in suite SSS.1.0 (Velev, 1999), consisting of 48 variants of 2×DLX-CC. His BDD package was up to 3 times faster on 8 of the benchmarks, but required comparable CPU time for the rest. He does not present results for the more challenging 100 buggy variants of 2×DLX-CC-MC-EX-BP.

When verifying the correct version of 2×DLX-CC-MC-EX-BP (4,583 CNF variables, and 41,704 clauses), `BerkMin` was the fastest—requiring 15 seconds, followed by `Chaff` with 22 seconds. BDDs took 2,635 seconds (Velev and Bryant, 2000), while `QSAT`—14 hours and 37 minutes. `CGRASP`, `SATO`, `GRASP`, and `GRASP` used in a mode with restarts, randomization, and recursive learning did not finish in 24 hours.

We then compared `Chaff` and `BerkMin` on the 100 buggy VLIW designs: `Chaff` was better in 73 cases; `BerkMin` was faster by at least 60 seconds on only 7 benchmarks. For `Chaff`, the minimum time per benchmark was 3.7 seconds and the maximum 180.4 seconds, as compared to a minimum of 8.7 seconds and a maximum of 151.4 seconds for `BerkMin`. The average time was 32.5 seconds for `Chaff`, and 43.6 seconds for `BerkMin`. The variations in the times to find a satisfying assignment, i.e., to detect bugs, can be explained with the fact that the single or multiple errors in a buggy design affect a different number of architectural state elements and under different conditions than the errors in another buggy design. Generally, errors that affect fewer architectural state elements and under rare conditions are harder to detect. The SAT-checker `DLM-3` did not complete 5 of the VLIW benchmarks in 3,600 seconds. The CNF formulas from verification of the 100 buggy VLIW designs are available as benchmark suite VLIW-SAT.1.0 (Velev, 2000b).

The correct 9VLIW-MC-BP had a CNF formula with 20,093 variables, and 179,492 clauses. `Chaff` took 759 seconds to prove the unsatisfiability of that formula, while `BerkMin` required 224 seconds. In the original experiments, BDDs took 31.5 hours (Velev, 2000a).

While preparing the final version of this paper, we learned of another proprietary SAT-checker (Pilarski and Hu, 2002), recently developed at Synopsys, Inc., and also extending the ideas from `Chaff`. Pilarski and Hu (2002) report that their SAT solver is 2.4 times faster than `zChaff`[10] on the 100 buggy superscalar benchmarks (SSS-SAT.1.0), 3.3 times faster on the 100 buggy VLIW benchmarks (VLIW-SAT.1.0), and more than 6 times faster on the unsatisfiable CNF instances from correct designs (FVP-UNSAT.1.0) that include the VLIW processor. However, we found `zChaff` to be slower than `mChaff`, which is used in this paper. We could not obtain Pilarski and Hu's SAT-checker in order to directly compare it with `Chaff` (i.e., `mChaff`) and `BerkMin`.

Figure 7 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 simpler correctness criteria in parallel (Velev, 2000a)—see Section 7. The assumption is that there are enough computing resources for parallel runs of the verification tool `EVC` (Velev and Bryant, 2001b) that can directly use BDDs, instead of saving the formula in CNF format. As soon as one of these parallel runs finds 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. In a different body of work—Bounded Model Checking—Clarke, *et al.* (2001), Bjesse, *et al.* (2001), and Copty, *et al.* (2001) also report that SAT-checkers performed much better than BDDs.

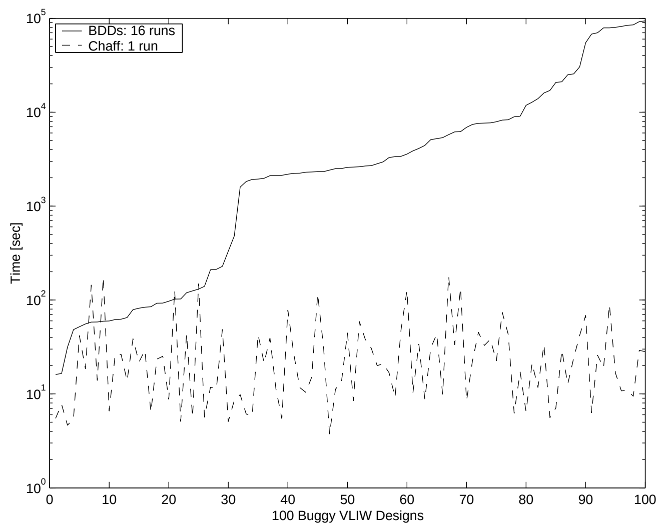


Figure 7: Comparison of `Chaff` (with 1 run) and BDDs (with 16 parallel runs) on the 100 buggy versions of 9VLIW-MC-BP. The benchmarks are sorted in ascending order of their times for the BDD-based experiments.

[10]Available from `http://www.ee.princeton.edu/~chaff/software.php`.

Applying the script `simplify` (Marques-Silva, 2000) in order to algebraically simplify 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 that the buggy VLIW designs had CNF formulas of up to 25,000 variables, and up to 450,000 clauses. Another simplifier, presented by Brafman (2001), took 130 seconds to process the CNF formula, but did not speed up `Chaff`. We also tried the MINCE heuristic (Aloul, *et al.*, 2001) that uses a min-cut linear placement algorithm in order to statically rename the CNF variables in a way that reduces the cutwidth of the formula. MINCE took 3,203 seconds, and the resulting CNF formula required an almost doubled CPU time for a solution by `Chaff`. Finally, Wang, *et al.* (2001) propose another algorithm for computing a CNF variable ordering that reduces the cutwidth. They ran it on the 9 satisfiable CNF instances in benchmark suite SSS.1.0a (Velev, 1999)—formulas generated in the formal verification of buggy variants of 2×DLX-CC. Their algorithm could not complete five of the instances within a time limit of 10,000 seconds for each, and solved the other four after more than 5,700 seconds total. In contrast, `Chaff` solves these 9 instances in 26 seconds total. Therefore, attempts to preprocess CNF formulas prior to SAT-checking did not yield improvement.

Hence, based on experiments with two benchmark suites, each consisting of one correct high-level processor and 100 buggy variants of the same design, we identified `Chaff` and `BerkMin` as the most efficient SAT procedures for solving satisfiable CNF instances generated in the formal verification of incorrect processors, with `BerkMin` being significantly faster on unsatisfiable instances from formal verification of the correct designs. As observed in our earlier work (Velev and Bryant, 2001a), the breakthrough occurred with `Chaff`, which was more than 2 orders of magnitude faster than the other SAT solvers available at the time. `BerkMin` was created later and further develops the ideas from `Chaff`, but is a proprietary SAT solver that is not publicly available.[11] How do such powerful SAT-checkers change the frontier of possibilities? The rest of the paper examines ways to increase the productivity in microprocessor formal verification by using `Chaff` and `BerkMin` as the back-end SAT procedures.

5. Impact of Variations in Eliminating UFs and UPs

When translating the EUFM correctness formula to an equivalent Boolean formula, we can apply the following two structural variations:

- **Early reduction of p-equations.** In eliminating UFs and UPs and enforcing functional consistency with nested *ITE*s (see Section 2.2), the translation algorithm introduces equations between argument terms in order to control the nested *ITE*s. Although such equations are implicitly negated for the case when they select the else-term of one of these nested *ITE*s, we

[11]We thank E. Goldberg for releasing `BerkMin` to us.

can still treat them as p-equations as long as each of their argument terms has only p-term variables in its support (Velev and Bryant, 1999)(Bryant, German, and Velev, 2001). This results in replacing the UFs and UPs with lookup tables that map each unique combination of symbolic expressions at the inputs of the UFs or UPs to a corresponding new term variable or Boolean variable, respectively, for the output value. The elimination of UFs and UPs is done recursively, starting from the leaves of the EUFM correctness formula. Then, when an application of an UF or UP is eliminated, the expressions for its input terms will consist of only nested *ITE*s that select one from a set of supporting term variables. If the terms on both sides of an equation have disjoint supports of p-term variables, then the two compared terms will not be equal under a maximally diverse interpretation, and their equation can be replaced with the constant **false**. This is done in the final step of the translation algorithm (Velev and Bryant, 1999). However, an early reduction of such equations will result in a different (but equivalent) structure of the Boolean correctness formula, i.e., in a different (but equivalent) CNF formula to be evaluated by SAT-checkers.

