Beaver: An SMT Solver for Quantifier-free Bit-vector Logic

Rhishikesh Shrikant Limaye
Sanjit A. Seshia

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2010-67
http://www.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-67.html

May 13, 2010

Copyright © 2010, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

Acknowledgement

I thank my advisor, Professor Sanjit Seshia, and a fellow colleague student, Susmit Jha, with whom I conducted most of this work. Part of this report appears in a paper jointly written with Susmit Jha and Prof. Seshia -- Susmit Jha, Rhishikesh Limaye, and Sanjit Seshia. Beaver: Engineering an efficient SMT solver for bit-vector arithmetic. In Computer Aided Verification, pages 668--674. 2009.

Beaver: An SMT Solver for Quantifier-free Bit-vector Logic

Rhishikesh Limaye

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

Professor Sanjit Seshia
Research Advisor

Date

* * * * * *

Professor Rastislav Bodik
Second Reader

Date

Contents

1 Introduction **1**
1.1 Theory of Bit-vectors and QF_BV Logic 2
1.2 Bit-vector SMT Solvers 2

2 Beaver Design and Implementation **6**
2.1 Word-level Simplifications 8
2.1.1 Generic Structural Simplifications 9
2.1.2 Constant Propagation 10
2.1.3 Don't-care Propagation 12
2.1.4 Symbolic / Structural Evaluation 12
2.1.5 Equality Propagation 14
2.1.6 Division / Remainder Simplification 15
2.2 Event-driven, On-the-fly Simplification 16
2.3 Bit-blasting and Mapping to SAT 18

3 Experiments using SMT-LIB benchmarks **20**
3.1 Effectiveness of simplifications in Beaver 22

4 Conclusion **28**

Abstract

This thesis presents Beaver – an efficient SMT solver for the quantifier-free fixed-size bit-vector logic (QF_BV). Beaver is an eager solver, that is, given an SMT formula, it first performs word-level simplications and then bitblasts the simplified formula to a Boolean formula, which is then solved using any SAT solver. Several engineering techniques are behind its efficiency: 1) efficient constant/constraint propagation using event-driven approach, 2) several word-level rewrite rules, 3) efficient bitblasting to SAT, by first converting to and-inverter-graph (AIG) representation and using Boolean simplication techniques of ABC logic synthesis system [33].

In this thesis, we highlight the implementation details of Beaver that distinguishes it from other solvers. We also present an experimental evaluation and analysis of the effectiveness of our solver against all available QF_BV solvers on the SMT-LIB benchmark suite.

Beaver is an open-source tool implemented in OCaml, usable with any back-end SAT engine, and has a well-documented extensible code base that can be used to experiment with new algorithms and techniques.

Chapter 1

Introduction

The Boolean satisfiability (SAT) problem is defined as checking whether a given Boolean formula is satisfiable, that is, whether there exists a Boolean assignment to the inputs that make the formula true. Although it is a classic NP-complete problem, in practice, SAT solving techniques have improved dramatically over the last decade. Several fast SAT solvers, capable of handling significantly large problem instances, are easily available today. Still, SAT solvers perform rather low-level reasoning with only Boolean variables, whereas most problem domains have higher-level information and data types such as integers, reals, finite precision (e.g. 32-bit) integers (i.e. bit-vectors), arrays, strings, lists, and so on.

The problem of *Satisfiability Modulo Theories (SMT)* concerns satisfiability of a formula made up of not just Boolean variables and operators, but also those from other first-order theories. Most obvious ways of encoding of such formulas to Boolean SAT are often too expensive. Hence the idea is to apply higher-level reasoning, instead of or before handing it down to SAT. Additionally, SMT provides generic ways to *combine* multiple theories in a single problem instance. And it gives guidance in *defining* new theories specific for one's problem domain. More rigorous exposition to the field of *SMT solvers*, or *decision procedures* can be obtained from a recent book [25], and a book chapter [5].

Although researched since 70's, decision procedures and SMT solvers have seen especially dramatic improvements in their practical capabilities in the last decade. This has been a consequence and a cause of developments in multitude of applications of SMT. SMT solvers are one of the core components of tools in several areas, such as formal verification of hardware, static

and dynamic program analysis, software test generation, software synthesis, and software security.

1.1 Theory of Bit-vectors and QF_BV Logic

A bit-vector is a sequence of bits. The size of a bit-vector is the length of this sequence. For example, `0110` is a bit-vector of length 4. A fixed-size bit-vector is a bit-vector of a known constant size. The theory of fixed-size bit-vectors [32] is made of variables and constants of arbitrary but fixed sizes, and functions and predicates operating on them. The quantifier-free bit-vector (QF_BV) logic [30] comprises of formulas built using the theory of bit-vectors and Boolean connectives, constants, and variables. These formulas can be expressed using the grammar of Figure 1.1. This grammar contains all the operations defined by the SMT-LIB standard, and adds a few extra operations. An example formula – `(bvule (bvadd a b) bv7[32])`, where `a, b` are 32-bit variables – is equivalent a C expression `a + b <= 7`, where `a, b` are unsigned ints. Bit-vector logic provides more accurate modeling of the finite-precision implementation details as compared to integer logic.