- **Eliminating UPs with Ackermann constraints.** Using Ackermann constraints (Ackermann, 1954) to enforce the functional consistency of eliminated UFs and UPs, as discussed in Section 2.2, yields a negated equation for the new variables, c_1 and c_2, that replace the original UF or UP applications:

$$[(a_1 = a_2) \wedge (b_1 = b_2) \Rightarrow (c_1 = c_2)] \;\Rightarrow\; F',$$

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 new variables 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 to eliminate UFs whose results appear only in positive equations. However, Ackermann constraints can be used to eliminate 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 to eliminate UPs.

In order to exploit the above structural variations, we can run parallel copies of the formal verification tool flow, all of them applied to the same design, and each using a different structural variation or combination thereof. Then, one satisfying assignment is enough to detect a bug, i.e., we take the minimum of the run times as the time to find the bug.

As Table 2 shows, when each of `Chaff` and `BerkMin` was used in 4 parallel runs—1 base run without structural variations and 3 runs with such variations (i.e., base, ER, AC, ER+AC, as explained in the table)—the average time was reduced by a factor of 2—from 32.5 to 14.4 seconds for `Chaff`, and from 43.6 to 20.3 seconds for `BerkMin`. Similarly, the maximum time was reduced by a factor of 2.5 for both tools—from 180.4 to 74.9 seconds for `Chaff`, and from 151.4 to 62.0 seconds for `BerkMin`. We also ran `Chaff` with 3 parameter variations in the base configuration file, `cherry_032301`, as suggested by Moskewicz (2001): 1) the restart period was increased from 2,000 to 3,000; 2) the restart period was increased from 2,000 to 4,000; and 3) the randomness at restart was increased from 3 to 10. The results of these 3 runs with parameter variations, combined with the base run, are summarized in the last row of Table 2. The reduction in the average time was comparable to that achieved in any of the other experiments with 4 parallel runs where `Chaff` was used. `BerkMin` was released to us without the option to vary its command parameters. However, when verifying the correct VLIW processor, structural variations slowed both `Chaff` and `BerkMin`, while parameter variations slowed `Chaff`.

SAT-Checker	Variations for each tool	Comment	Parallel Runs	Max. Time [sec]	Average Time [sec]
`Chaff`	base	—	1	180.4	32.5
`BerkMin`	base	—	1	151.4	43.6
`Chaff`, `BerkMin`	base	1 run per tool	2	138.3	22.3
`Chaff`	base, ER, AC, ER+AC	—	4	74.9	14.4
`BerkMin`	base, ER, AC, ER+AC	—	4	62.0	20.3
`Chaff`, `BerkMin`	base, ER	2 runs per tool	4	132.0	17.1
`Chaff`, `BerkMin`	base, AC	2 runs per tool	4	61.3	17.0
`Chaff`, `BerkMin`	base, ER+AC	2 runs per tool	4	68.2	15.1
`Chaff`	base, base1, base2, base3	—	4	176.8	15.0

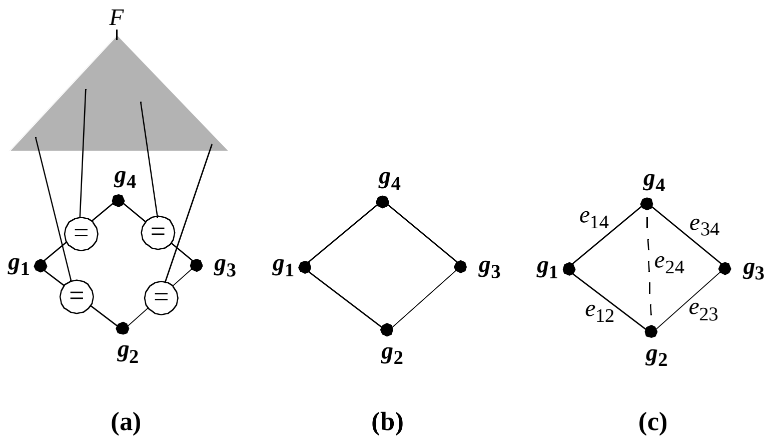
Table 2: Maximum and average times for finding satisfying assignments in the formal verification of the 100 buggy VLIW designs when structural and parameter variations were used: "base" means no structural variations; "ER" stands for early reductions of p-equations; "AC" for Ackermann constraints in eliminating UPs; "base1," "base2," and "base3" mean no structural variations when generating the Boolean correctness formula, but variations of `Chaff`'s command parameters as explained in the text.

Therefore, although structural or parameter variations can speedup the detection of bugs, if resources are available for parallel runs of the tool flow, that was not critical for the 100 buggy VLIW designs, since the maximum and average times for the base runs with either `Chaff` or `BerkMin` were so low. Neither structural nor parameter variations accelerated the verification of the correct VLIW processor, regardless of the SAT-checker.

6. Impact of G-Equation Encodings

When translating the EUFM correctness formula to an equivalent Boolean formula, we can encode the g-equations with one of the following two schemes:

- **The e_{ij} encoding.** The equation $g_i = g_j$, where g_i and g_j are g-term variables, is replaced by a new Boolean variable e_{ij} (Goel, *et al.*, 1998). Transitivity of equality, i.e., the property $(g_i = g_j) \wedge (g_j = g_k) \Rightarrow (g_i = g_k)$, has to be enforced additionally, e.g., by triangulating the equality comparison graph of the e_{ij} variables that affect the final Boolean formula and then enforcing transitivity for each of the resulting triangles, as done in our sparse transitivity scheme (Bryant and Velev, 2002)—see Figure 8 for an example. The triangulation is done iteratively, in a greedy manner, such that at each step: nodes of degree 1 and their single edges are removed, since such nodes are not part of cycles for which transitivity of equality has to hold; the node of the smallest degree $n \geq 2$ is found; up to $n-1$ extra edges are added, if they do not exist already, in order to form $n-1$ triangles with the node's edges (e.g., edge g_2–g_4 is added in Figure 8.c in order to triangulate the two edges, g_1–g_2 and g_1–g_4, of node g_1); the node and its edges are removed, and the procedure is applied to the remaining nodes by considering the newly added edges; finally, the original and the extra edges are put together to form the triangulated equality comparison graph. Although not every correct microprocessor requires transitivity for its correctness proof, that property is needed in order to avoid false negatives for buggy designs or for processors that do need transitivity.



Transitivity of equality for g_1–g_2–g_4:

$(\neg e_{12} \vee \neg e_{24} \vee e_{14})$

$(\neg e_{24} \vee \neg e_{14} \vee e_{12})$

$(\neg e_{14} \vee \neg e_{12} \vee e_{24})$

Transitivity of equality for g_2–g_3–g_4:

$(\neg e_{23} \vee \neg e_{34} \vee e_{24})$

$(\neg e_{34} \vee \neg e_{24} \vee e_{23})$

$(\neg e_{24} \vee \neg e_{23} \vee e_{34})$

(d)

Figure 8: The e_{ij} encoding of g-equations. **(a)** An EUFM formula F, where g-term variables g_1, g_2, g_3, and g_4 are compared for equality in a cycle of length four; **(b)** the equality comparison graph between g_1, g_2, g_3, and g_4—an edge indicates an equality comparison; **(c)** the triangulated equality comparison graph, with one extra edge g_2–g_4 added, and e_{ij} variables assigned to the edges; **(d)** transitivity of equality constraints for the two triangles of the graph in (c).