1.2 Bit-vector SMT Solvers

Contemporary QF_BV solvers can be listed quickly by looking at the yearly SMT competition webpages – SMT-COMP'07 [27], SMT-COMP'08 [28], SMT-COMP'09 [29]. Apart from the solvers described herein, currently available solvers are Boolector, Z3, MathSAT, STP, OpenSMT, Yices, Spear, and CVC3.

Currently all bit-vector SMT solvers ultimately rely on *bit-blasting*. Bit-blasting means encoding a word-level, QF_BV formula to an equisatisfiable, Boolean representation by representing bit-vector variables as a string of Boolean variables and encoding bit-vector operations using their corresponding Boolean circuits. This Boolean formula is then solved using a SAT solver. Thus, bit-vector SMT is still heavily dependent on SAT solvers. However, word-level simplifications done before bit-blasting do make a vital difference in the performance of the solver. And that is where all the SMT solvers differ.

Spear [2,3] is a bit-vector-only solver that won SMT-COMP'07 in QF_BV

$$
\begin{aligned}
\psi \;::=\;& p \qquad\qquad p \in P \quad \text{(set of Boolean variables)}\\
&\mid \textbf{true} \mid \textbf{false}\\
&\mid \textbf{not}\ \psi \mid \textbf{and}\ \psi^+ \mid \textbf{or}\ \psi^+ \mid \textbf{xor}\ \psi^+ \mid \textbf{iff}\ \psi^+\\
&\mid \textbf{implies}\ \psi_1\ \psi_2 \mid \textbf{if_then_else}\ \psi_1\ \psi_2\ \psi_3\\
&\mid =\ \tau^+ \mid \textbf{distinct}\ \tau^+\\
&\mid \textbf{bvule}\ \tau_1\ \tau_2 \mid \textbf{bvult}\ \tau_1\ \tau_2 \mid \textbf{bvuge}\ \tau_1\ \tau_2 \mid \textbf{bvugt}\ \tau_1\ \tau_2\\
&\mid \textbf{bvsle}\ \tau_1\ \tau_2 \mid \textbf{bvslt}\ \tau_1\ \tau_2 \mid \textbf{bvsge}\ \tau_1\ \tau_2 \mid \textbf{bvsgt}\ \tau_1\ \tau_2
\end{aligned}
$$

$$
\begin{aligned}
\tau \;::=\;& v \qquad\qquad v \in V \quad \text{(set of bit-vector variables)}\\
&\mid b \qquad\qquad b \in B \quad \text{(set of bit-vector numerals)}\\
&\mid \textbf{ite}\ \psi\ \tau_1\ \tau_2\\
&\mid \textbf{bvadd}\ \tau^+ \mid \textbf{bvmul}\ \tau^+ \mid \textbf{bvsub}\ \tau_1\ \tau_2 \mid \textbf{bvneg}\ \tau\\
&\mid \textbf{bvuaddp}\ \tau^+ \mid \textbf{bvumulp}\ \tau^+\\
&\mid \textbf{bvuaddo}\ \tau^+ \mid \textbf{bvumulo}\ \tau^+\\
&\mid \textbf{bvuaddno}\ \tau^+ \mid \textbf{bvumulno}\ \tau^+\\
&\mid \textbf{bvudiv}\ \tau_1\ \tau_2 \mid \textbf{bvurem}\ \tau_1\ \tau_2\\
&\mid \textbf{bvsdiv}\ \tau_1\ \tau_2 \mid \textbf{bvsrem}\ \tau_1\ \tau_2 \mid \textbf{bvsmod}\ \tau_1\ \tau_2\\
&\mid \textbf{bvnot}\ \tau \mid \textbf{bvand}\ \tau^+ \mid \textbf{bvor}\ \tau^+ \mid \textbf{bvxor}\ \tau^+\\
&\mid \textbf{bvnand}\ \tau^+ \mid \textbf{bvnor}\ \tau^+ \mid \textbf{bvxnor}\ \tau^+\\
&\mid \textbf{bvredor}\ \tau \mid \textbf{bvredand}\ \tau\\
&\mid \textbf{bvshl}\ \tau_1\ \tau_2 \mid \textbf{bvlshr}\ \tau_1\ \tau_2 \mid \textbf{bvashr}\ \tau_1\ \tau_2\\
&\mid \textbf{shift_left}[n]\ \tau \mid \textbf{shift_right_logical}[n]\ \tau \mid \textbf{shift_right_arith}[n]\ \tau\\
&\mid \textbf{concat}\ \tau^+ \mid \textbf{extract}[i:j]\ \tau \mid \textbf{repeat}[n]\ \tau\\
&\mid \textbf{zero_extend}[n]\ \tau \mid \textbf{sign_extend}[n]\ \tau\\
&\mid \textbf{rotate_left}[n]\ \tau \mid \textbf{rotate_right}[n]\ \tau
\end{aligned}
$$

Figure 1.1: Grammar for formulas in QF_BV. ψ denotes formulas, τ denotes terms, which are of type BitVec[n], where n is bit-width. Not shown are the typing constraints on the terms (e.g. **bvadd** takes arguments of identical bit-width, and produces the result of the same bit-width). **bvuaddp** etc. are our extensions to the SMT-LIB grammar, and their meanings are listed in Table 1.1.

Function	Meaning
bvuaddp	Precise unsigned addition, i.e. if inputs are of bitwidth w, the output has $w + 1$ bits to accomodate the full result.
bvumulp	Precise unsigned multiplication, i.e. if inputs are of bitwidth w, the output has $2w$ bits to accomodate the full result.
bvuaddo	One-bit output equal to the overflow bit of unsigned addition of inputs. Also present in Boolector.
bvumulo	One-bit output equal to the overflow bit of unsigned multiplication of inputs. Also present in Boolector.
bvuaddno	Unsigned addition with implicit assumption that no overflow occurs. This is equivalent to normal **bvadd** plus top-level inequality assumption of no overflow during addition.
bvumulno	Unsigned multiplication with implicit assumption that no overflow occurs.

Table 1.1: Extensions to SMT-LIB QF_BV logic.

category. It is based on bit-blasting using several word-level simplification rules and a specialized, fast SAT solver with numerous optimization parameters. The parameters are tuned for a particular class of benchmarks using automatic exploration of the parameter value space [20].

STP [19] is a decision procedure for both bit-vector arithmetic and the theory of arrays that is especially geared towards solving path feasibility queries in software verification and testing. The main novelty of STP's bit-vector component was an online solver for general linear constraints modulo a power of two. Our experience with Beaver on such path feasibility queries indicates that the full power of linear constraint solving is not quite needed for practical formulas.

Boolector [8] uses word-level rewrites, followed by bit-blasting to PicoSAT [7] with the use of under-approximation techniques that rely strongly on the connection to PicoSAT. This well-optimized implementation earned Boolector 1st spot in SMT-COMP'08.

Z3 [16] is a feature-rich SMT solver from Microsoft that supports multiple theories, and also quantifiers to some extent. Not many details are available regarding its bit-vector handling. MathSAT [9, 10] is a lazy, layered SMT

solver for multiple theories. Its bit-vector performance improved dramatically for SMT-COMP'09. Other solvers in the lazy, multi-theory category are OpenSMT [11], Yices [17], and CVC3 [6]. Roughly, these solvers use some word-level simplifications followed by bit-blasting that adds new clauses to the core DPLL engine.

UCLID [13, 14], precursor to Beaver, uses an abstraction-refinement approach to solving bit-vector formulas, into which any model-generating SMT solver for QF_BV, including Beaver, can be easily integrated.

A few other solvers have been developed for specific domains, but are not directly usable for solving generic SMT formulas coming from multiple applications. Case in point is BAT [26], which focuses on hardware verification problems. It is in fact quite similar to Beaver in theory, however a few differences can be noted. It uses bit-blasting, but to an internal Boolean DAG format called NICE, from which efficient CNF generation is employed. In contrast with BAT, Beaver uses an and-inverter graph back-end with circuit optimization techniques drawn from the logic synthesis literature, as well as offline template optimizations (described in the following section), which is an automated optimization, distinct from the use of user-defined functions in BAT.

One distinguishing feature of Beaver is that it can use any off-the-shelf SAT solver, including circuit-based SAT solvers, e.g. NFLSAT [22], . In fact, currently NFLSAT gives us the best performance as compared to other CNF-based SAT solvers. This ability to switch SAT solvers is absent in all other SMT solvers, as they integrate one specific, often specialized, implementation of a SAT solver in a tight coupling with the word-level processing. Freedom to switch a SAT solver is important because for different formulas, different SAT heuristics work better.

Finally, during last year there have been attempts to move away from bit-blasting. One partial solution is to restrict to extractions and concatanations of bit-vectors, which are easily translated to equality logic [12]. This would be useful for formulas that make heavy use of these operations as compared to other operations of bit-vector theory. Alternatively, [4] proposes a direct way of solving generic bit-vector constraints without using SAT. However, it only improves upon the previous similar attempts that used constraint programming, but still does not prove to be efficient alternative to bit-blasting+SAT.

Further reading about current and past bit-vector decision procedures can be found in [14].