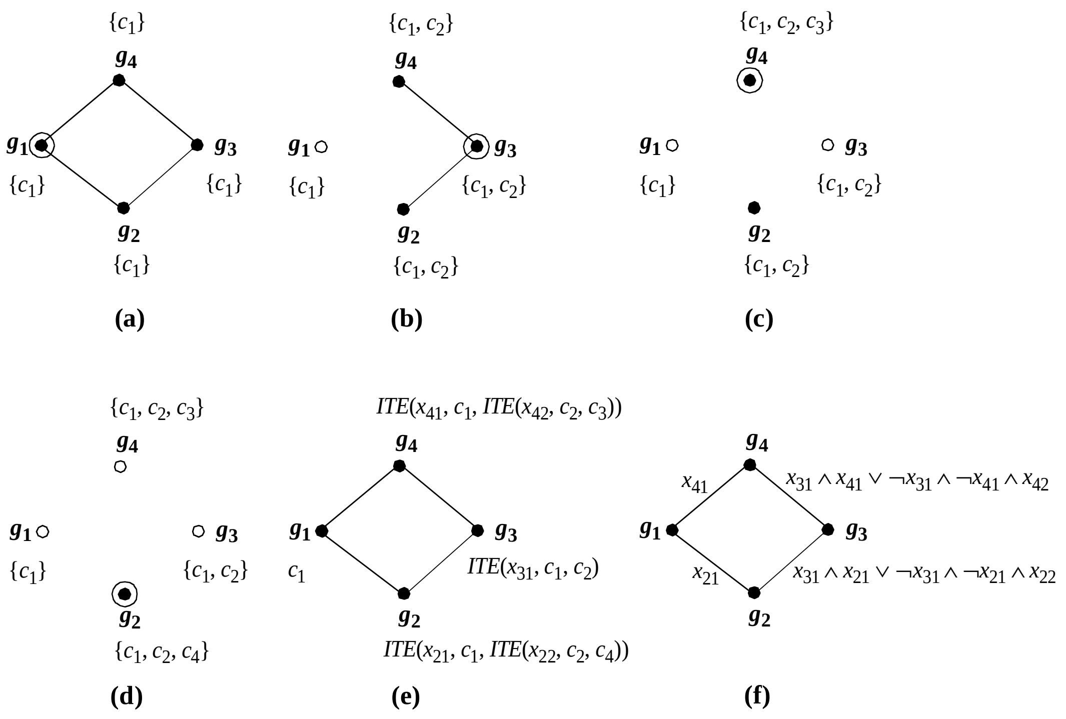


Figure 9: The small-domain encoding of g-equations, applied to the equality comparison graph in Figure 8.b with the greedy strategy of assigning a characteristic constant to the unprocessed node of highest degree. Ties are broken randomly. A circled node, e.g., g_1 in (a), means that the node is currently being processed, i.e., assigned a characteristic constant. The same constant is also added to the sets of constants for the nodes that can be reached via a path of edges starting from the currently processed node. After a node is processed, its edges are removed. An empty node, e.g., g_1 in (b), means that the node has already been processed. **(a)** Node g_1 was chosen randomly, since all nodes have degree 2 initially. **(b)** Node g_3 was chosen, since it has the highest degree 2. **(c)** Node g_4 was chosen randomly from the unprocessed nodes g_2 and g_4 of degree 0. **(d)** The only unprocessed node, g_2, was assigned a characteristic constant. **(e)** Based on the constants in its set, each g-term variable g_i is assigned a nested-*ITE* expression that is controlled by new indexing variables x_{ik}, and evaluates to one of these constants, given an assignment to the variables x_{ik}. **(f)** Each edge in the equality comparison graph is labeled with a Boolean formula, encoding the conditions when the two g-term variables at the ends of the edge will simultaneously evaluate to a common constant, i.e., will be equal.

- **The small-domain encoding.** Every g-term variable is assigned a set of constants that it can take on in a way that allows it to be either equal to or different from any other g-term variable with which it can be transitively compared for equality (Pnueli, *et al.*, 1999)—see Figure 9.a–d. If there are N constants in the set for a g-term variable, those can be indexed with $\lceil log_2(N) \rceil$ new Boolean variables that will be used to control nested *ITE*s selecting a mapping of the g-term variable to a constant in the set—see Figure 9.e. For example, g-term variable g_2 is assigned a set of three constants $\{c_1, c_2, c_4\}$ in Figure 9.d, so that we can introduce two indexing variables, x_{21} and x_{22}, and form the expression $ITE(x_{21}, c_1, ITE(x_{22}, c_2, c_4))$ that will be used to replace g_2. Then, two g-term variables will be equal if their indexing variables simultaneously select the same common constant—see Figure 9.f. Hence, g-term variables in a cycle can be equal if they simultaneously evaluate to the same common constant, so that transitivity of equality is automatically enforced in this encoding. Depending on the structure of the equality comparison graph, the small-domain encoding might introduce fewer primary Boolean variables than the e_{ij} encoding. That would mean a smaller search space. However, now many g-equations will get replaced by a Boolean formula—a disjunction of conjuncts, each consisting of many Boolean variables or their complements, and encoding the possibility that two g-term variables evaluate to the same common constant. In contrast, in the e_{ij} encoding, a g-equation always gets replaced by a single Boolean variable.

We compared the two encodings on the 100 buggy VLIW designs—see Table 3. When using `Chaff` with a single run of the tool flow, the e_{ij} encoding (used for the experiments before this section) resulted in 3 times faster detection of bugs—the maximum and average times were 180.4 and 32.5 seconds, compared to 594.0 and 100.4 seconds with the small-domain encoding. Constraints for transitivity of equality were included when using the e_{ij} encoding. When four parallel runs with structural variations were employed (base, ER, AC, ER+AC—see Section 5), the e_{ij} encoding was again faster—at least 2.5 times—with maximum and average times of 74.9 and 14.4 seconds, compared to 338.4 and 35.2 seconds with the small-domain encoding. `BerkMin` was similarly faster with the e_{ij} encoding. Figure 10 shows a detailed plot of `BerkMin`'s performance on one run with each encoding—the e_{ij} encoding resulted in faster detection of bugs for 87 of the 100 designs.

SAT-Checker	Parallel Runs	G-Equation Encoding			
		e_{ij}		small-domain	
		Max. Time [sec]	Average Time [sec]	Max. Time [sec]	Average Time [sec]
`Chaff`	1	180.4	32.5	594.0	100.4
	4	74.9	14.4	338.4	35.2
`BerkMin`	1	151.4	43.6	245.0	85.0
	4	62.0	20.3	226.5	56.7

Table 3: Comparison of the e_{ij} and small-domain encodings on the 100 buggy versions of 9VLIW-MC-BP, using both `Chaff` and `BerkMin`. The experiments with 1 run of the tool flow are without structural variations. The experiments with 4 runs also include a run with early reduction of p-equations, another with Ackermann constraints used to eliminate UPs, and a fourth run with both of these transformations (see Section 5).