Chapter 2

Beaver Design and Implementation

Being an eager SMT solver, Beaver is essentially a compiler from word-level, bit-vector SMT formula to Boolean SAT formula. Thus, like any compiler, it has a pipe-and-filter architecture as shown in Figure 2.1. In this chapter, we give the algorithms and implementation tricks underlying this architecture.

Beaver accepts as input a QF_BV formula in SMT-LIB syntax [31] [1]. The parser builds a formula graph by calling the API functions of the formula graph data structure. The simplifier performs word-level simplifications on the formula graph. Although builder and simplifier are shown as separate stages in the figure, they are in fact interleaved, with simplification happening *on-the-fly*. This is important, because otherwise the intermediate, unoptimized formula graph would be too large to deal with. After the simplifications are done, bit-blaster converts the formula into a combinational And-Inverter-Graph (AIG) circuit using the ABC logic synthesis package [33]. Then we have the option of performing further simplifications on this Boolean circuit. And in the end, the circuit is passed to an off-the-shelf SAT solver either natively as AIG (if it is a circuit-based SAT solver), or by conversion to CNF (if it is a CNF-based SAT solver).

A natural representation for an SMT formula is a DAG – directed acyclic graph. The vertices represent the sub-expressions of the formula, and are labeled with the operator type (e.g. **bvmul**, **bvadd**, **and**, **not**), and sort

[1] Only version 1.2 is supported at this time. Support for newly proposed version 2.0 will be added later.

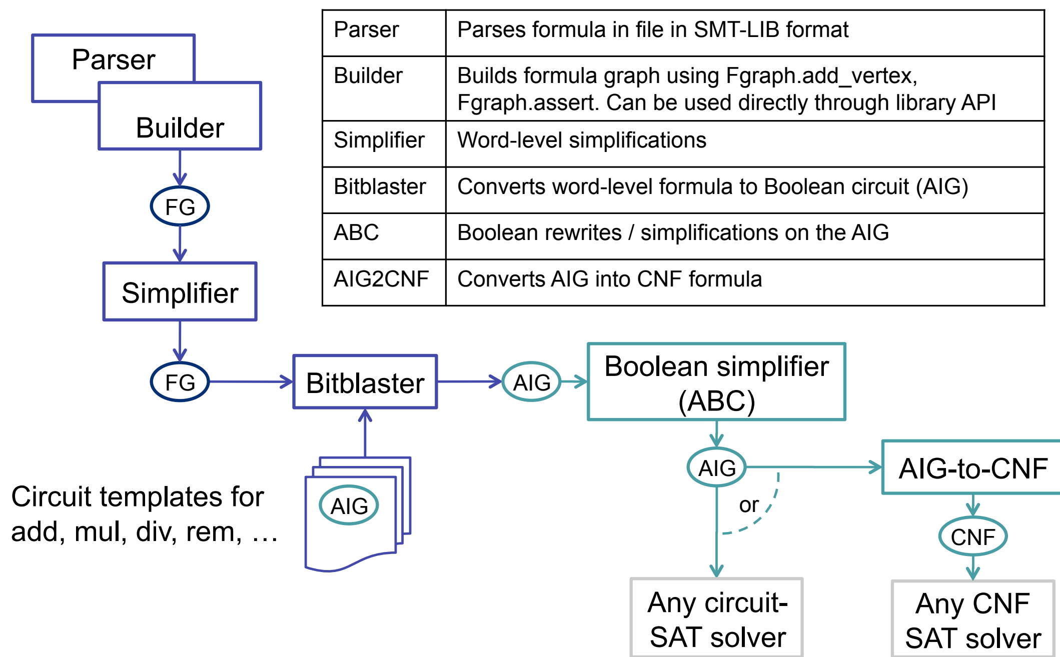


Figure 2.1: Beaver top-level flow

(etc. **BitVec[32]**, **Bool**), and a few other attributes. Each vertex has zero or more inputs (fan-ins), and zero or more outputs (fan-outs). The input variables of the formula (":extrafuns", ":extrapreds" of SMT-LIB) are the only vertices (excluding the constant numerals) with no inputs. The top-level assertions (":assertion", ":formula" of SMT-LIB) have no fan-outs, and are called roots of the formula graph.

The formula graph is implemented as a generic graph data structure having the same interface as a widely-used graph library – "ocamlgraph" [15]. This allows the use of generic graph algorithms of "ocamlgraph" directly on our formula graph.

In the rest of the chapter, we present all the word-level simplifications, event-driven simplifier, bit-blasting, and the few remaining details of mapping onto SAT.

2.1 Word-level Simplifications

Word-level simplifications are crucial in any bit-vector SMT solver. In Beaver, we have implemented a range of simplifications that are listed below.

- Common sub-expression elimination (that is, structural hashing)
- Associativity
- Commutativity
- Forward constant propagation
- Backward constant propagation
- Don't-care propagation
- Forward symbolic evaluation
- Equality propagation
- Div/rem rewrite
- Multiplication rewrite

All of the above simplifications should always be performed, and it does not matter in what order they are performed. Hence, conceptually, the simplifier is a loop that repeatedly performs the simplifications until no more are possible. Keep in mind that some simplifications, such as commutativity, have a canonical form so that we do not keep on applying them in an infinite loop. Although conceptually simple, the simplifier loop needs a few implementation tricks to be efficient, and we use a technique called *event-driven simplification* as described in Section 2.2.

Some of the simplifications that are not implemented in Beaver are as follows.

- **Canonization**: this is the process of expressing all the supported operators using a few canonical ones. For example, all the Boolean connectives can be expressed using just **and**, **not**. Or, the unsigned bit-vector predicates can be expressed as Boolean combinations of **bvult** and =. Advantage of canonization is that the simplifications need to worry about only the minimal set of operators, while a disadvantage is that some simplifications might be skipped because atomic expressions are converted to more complex ones. In Beaver, we do not use canonization at word-level, but it is used at Boolean level through AIG representation in ABC.
- **Solving general linear equalities**: general bit-vector linear equalities can be solved using a modified Gaussian elimination that takes care of modulo-2^n multiplication, addition and equality. However, we did not implement this general procedure. Some simple equalities do get simplified by backward propagation and equality propagation.
- **Fan-out-free cone elimination**: a fan-out-free cone in a formula graph is a subgraph in which every vertex has at most one fan-out, except for the vertex at the root (i.e. output) of the cone. Note that every function or predicate in our logic has at least one input combination for every possible output value. Thus, if we assert any value at the output of a fan-out-free cone, we can satisfy the cone trivially by choosing apropriate values for the internal vertices of the cone. This means that only the outputs of fan-out-free cones need to be visible to the SAT solver. This can help in cutting down the SAT formula size, and thus affect the search heuristics.

2.1.1 Generic Structural Simplifications

Common sub-expression elimination is an obvious simplification technique for formula graphs. It is also called *structural hashing* in the logic synthesis world. The basic operation involves combining two vertices of identical attributes and inputs, as shown in Figure 2.2. Associativity optimization uses associativity property of some bit-vector operators (e.g. **bvadd**, **bvmul**) to reduce expressions involving only one kind of operator to a canonical form,

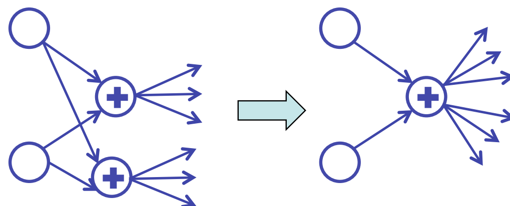


Figure 2.2: Common sub-expression elimination (i.e. structural hashing)

e.g. `(bvadd (bvadd a (bvadd b c)) d)` is reduced to `(bvadd a b c d)`. Commutativity optimization also reduces certain expressions to canonical form based on commutativity of the operators. Note that we make use of unique ordering of vertices based on their integral IDs, so that `(bvadd a b)` is chosen as canonical over `(bvadd b a)`, if `a` has lower vertex ID than `b`.

One local application of one of these simplifications creates further opportunities of simplification. These simplifications do make a vital difference in the performance of SAT solver after bit-blasting, because proving a formula such as $a + (b + c) = (b + a) + c$ using a SAT solver is not necessarily trivial when a, b, c have large bit-widths.

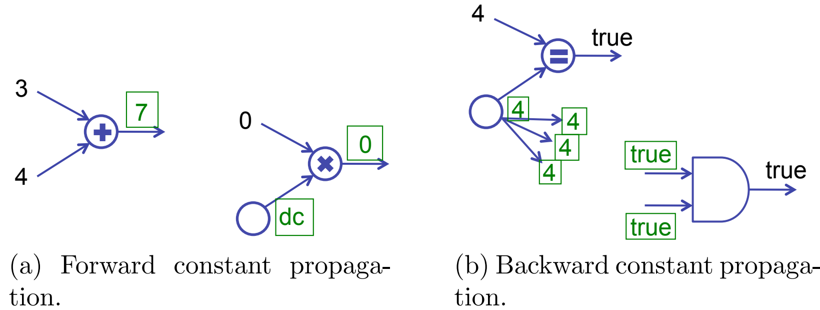
2.1.2 Constant Propagation

Forward constant propagation means inferring a constant value (that is, concrete value) at the output of a vertex when some or all of its inputs have constant values. Backward constant propagation means inferring constant values at some or all inputs of a vertex when its output is constrained to be equal to a constant value. Thus, each constant propagation rule looks at a single vertex and values at its output and inputs. Figure 2.3 illustrates these simplifications.