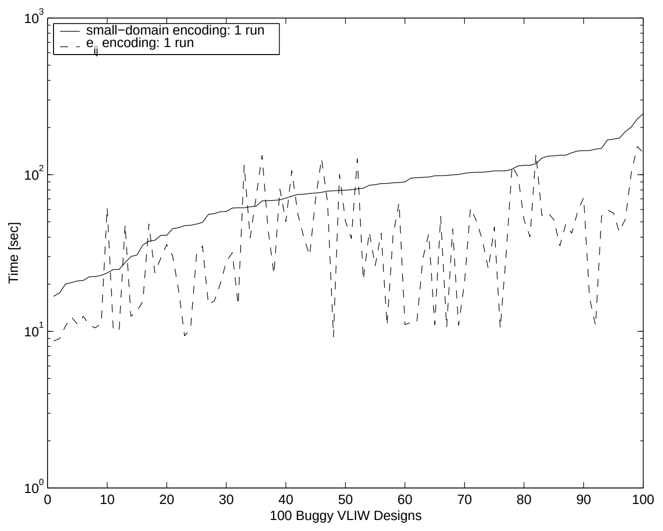


Figure 10: Comparison of the e_{ij} and small-domain encodings on the 100 buggy versions of 9VLIW-MC-BP, using `BerkMin` and one run of the tool flow without structural variations. The benchmarks are sorted in ascending order of their times with the small-domain encoding. `BerkMin` was used for this plot, because `BerkMin` performed better than `Chaff` on one run with the small-domain encoding (see Table 3).

When verifying the correct 9VLIW-MC-BP, the e_{ij} encoding required more than twice as many primary Boolean variables as the small-domain encoding, but half the CPU time for SAT-checking with `Chaff`—2,615 primary Boolean variables (2,353 of them being e_{ij} variables) and 759 seconds of CPU time, compared with 1,152 primary Boolean variables (890 of them being indexing variables) and 1,479 seconds of CPU time with the small-domain encoding. `BerkMin` took 224 seconds with the e_{ij} encoding, but 418 seconds with the small-domain encoding. Again, constraints for transitivity of equality were included in the formula generated with the e_{ij} 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 (Hennessy and Patterson, 2002) 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.

As shown in Table 4, the small-domain encoding introduces fewer primary Boolean variables—one fourth of those required by the e_{ij} encoding for the 6-wide design—but results in approximately 50% more CNF variables, and 10–20% more CNF clauses, and thus in longer CPU times with either SAT-checker—see Table 5. `BerkMin` was an order of magnitude faster than `Chaff` on the 4-, 5-,

Issue Width	G-Equation Encoding					
	e_{ij}			small-domain		
	Primary Boolean Variables	CNF Variables	CNF Clauses	Primary Boolean Variables	CNF Variables	CNF Clauses
2	139	925	8,213	81	1,294	9,803
3	308	2,577	33,270	127	3,780	39,475
4	553	5,525	96,480	194	8,362	112,636
5	857	10,113	240,892	249	15,647	275,581
6	1,243	17,186	528,962	304	26,738	590,832

Table 4: Statistics for the e_{ij} and small-domain encodings when verifying correct out-of-order superscalar processors. These designs require transitivity of equality for their correctness proofs. The results are listed as a function of the processor issue width.

Issue Width	G-Equation Encoding			
	e_{ij}		small-domain	
	`Chaff` Time [sec]	`BerkMin` Time [sec]	`Chaff` Time [sec]	`BerkMin` Time [sec]
2	3.9	1.6	7.3	1.7
3	46	15	49	19
4	653	65	1,049	99
5	1,381	154	1,864	255
6	68,896	1,957	132,428	3,206

Table 5: CPU time to prove unsatisfiability when verifying correct out-of-order superscalar processors. These designs require transitivity of equality for their correctness proofs. Both `Chaff` and `BerkMin` were run on the same correctness formula, which was generated without structural variations. The results are listed as a function of the processor issue width.

and 6-wide designs, due to `BerkMin`'s heuristics that are fine-tuned for CNF formulas derived from deeply nested expressions (Goldberg, 2002)—the case in these benchmarks. The CNF formulas from the out-of-order processors are available as benchmark suite FVP-UNSAT.2.0 (Velev, 2000b).

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 branching behavior in instruction execution. 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 enforced by automatic application of the unit-clause rule implemented in SAT-checkers—if there is a single unassigned literal in a CNF clause, with the rest of the literals being **false**, then the unassigned literal has to get value **true** in order for the clause to be satisfied. Such an assignment is called an implication. Hence, as soon as two variables, e_{ij} and e_{jk}, in triangle e_{ij}–e_{jk}–e_{ki} become **true**, the third variable e_{ki} in that triangle will be assigned value **true**, due to the imposed transitivity constraint $(\neg e_{ij} \vee \neg e_{jk} \vee e_{ki})$, where both $\neg e_{ij}$ and $\neg e_{jk}$ will be already **false**. Transitivity of equality is similarly enforced on cycles of any length (Bryant and Velev, 2002), since cycles longer than 3 are triangulated with new e_{ij} variables.

The small-domain encoding requires more CNF variables and more implications to enforce transitivity of equality. In the example cycle of length 4 in Figure 9.f, we will need to introduce auxiliary Boolean variables f_{23} and f_{34} to represent the values of the Boolean functions that encode the equality comparisons $(g_2 = g_3)$ and $(g_3 = g_4)$, respectively. Also, since each of these formulas is a disjunction of two conjuncts, we will need an auxiliary variable for the output of each conjunct, for a total of 6 auxiliary Boolean variables, in addition to the 5 indexing variables—x_{21}, x_{22}, x_{31}, x_{41}, and x_{42}. Hence, the small-domain encoding will require 11 CNF variables to encode the equality comparisons (no additional constraints are needed to enforce transitivity) in the example cycle of length 4. In contrast, the e_{ij} encoding will introduce 5 CNF variables—one for each of the original 4 equality comparisons, and another for the triangulating edge $(g_2 = g_4)$ that was added to enforce transitivity—see Figure 8.c. Therefore, given an assignment of values to 3 CNF variables representing the outputs of 3 of the g-equations in the cycle of length 4, the small-domain encoding will require up to 8 implications (one for each of the other 8 CNF variables related with the cycle) to enforce transitivity of equality, as compared to at most 2 implications with the e_{ij} encoding, where each triangle may trigger an implication.

With the e_{ij} encoding, the number of Boolean variables, encoding equality and transitivity constraints for a cycle, does not depend on the number of outside edges (i.e., equality comparisons) that are not part of a cycle and that are connected to nodes in the cycle. On the other hand, with the small-domain encoding, outside edges might result in more constants being added to the sets for all or some of the nodes in the cycle. Those constants will come from outside nodes (i.e., g-term variables) that are not part of the cycle, but that can be transitively compared for equality with each node in the cycle. Such extra constants might result in extra indexing variables and conjuncts in the disjunctive formulas encoding the equality comparisons in the cycle, thus increasing both the complexity of those formulas and the number of implications required to enforce transitivity. Also, if the same additional constant gets included in the sets of all g-term variables in a cycle—e.g., due to an outside node that gets assigned a characteristic constant before the nodes in the cycle, such that this node is transitively connected with the nodes in the cycle—then the nodes in the cycle can be equal in more than one way, which will increase the number of implications required to enforce transitivity. Hence, the small-domain encoding enumerates all mappings of g-term variables to a sufficient set of distinct constants, thus introducing more information than actually required to solve the problem.

An additional source of inefficiency in the small-domain encoding is that an f_{ij} variable, representing the output of a function encoding the equality $g_i = g_j$, can be **true** for many assignments to its supporting indexing variables, and can be **false** for many other assignments to those variables. Hence, it is possible that portions of the search space (e.g., where $f_{ij} = \mathbf{true}$) will be revisited multiple times. Although learning, employed in both `Chaff` and `BerkMin`, can reduce or even eliminate such revisits, both SAT-checkers age the learned clauses and periodically discard old learned clauses—hence the possibility to revisit portions of the search space. Note that each feasible assignment to e_{ij} variables (i.e., assignment not violating transitivity of equality) is a feasible assignment to f_{ij} variables, except that it can be justified with many possible assignments to the indexing variables. Therefore, multiple branches in the formula could be revisited for what would be just one visit with the e_{ij} encoding. As a result of all these factors, the e_{ij} encoding is more efficient than the small-domain encoding.

In a different application—encoding constraint satisfaction problems as SAT instances—Hoos (1999) similarly found that better performance is achieved with an encoding that introduces more primary Boolean variables, but results in conceptually simpler search spaces.

In their decision procedure based on the small-domain encoding, Pnueli, *et al.* (1999) use Ackermann constraints to eliminate all UFs, including those that appear only in p-equations in the EUFM correctness formula. Thus, when enforcing functional consistency, they introduce negated equations for the new term variables that replace such UFs (as discussed in Section 5), turning these term variables into g-terms, whose equations will have to be encoded with Boolean functions. In contrast, by exploiting Positive Equality and the nested-*ITE* scheme for eliminating UFs, we treat the new term variables as p-terms, thus reducing the number of g-equations that have to be encoded with new Boolean variables.

7. Impact of Decomposing the Correctness Criterion

The correctness criterion can be evaluated with one monolithic computation:

$$(f_{0,1} \wedge f_{0,2} \wedge \ldots \wedge f_{0,N}) \vee \ldots \vee (f_{k,1} \wedge f_{k,2} \wedge \ldots \wedge f_{k,N}) = \mathbf{true},$$

where $f_{l,m}$ is a Boolean formula checking whether memory element m is updated by l instructions, $0 \le l \le k$, given the fetch width k of the processor. Formulas $f_{l,m}$ are produced after translating a corresponding EUFM formula to a Boolean formula by exploiting Positive Equality. However, the evaluation can be decomposed (Velev, 2000a) by selecting a set of disjoint window functions w_l, one for each index l, where w_l consists of either just one of the functions $f_{l,m}$ or a conjunction of several of them with the same index l, such that $w_0 \vee \ldots \vee w_k = \mathbf{true}$, and then proving that $w_l \Rightarrow f_{l,i}$ for each l and each i such that $f_{l,i}$ is not used in forming w_l. That results in a set of smaller computations (weak correctness criteria), each depending on only a subset of the formulas $f_{l,m}$ in the monolithic computation. However, proving all of these weak criteria is sufficient to imply that the monolithic criterion is **true**, without actually evaluating it. That resulted in a factor of 4 reduction in the CPU time for the BDD-based evaluation of the correct 9VLIW-MC-BP (Velev, 2000a). Note that when proving correctness with multiple parallel runs by using decomposition, we need to wait until all of them complete, taking the maximum CPU time as the verification time. All experiments in this section use the e_{ij} encoding of g-equations.

The benefits of decomposition when verifying the 100 buggy versions of 9VLIW-MC-BP with `Chaff` and `BerkMin` are shown in Table 6. Using `Chaff` and running 8 weak correctness criteria, the maximum CPU time is reduced from 180.4 to 31.3 seconds and the average from 32.5 to 4.1 seconds, while running 16 weak criteria results in a maximum of 17.5 seconds and an average of 2.8 seconds. The performance of `BerkMin` is very similar on these benchmarks. While the achieved reductions are not critical for the present set of benchmarks, decomposition might become important for detecting bugs in more complex designs.

Parallel Runs	Chaff			BerkMin		
	Min.	Max.	Average	Min.	Max.	Average
1	3.7	180.4	32.5	8.7	151.4	43.6
8	0.3	31.3	4.1	2.2	32.7	8.5
16	0.2	17.5	2.8	2.3	18.6	6.3

Table 6: CPU time (in seconds) to detect error in the 100 buggy versions of 9VLIW-MC-BP, using `Chaff` or `BerkMin`. Each SAT-checker was run in parallel on up to 16 weak correctness criteria, stopping as soon as one of the runs finds a satisfying assignment that triggers a bug.

Both `Chaff` and `BerkMin` had sufficient capacity to verify an extension of 9VLIW-MC-BP that implements exceptions, yielding the processor 9VLIW-MC-BP-EX. The Instruction Memory, the ALUs, and the Data Memory could each raise an exception. The exception conditions were stored in three new architectural state elements—one for each of the exception sources. The Program Counter (PC) of the instruction that raises an exception was stored in another new architectural state element, the Exception PC (EPC). The design also implemented a return-from-exception instruction that transfers the value of the EPC to the PC, allowing the program execution to resume after a software exception handler fixes the cause of the exception.

Four bugs were generated inadvertently when creating 9VLIW-MC-BP-EX, but were detected in 12.2 to 108.4 seconds by `Chaff` when run on a monolithic correctness criterion—see Table 7. `BerkMin` was consistently slower than `Chaff` when using a monolithic correctness criterion, but was faster when detecting Bugs 3 and 4 with 22 weak correctness criteria checked in parallel.

Bugs while Designing 9VLIW-MC-BP-EX	Parallel Runs	Max. Primary Boolean Variables	`Chaff` CPU Time [sec]	`BerkMin` CPU Time [sec]
Bug1	1	5,127	16.2	65.0
	20	4,926	10.2	15.4
Bug2	1	5,400	12.2	50.0
	20	5,043	10.9	16.4
Bug3	1	3,500	29.3	53.0
	22	3,106	18.3	5.4
Bug4	1	3,500	108.4	153.0
	22	3,106	39.5	22.0

Table 7: Effect of decomposing the correctness condition when detecting 4 actual bugs in the design of 9VLIW-MC-BP-EX, using `Chaff` and `BerkMin`. The experiments with 1 run are based on a monolithic correctness criterion.

Processor	Parallel Runs	Max. Primary Boolean Variables	`Chaff` CPU Time [sec]	`BerkMin` CPU Time [sec]
9VLIW-MC-BP	1	3,108	759	224
	8	2,273	349	134
	16	2,273	264	63
9VLIW-MC-BP-EX	1	3,587	1,094	347
	11	3,243	519	167
	22	3,175	473	173

Table 8: Effect of decomposing the correctness condition when verifying the correct versions of 9VLIW-MC-BP and its extension with exceptions, 9VLIW-MC-BP-EX, using `Chaff` and `BerkMin`. The experiments with 1 run are based on a monolithic correctness criterion.

Table 8 shows the effect of decomposition when verifying correct designs. Using 8 weak correctness criteria to verify 9VLIW-MC-BP resulted in approximately a factor of two speedup with both `Chaff` and `BerkMin`. Doubling the weak correctness criteria to 16 produced another factor of two speedup for `BerkMin`, but a smaller speedup for `Chaff`. When verifying the more complex 9VLIW-MC-BP-EX, using 11 weak correctness criteria resulted in a factor of two speedup for both SAT-checkers, but 22 weak correctness criteria produced a negligible speedup for `Chaff` and slightly lengthened the run time for `BerkMin`. Hence, extensive decomposition has diminishing returns for complex designs, but does help reduce the CPU time. While `Chaff` was usually faster when detecting bugs in the four incorrect variants of 9VLIW-MC-BP-EX (see Table 7), especially when using a monolithic correctness criterion, `BerkMin` was approximately 3 times faster than `Chaff` when verifying the two correct designs in Table 8.

8. Impact of Conservative Approximations and Positive Equality

We previously used conservative approximations, such as:

- **Translation boxes.** These are dummy UFs or UPs with one input (Velev and Bryant, 2000) that are manually inserted before the inputs of architectural state elements in both the implementation and the specification processor. Such UFs or UPs result in common subexpression substitution, and could produce simpler Boolean correctness formulas.
- **Automatically abstracted memories.** The interpreted functions *read* and *write* are abstracted automatically (Velev, 2000a, 2001) with completely general UFs that do not satisfy the forwarding property of the memory semantics.