Forward constant propagation

The simplest forward propagation happens when all inputs of a vertex have constant values. It is then trivial to evaluate the vertex based on its type and infer a constant value at the output.

Controlling values for some functions and predicates allow inference even if some inputs do not have a constant value. For example, “false” is the controlling value for **and**, “zero” is the controlling value for **bvmul**. Another



(a) Forward constant propagation.

(b) Backward constant propagation.

Figure 2.3: Constant propagation. Black values denote existing values of vertices, and green, boxed values denote new inferred values. “dc” denotes don’t-care attribute.

Vertex type	Controlling value
and	false
or	true
implies	first input false
bvmul	zero
bvurem, bvsrem, bvsmod	second input bv1
bvlshr, bvashr, bvshl	first input zero

Table 2.1: Controlling values

effect of controlling value is that, the non-controlling inputs become *don’t-cares* (see Section 2.1.3). The list of (almost) all controlling values is given in Table 2.1.

Backward Constant Propagation

If for a function or predicate, a particular constant value is generated by exactly one input vector, then inputs are inferred when the output is constrained to have that value. Examples are **true** for **and**, **false** for **or**, **false** for **implies**. For the following functions any constant output value can be propagated backward: **concat**, **zero$_e$xtend**, **sign$_e$xtend**.

Another form of backward propagation results when the output value and some input values are used to infer value at another input. Example is illustrated by the following rule for **bvmul**.

$$\frac{n \coloneqq n_0 \times_w n_1 \qquad value(n) = c \qquad value(n_0) = c_0 \qquad c_0 \text{ odd}}{value(n_1) = c \times_w (c_0)^{-1_w}}$$

where, $\times_w$ denotes **bvmul** of bit-width w, and x^{-1_w} computes multiplicative inverse of x modulo 2^w. This backward propagation for **bvmul** does the job of solving simple linear equalities. For example, an assertion "(= (bvmul x 7) 3)" is solved to $x = 37$, if the bit-width of x is 8.

2.1.3 Don't-care Propagation

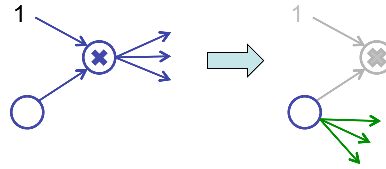
Marking a vertex as don't-care means that the value of the vertex does not affect satisfiability or unsatisfiability of the formula. Vertices unreachable from the formula graph roots fall trivially into this category. They comprise of *dead code* in the formula. Additionally, during constant propagation, inputs of certain vertices might be marked as don't-cares. Conceptually, the don't-care attributes propagate backwards through the formula graph according to the following rule.

> If all fan-outs of a vertex are don't-cares, then the vertex is marked as don't-care.

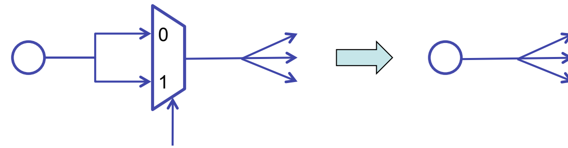
However, instead of implementing this propagation as it is, we use a different approach as described in Section 2.2.

2.1.4 Symbolic / Structural Evaluation

This is a type of forward propagation in which a particular combination of input values, or structure of input vertices allows inferring that the output has a constant value, or is equal to one of the inputs. Existence of identities for certain functions and predicates enables such inferences as listed in Table 2.2. The way to read the table is as follows: if for example there is vertex `n := (bvmul n1 bv1)`, then inference is that "n = n1", and hence all fan-outs of "n" are transferred to "n1", that is wherever "n" is used in the formula, "n1" will be used instead. Some structural evaluations are listed in Table 2.3. Figure 2.4 illustrates these simplifications.



(a) Symbolic evaluation.



(b) Structural evaluation.

Figure 2.4: Symbolic / structural evaluation.

Input values that triggers reduction	Equivalent reduction
and n1 true	n1
bvshl n1 bv0	n1
bvudiv n1 bv1	n1
bvsdiv n1 bv1	n1
bvmul n1 bv1	n1
bvadd n1 bv0	n1
bvsub n1 bv0	n1

Table 2.2: Symbolic evaluation

Specific structure that triggers reduction	Equivalent reduction
and n1 n1	n1
implies n1 n1	true
ite X n1 n1	n1
ite F X n1	n1
ite T n1 X	n1
bvudiv n1 n1	bv1
bvurem n1 n1	bv0
extract[bw-1:0] n1 (where bw is bit-width of n1)	n1