These conservative approximations have the potential to speed up the verification of correct designs, but might result in false negatives requiring manual intervention and analysis. When such optimizations were not used in the verification of the correct 9VLIW-MC-BP-EX, `Chaff` took 914 seconds to prove the unsatisfiability of the CNF formula, compared to 660 seconds with the optimizations; `BerkMin` took 969 seconds (longer than `Chaff`), compared to 275 seconds with the optimizations. In both cases, the verification was done with the e_{ij} encoding and monolithic evaluation of the correctness criterion. Hence, the overhead is insignificant, compared with the time to manually identify false negatives that might result from the optimizations.

We then evaluated the benefits of exploiting Positive Equality, given the extremely efficient SAT-checkers `Chaff` and `BerkMin`. This was implemented by introducing an e_{ij} Boolean variable for the equality comparison of two syntactically distinct p-term variables—as done originally by Goel, *et al.* (1998)—instead of treating such p-term variables as not equal. The results are listed in Table 9.

As Table 9 shows, when verifying the first three benchmarks, Positive Equality resulted in up to 4 orders of magnitude speedup for `Chaff`, and in up to 3 orders of magnitude speedup for `BerkMin`. When verifying the last three benchmarks that are much more complex, and when we did not use Positive Equality, `Chaff` did not complete in 24 hours for 2×DLX-CC-MC-EX-BP, and ran out of memory (given the available 4 GB) for 9VLIW-MC-BP-buggy and 9VLIW-MC-BP; `BerkMin` did not finish in 24 hours for each of these three benchmarks. In contrast, with Positive Equality, `Chaff` needed 27 MB of memory for the satisfiable CNF formula from 9VLIW-MC-BP-buggy, and 241 MB for the unsatisfiable CNF formula from 9VLIW-MC-BP.

Processor	Chaff		BerkMin	
	Positive Equality	No Positive Equality	Positive Equality	No Positive Equality
1×DLX-C-buggy	0.13	17	0.02	2
1×DLX-C	0.19	9,177	0.07	229
2×DLX-CC-MC-EX-BP-buggy	12	9,409	4	2,816
2×DLX-CC-MC-EX-BP	22	> 24 h	15	> 24 h
9VLIW-MC-BP-buggy	5	out of memory	10	> 24 h
9VLIW-MC-BP	759	out of memory	224	> 24 h

Table 9: Time for satisfiability checking with and without Positive Equality. Unless specified, the time is measured in seconds. The experiments were run on a 336-MHz Sun4 with 4 GB of physical memory.

Therefore, Positive Equality is still the main reason for the efficiency of our tool flow when formally verifying complex microprocessors. The CNF formulas generated without Positive Equality are available as benchmark suite NPE-1.0 (Velev, 2002).

9. Conclusions

We found the SAT-checkers `Chaff` (Moskewicz, *et al.*, 2001; Zhang, *et al.*, 2001) and `BerkMin` (Goldberg and Novikov, 2002) to be the most efficient for evaluating Boolean formulas generated in the formal verification of both correct and buggy microprocessors, dramatically outperforming 29 SAT-checkers, 2 ATPG tools, and 2 decision diagrams—BDDs (Bryant, 1986, 1992) and BEDs (Williams, 2000). The microprocessors were described in a high-level hardware description language (Velev and Bryant, 2001b) based on the logic of Equality with Uninterpreted Functions and Memories (EUFM), proposed by Burch and Dill (1994). The formal verification was done with Burch and Dill's correctness criterion, using flushing of the implementation processor to map its state to the state of the specification. The EUFM correctness formula was translated to an equivalent Boolean formula by exploiting Positive Equality (Bryant, German, and Velev, 2001), and using the automatic tool `EVC` (Velev and Bryant, 2001b).

Reassessing various optimizations that can be applied when generating the Boolean formulas for the microprocessor correctness, we conclude that the single most important step is exploiting Positive Equality. Without it, neither `Chaff` nor `BerkMin` would have scaled for realistic superscalar and VLIW microprocessors with exceptions, multicycle functional units, branch prediction, and other speculative features. `BerkMin` was consistently faster on unsatisfiable CNF formulas from complex correct designs, since `BerkMin` was developed after `Chaff` and was better optimized for CNF formulas derived from expressions with many levels (Goldberg, 2002).

Exploiting the e_{ij} encoding (Goel, *et al.*, 1998) of g-equations resulted in a speedup of 2 for the base VLIW processor, 9VLIW-MC-BP, compared to the small-domain encoding (Pnueli, *et al.*, 1999) when verifying correct designs, and consistently performed better on buggy versions. Although the e_{ij} encoding introduces more than twice as many primary Boolean variables for our benchmarks, it results in less CNF variables and less CNF clauses than the small-domain encoding, and produces a conceptually simpler search space—with each e_{ij} Boolean variable naturally encoding the equality between a pair of g-term variables. Transitivity of equality is enforced with fewer implications than in the case of the small-domain encoding. In contrast, the small-domain encoding enumerates all mappings of g-term variables to a sufficient set of distinct constants, thus introducing more information than actually required to solve the problem. This results in the potential to revisit portions of the search space, for what would be just one visit with the e_{ij} encoding.

Conservative approximations, such as manually inserted translation boxes (Velev and Bryant, 2000) or automatically abstracted memories (Velev, 2000a, 2001), are not as essential to the fast verification of correct VLIW and dual-issue superscalar processors when using `Chaff` or `BerkMin`, as these optimizations were when using BDDs—previously the most efficient SAT procedure for correct designs.

Decomposing the evaluation of the Boolean correctness formula (Velev, 2000a), by evaluating many simpler formulas in parallel, resulted in a speedup of up to 3.5 times for the base VLIW processor, but in a speedup of 2 for its version with exceptions. Decomposition consistently accelerated the generation of counterexamples for buggy microprocessors.

Structural variations in generating the Boolean correctness formulas—early reductions of p-equations, and using Ackermann constraints for eliminating uninterpreted predicates—as well as variations of `Chaff`'s command parameters (we could not vary `BerkMin`'s command parameters) accelerated the detection of bugs, although no single variation performed best. Again, the assumption is that we can run several parallel copies of the tool flow. Neither structural nor parameter variations accelerated the verification of the correct base VLIW processor.

Algebraic simplifications (Marques-Silva, 2000)(Brafman, 2001), or renaming the CNF variables in order to minimize the cutwidth of the formulas (Aloul, *et al.*, 2001)(Wang, *et al.*, 2001) did not result in speedups, due to the complexity of the CNF formulas generated in the formal verification of realistic microprocessors.

To conclude, we showed that `Chaff` and `BerkMin` can easily handle complex CNF formulas that are produced in microprocessor formal verification without applying conservative transformations. Such transformations were previously needed in BDD-based evaluations, but have the potential to result in false negatives, taking extensive human effort to analyze. We identified the optimizations that help increase the performance of `Chaff` and `BerkMin` on realistic dual-issue superscalar and VLIW designs—Positive Equality, combined with the e_{ij} encoding. Helpful, but not essential, are decomposed evaluation of the Boolean correctness formulas, and use of 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 (Tennenhouse, 2000). The benchmarks used in this paper are available as (Velev, 2000b, 2002).

References

W Ackermann, (1954). *Solvable Cases of the Decision Problem*, North-Holland, Amsterdam,

F A Aloul, I L Markov, K A Sakallah, (2001). Faster SAT and Smaller BDDs via Common Function Structure, *International Conference on Computer-Aided Design (ICCAD '01)*, 443–448.

L Baptista, J P Marques-Silva, (2000). Using Randomization and Learning to Solve Hard Real-World Instances of Satisfiability[12], *Principles and Practice of Constraint Programming (CP '00)*,

R J Bayardo, Jr., R Schrag, (1997). Using CSP Look-Back Techniques to Solve Real World SAT Instances, *14th National Conference on Artificial Intelligence (AAAI '97)*, 203–208.

B Bentley, (2001). Validating the Intel® Pentium® 4 Microprocessor, *38th Design Automation Conference (DAC '01)*, 244–248.

P Bjesse, T Leonard, A Mokkedem, (2001). Finding Bugs in an Alpha Microprocessor Using Satisfiability Solvers, *Computer-Aided Verification (CAV '01)*, G Berry, H Comon, A Finkel, *eds.*, **LNCS 2102**, Springer-Verlag, 454–464.

R I Brafman, (2001). A Simplifier for Propositional Formulas with Many Binary Clauses, *International Joint Conference on Artificial Intelligence (IJCAI '01)*, 515–520.

F Brglez, H Fujiwara, (1985). A Neutral Netlist of 10 Combinational Benchmark Circuits[13], *International Symposium on Circuits and Systems (ISCAS '85)*,

F Brglez, D Bryan, K Kozminski, (1989). Combinational Profiles of Sequential Benchmark Circuits[13], *International Symposium on Circuits and Systems (ISCAS '89)*,

[12]Available from: `http://sat.inesc.pt/~jpms`.

[13]Available from: `http://cbl.ncsu.edu/benchmarks`.

R E Bryant, (1986). Graph-Based Algorithms for Boolean Function Manipulation, *IEEE Transactions on Computers*, **C-35(8)**, 677–691.

R E Bryant, (1992). Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams, *ACM Computing Surveys*, **24(3)**, 293–318.

R E Bryant, S German, M N Velev, (2001). Processor Verification Using Efficient Reductions of the Logic of Uninterpreted Functions to Propositional Logic[14], *ACM Transactions on Computational Logic (TOCL)*, **2(1)**, 93–134.

R E Bryant, M N Velev, (2002). Boolean Satisfiability with Transitivity Constraints[14], *ACM Transactions on Computational Logic (TOCL)*, **3(4)**,

J R Burch, D L Dill, (1994). Automated Verification of Pipelined Microprocessor Control, *Computer-Aided Verification (CAV '94)*, D L Dill, *ed.*, **LNCS 818**, Springer-Verlag, 68–80.

J R Burch, (1996). Techniques for Verifying Superscalar Microprocessors, *33rd Design Automation Conference (DAC '96)*, 552–557.

P Chatalic, L Simon, (2000). Multi-Resolution on Compressed Sets of Clauses, *12th International Conference on Tools with Artificial Intelligence (ICTAI '00)*, 2–10.

E Clarke, A Biere, R Raimi, Y Zhu, (2001). Bounded Model Checking Using Satisfiability Solving, *Journal on Formal Methods in System Design (FMSD)*, **19(1)**, 7–34.

F Copty, L Fix, R Fraer, E Giunchiglia, G Kamhi, A Tacchella, M Y Vardi, (2001). Benefits of Bounded Model Checking at an Industrial Setting, *Computer-Aided Verification (CAV '01)*, G Berry, H Comon, A Finkel, *eds.*, **LNCS 2102**, Springer-Verlag, 436–453.

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

M Davis, H Putnam, (1960). A Computing Procedure for Quantification Theory, *Journal of the ACM*, **7(3)**, 201–215.

M Davis, G Logemann, D Loveland, (1962). A Machine Program for Theorem Proving, *Communications of the ACM*, **5(7)**, 394–397.

[14]Available from: `http://www.ece.cmu.edu/~mvelev`.

O Dubois, P Andre, Y Boufkhad, J Carlier, (1993). Can a Very Simple Algorithm Be Efficient for SAT?[15], *The Second DIMACS Implementation Challenge, DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, D S Johnson, M A Trick, *eds.*,

J W Freeman, (1995). *Improvements to Propositional Satisfiability Search Algorithms*, Ph.D. Thesis, Department of Computer and Information Science, University of Pennsylvania,

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

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

E Goldberg, (2002). Personal Communication,

C P Gomes, B Selman, N Crator, H A Kautz, (2000). Heavy-Tailed Phenomena in Satisfiability and Constraint Satisfaction Problems, *Journal of Automated Reasoning*, **24(1–2)**, 67–100.

J F Groote, J P Warners, (2000). The Propositional Formula Checker HeerHugo, *Journal of Automated Reasoning*, **24(1–2)**, 101–125.

I Hamzaoglu, J H Patel, (1999). New Techniques for Deterministic Test Pattern Generation, *Journal of Electronic Testing: Theory and Applications*, **15(1–2)**, 63–73.

J E Harlow III, F Brglez, (2001). Design of Experiments and Evaluation of BDD Ordering Heuristics, *International Journal on Software Tools for Technology Transfer (STTT)*, **3(2)**, 193–206.

J L Hennessy, D A Patterson, (2002). *Computer Architecture: A Quantitative Approach, 3rd edition*, Morgan Kaufmann Publishers, San Francisco, CA,

E A Hirsch, A Kojevnikov, (2001). Solving Boolean Satisfiability Using Local Search Guided by Unit Clause Elimination, *Principles and Practice of Constraint Programming (CP '01)*, 605–609.

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

[15]Available from:
`ftp://dimacs.rutgers.edu/pub/challenge/satisfiability/contributed/dubois`.

Intel Corporation, (1999). IA-64 Application Developer's Architecture Guide[16],

K Iwama, H Abeta, E Miyano, (1992). 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., 322–328.

K Iwama, K Hino, (1994). Random Generation of Test Instances for Logic Optimizers, *31st Design Automation Conference (DAC '94)*, 430–434.

G Janssen, (2001). Design of a Pointerless BDD Package, *10th Int'l. Workshop on Logic & Synthesis (IWLS '01)*,

D S Johnson, M A Trick, *eds.*, (1993). The Second DIMACS Implementation Challenge. DIMACS Series in Discrete Mathematics and Theoretical Computer Science. Available from: `http://dimacs.rutgers.edu/Challenges`,

R B Jones, (2002). *Symbolic Simulation Methods for Industrial Formal Verification*, Kluwer Academic Publishers, Boston/Dordrecht/London,

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

T Larrabee, (1992). Test Pattern Generation Using Boolean Satisfiability, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, **11(1)**, 4–15.

C M Li, Anbulagan, (1997). Heuristics Based on Unit Propagation for Satisfiability Problems, *International Joint Conference on Artificial Intelligence (IJCAI '97)*, 366–371.

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

I Lynce, L Baptista, J P Marques-Silva, (2001). Stochastic Systematic Search Algorithms for Satisfiability, *LICS Workshop on Theory and Applications of Satisfiability Testing (LICS-SAT)*,

S Malik, A R Wang, R K Brayton, A Sangiovani-Vincentelli, (1988). Logic Verification Using Binary Decision Diagrams in a Logic Synthesis Environment, *International Conference on Computer-Aided Design (ICCAD '88)*, 6–9.

[16]Available from: `http://developer.intel.com/design/ia-64/architecture.htm`.

J P Marques-Silva, K A Sakallah, (1999). GRASP: A Search Algorithm for Propositional Satisfiability, *IEEE Transactions on Computers*, **48(5)**, 506–521.