Table 2.3: Structual evaluation

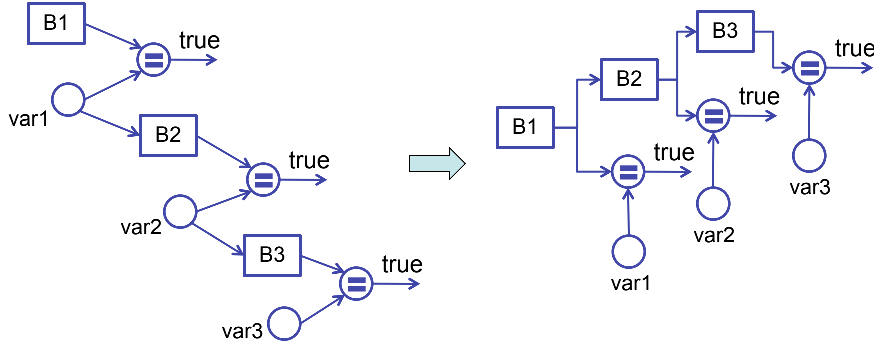


Figure 2.5: Equality Propagation

2.1.5 Equality Propagation

Equality propagation implements the following simple rule:

> If (= inp (expr)) is constrained to be true, where inp is a term of type input, and expr is any arbitrary term, then transfer all fan-outs of inp to term.

Transferring fan-outs from vertex v_1 to v_2 is equivalent to expression substitution – replacing all occurrences of v_1 in the formula by v_2.

This rule helps significantly in simplifying formulas arising from various software analyses that use static single assignment (SSA) form. These formulas consist of large number of intermediate variables, and conjunction of large number of shallow assertions, as illustrated in the following example.

Example
```
:assumption (= var1 (bvadd a b))
:assumption (= var2 (bvmul var1 c))
:assumption (= var3 (bvadd var2 d))
:assumption (bvugt var3 bv0)
```

The equality propagation would simplify the above three assumptions to build a single assumption with deeper expression as given below.

Example
```
:assumption (bvugt (bvadd (bvmul (bvadd a b) c) d) bv0)
```

This is equivalent to $(a +_w b) \times_w c +_w d >_u 0$. This is also illustrated in Figure 2.5. Deeper expressions give further opportunities of word-level simplification. Even if that does not happen, complexity of the SAT problem after bit-blasting is greatly reduced, because the superfluous variables `var1`, `var2`, `var3` do not need to be passed to the SAT solver.

2.1.6 Division / Remainder Simplification

As bit-vector division generates larger circuit as compared to multiplication for the same bitwidth, we replace division by multiplication. Essentially, $q = a/b$ is expressed using constraints $(a = q * b + r) AND (r < b)$. One has to be careful about overflows. The exact details are as follows.

For "n := (bvudiv a b)",

1. Add new inputs "q", "r" of same bitwidth as "n"
2. Add constraint "(= a (bvuaddno (bvumulno q b) r))"
3. Add constraint "(bvult r b)"
4. Transfer fan-outs from "n" to "q"

Here, "bvuaddno" is non-overflowing unsigned addition, that is bit-vector addition with implicit constraint that overflow must not occur. This implicit constraint is easily enforced during bit-blasting. Similarly, "bvumulno" is non-overflowing multiplication.

When fan-outs are transferred from n to q, n becomes unreachable from roots of the formula, and is therefore eliminated.

Similarly, for unsigned remainder, if "n := (bvurem a b)",

1. Add new inputs "q", "r" of same bitwidth as "n"
2. Add constraint "(= a (bvuaddno (bvumulno q b) r))"
3. Add constraint "(bvult r b)"
4. Transfer fan-outs from "n" to "r"

Care is taken to avoid adding duplicate constraints if both "(bvudiv a b)" and "(bvurem a b)" occur in the formula.

Signed division (bvsdiv) and remainder (bvsrem) are handled by first expressing them using their unsigned counterparts as follows. Vertex "n := bvsdiv a b" is equivalent to the output of following collection of vertices.

```
param bitwidth = bitwidth of n

let zero = bv0[bitwidth] in
let abs_a = (ite (bvslt a zero) (bvneg a) a) in
let abs_b = (ite (bvslt b zero) (bvneg b) b) in
let uresult = (bvudiv abs_a abs_b) in
let neg_ureselt = (bvneg uresult) in
let condition = (xor (bvslt a zero) (bvslt b zero) in
  (ite condition (bvneg uresult) uresult)
```

Vertex “n := bvsrem a b” is equivalent to the output of following collection of vertices.

```
param bitwidth = bitwidth of n

let zero = bv0[bitwidth] in
let abs_a = (ite (bvslt a zero) (bvneg a) a) in
let abs_b = (ite (bvslt b zero) (bvneg b) b) in
let uresult = (bvurem abs_a abs_b) in
let neg_ureselt = (bvneg uresult) in
let condition = (bvslt a zero) in
  (ite condition (bvneg uresult) uresult)
```

The only difference from “bvsdiv” is in use of “bvurem” instead of “bvudiv”, and modified “condition”.