J P Marques-Silva, (1999). The Impact of Branching Heuristics in Propositional Satisfiability Algorithms[12], *9th Portuguese Conference on Artificial Intelligence (EPIA)*,

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

J P Marques-Silva, (2000). Algebraic Simplification Techniques for Propositional Satisfiability[12], *Principles and Practice of Constraint Programming (CP '00)*, 537–542.

S.-I Minato, (1996). *Binary Decision Diagrams and Applications for VLSI CAD*, Kluwer Academic Publishers, Boston/Dordrecht/London,

S.-I Minato, (2001). Zero-Suppressed BDDs and Their Applications, *International Journal on Software Tools for Technology Transfer (STTT)*, **3(2)**, 156–170.

D Mitchell, B Selman, H Levesque, (1992). Hard and Easy Distributions of SAT Problems, *10th National Conference on Artificial Intelligence (AAAI '92)*, 459–465.

M W Moskewicz, (2001). Personal Communication,

M W Moskewicz, C F Madigan, Y Zhao, L Zhang, S Malik, (2001). Engineering a Highly Efficient SAT Solver, *38th Design Automation Conference (DAC '01)*, 530–535.

S Pilarski, G Hu, (2002). SAT with Partial Clauses and Back-Leaps, *39th Design Automation Conference (DAC '02)*, 743–746.

D A Plaisted, A Biere, Y Zhu, (2002). A Satisfiability Procedure for Quantified Boolean Formulae, *Discrete Applied Mathematics, Renesse Special Issue Devoted to the International Symposium on Theory and Applications of Satisfiability Testing (SAT '00)*, P Hammer, *ed.*,

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

S D Prestwich, (2000). Stochastic Local Search in Constrained Spaces, *Practical Application of Constraint Technology and Logic Programming (PACLP '00)*, 27–39.

J Rintanen, (1999). Improvements to the Evaluation of Quantified Boolean Formulae, *International Joint Conference on Artificial Intelligence (IJCAI '99)*, 1192–1197.

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

B Selman, H Kautz, B Cohen, (1996). Local Search Strategies for Satisfiability Testing, *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, **26**, 521–532.

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

Y Shang, B W Wah, (1998). A Discrete Lagrangian-Based Global-Search Method for Solving Satisfiability Problems[17], *Journal of Global Optimization*, **12(1)**, 61–99.

H Sharangpani, K Arora, (2000). Itanium Processor Microarchitecture, *IEEE Micro*, **September–October**, 24–43.

F Somenzi, (2001). Efficient Manipulation of Decision Diagrams, *International Journal on Software Tools for Technology Transfer (STTT)*, **3(2)**, 171–181.

G Stålmarck, (1989). *A System for Determining Propositional Logic Theorems by Applying Values and Rules to Triplets that are Generated from a Formula*, Swedish Patent No. 467076 (approved 1992), U.S. Patent No. 5276897 (1994), European Patent No. 0403454 (1995),

P Tafertshofer, A Ganz, K J Antreich, (2000). GRAINE—An Implication GRaph-bAsed engINE for Fast Implication, Justification, and Propagation, *IEEE Transactions on CAD*, **19(8)**, 907–927.

D Tennenhouse, (2000). Proactive Computing, *Communications of the ACM*, **43(5)**, 43–50.

D Van Campenhout, H Al-Asaad, J P Hayes, T Mudge, R B Brown, (1998). High-Level Design Verification of Microprocessors via Error Modeling, *ACM Transactions on Design Automation of Electronic Systems*, **3(4)**, 581–599.

[17] Available from: `http://manip.crhc.uiuc.edu`.

D Van Campenhout, T Mudge, J P Hayes, (2000). Collection and Analysis of Microprocessor Design Errors, *IEEE Design & Test of Computers*, **17(4)**, 51–60.

M N Velev, R E Bryant, (1998a). Incorporating Timing Constraints in the Efficient Memory Model for Symbolic Ternary Simulation[14], *International Conference on Computer Design (ICCD '98)*, 400–406.

M N Velev, R E Bryant, (1998b). Bit-Level Abstraction in the Verification of Pipelined Microprocessors by Correspondence Checking[14], *Formal Methods in Computer-Aided Design (FMCAD '98)*, G Gopalakrishnan, P Windley, *eds.*, **LNCS 1522**, Springer-Verlag, 18–35.

M N Velev, (1999). Benchmark Suites[14] SSS.1.0, SSS.1.0a.,

M N Velev, R E Bryant, (1999). Superscalar Processor Verification Using Efficient Reductions of the Logic of Equality with Uninterpreted Functions to Propositional Logic[14], *Correct Hardware Design and Verification Methods (CHARME '99)*, L Pierre, T Kropf, *eds.*, **LNCS 1703**, Springer-Verlag, 37–53.

M N Velev, R E Bryant, (2000). Formal Verification of Superscalar Microprocessors with Multicycle Functional Units, Exceptions, and Branch Prediction[14], *37th Design Automation Conference (DAC '00)*, 112–117.

M N Velev, (2000a). Formal Verification of VLIW Microprocessors with Speculative Execution[14], *Computer-Aided Verification (CAV '00)*, E A Emerson, A P Sistla, *eds.*, **LNCS 1855**, Springer-Verlag, 296–311.

M N Velev, (2000b). Benchmark Suites[14] SSS-SAT.1.0, VLIW-SAT.1.0, FVP-UNSAT.1.0, and FVP-UNSAT.2.0,

M N Velev, (2001). Automatic Abstraction of Memories in the Formal Verification of Superscalar Microprocessors[14], *Tools and Algorithms for the Construction and Analysis of Systems (TACAS '01)*, T Margaria, W Yi, *eds.*, **LNCS 2031**, Springer-Verlag, 252–267.

M N Velev, R E Bryant, (2001a). Effective Use of Boolean Satisfiability Procedures in the Formal Verification of Superscalar and VLIW Microprocessors[14], *38th Design Automation Conference (DAC '01)*, 226–231.

M N Velev, R E Bryant, (2001b). EVC: A Validity Checker for the Logic of Equality with Uninterpreted Functions and Memories, Exploiting Positive Equality and Conservative Transformations[14], *Computer-Aided Verification (CAV '01)*, G Berry, H Comon, A Finkel, *eds.*, **LNCS 2102**, Springer-Verlag, 235–240.

M N Velev, (2002). Benchmark Suite[14] NPE-1.0,

D Wang, E Clarke, Y Zhu, J Kukula, (2001). Using Cutwidth to Improve Symbolic Simulation and Boolean Satisfiability, *IEEE International High Level Design Validation and Test Workshop (HLDVT '01)*,

P F Williams, A Biere, E M Clarke, A Gupta, (2000). Combining Decision Diagrams and SAT Procedures for Efficient Symbolic Model Checking[18], *Computer-Aided Verification (CAV '00)*, E A Emerson, A P Sistla, *eds.*, **LNCS 1855**, Springer-Verlag, 124–138.

P F Williams, (2000). *Formal Verification Based on Boolean Expression Diagrams*[18], Ph.D. Thesis, Department of Information Technology, Technical University of Denmark, Lyngby, Denmark,

Z Wu, B W Wah, (1999). Solving Hard Satisfiability Problems: A Unified Algorithm Based on Discrete Lagrange Multipliers[17], *11th IEEE International Conference on Tools with Artificial Intelligence (ICTAI '99)*, 210–217.

H Zhang, (1997). SATO: An Efficient Propositional Prover, *Int'l. Conference on Automated Deduction (CADE '97)*, **LNAI 1249**, Springer-Verlag, 272–275.

L Zhang, C F Madigan, M W Moskewicz, S Malik, (2001). Efficient Conflict Driven Learning in a Boolean Satisfiability Solver, *International Conference on Computer-Aided Design (ICCAD '01)*, 279–285.

[18]Available from: `http://www.it-c.dk/research/bed`.