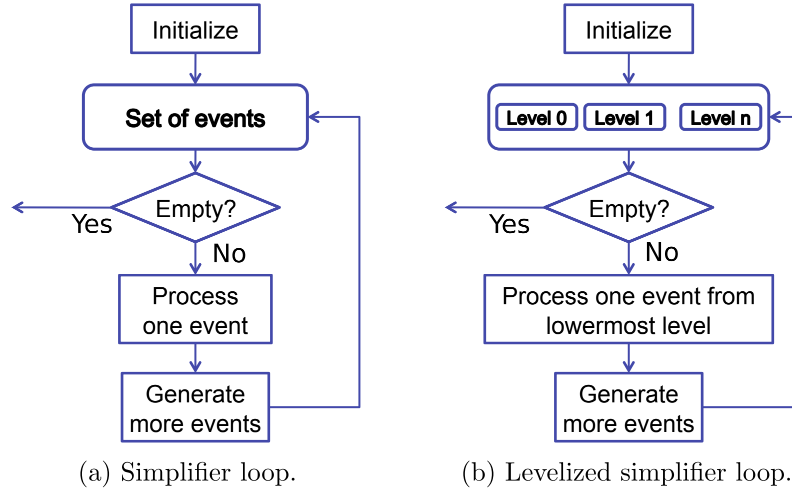
Constant-divisor division / remainder

Unsigned division by power-of-2 divisor is converted to logical right shift (**bvlshr**), while signed one is converted to arithmetic right shift (**bvashr**). Unsigned remainder is converted to extraction (**extract**), while signed remainder is first converted to unsigned remainder using the above conversion.

We also experimented with use of magic numbers to turn division by a constant into multiplication by a constant. The technique is taken from the book – Hacker’s Delight [34]. It is available as a command-line option in Beaver. However, neither did we use it in our experiments nor is it turned on by default.

2.2 Event-driven, On-the-fly Simplification

All of the word-level simplifications are local in nature, that is, they are centered at a single vertex, and depending on the connections and values surrounding that vertex, some modifications are made to the vertex and its surroundings. This motivates an implementation based on even-driven simulation. An event defined as a vertex that needs to be checked for opportunities of simplifications. Processing an event means checking whether any of the simplications are applicable to the vertex, and then doing the appropriate local modifications. These modifications might generate new simplification



(a) Simplifier loop. (b) Levelized simplifier loop.

Figure 2.6: Event-driven simplification.

opportunities at the neighbouring vertices, and hence those are queued in the form of new events. Thus, the simplifier is a event-processing loop as shown in Figure 2.6a.

In addition, many of the simplifications are forward in nature, that is, they only affect the fan-outs of vertices that they simplify. Thus, if one event is in the fan-out of another event, then it makes sense to process the latter event first. This is done by *levelizing* the formula graph where every vertex is assigned as integral level that is larger than the levels of its inputs, with inputs being assigned level 0. Then the event queue is split into a priority queue where the lower level means higher priority. This levelized simplifier loop is shown in Figure 2.6b.

Also, the simplifications are done on-the-fly while parsing and building the formula graph. This is implemented by adding simplification steps to the `add_vertex`, `assert_vertex` functions of the formula graph. Some simplifications, such as structural hashing, associativity, commutativity, forward evaluations, are done immediately while adding a vertex without going through the event queue. Others are put in the queue, which is processed periodically when it has collected enough events.

Finally, the don't-care elimination is implemented by keeping a "solved" flag for vertices and then traversing the vertices using DFS/BFS from the formula graph roots. The vertices that are unreachable, possibly because they

are hidden behind solved vertices, remain unvisited and thus eliminated.

2.3 Bit-blasting and Mapping to SAT

After word-level simplifications, the bit-vector formula graph is translated to an equi-satisfiable Boolean formula in the form of combinational AIG circuit. AIG – And-Inverter-Graph – has become a popular format for gate-level circuits in logic synthesis and verification. It allows us to use the circuit simplification techniques of ABC. In addition, ABC provides an extremely optimized implementation of AIG, which is important because for some SMT formulas, Boolean graphs can have over million gates.

Another novel feature of our bit-blaster is the use of *circuit templates*. Templates are AIG implementations of the bit-vector operators such as **bvadd**, **bvmul**, **bvudiv**. They are made by first writing structural Verilog implementations of these operators, and then flattening them to AIG representation. The Verilog implementations can be tested by simulation using test inputs. Having AIG templates also allows us to perform offline simplifications on them using ABC. The bit-blaster code consists of a generic instantiation procedure that reads templates and instantiates them for every vertex in the formula graph. If one wants to bit-blast certain operator differently, then just the template needs to replaced. This approach is more systematic, efficient, flexible and well-tested than the most obvious way of bit-blasting implemented by almost all other SMT solvers: given a Boolean graph data structure having basic functions such as `mk_and`, `mk_not`, implement functions such as `mk_fulladder`, `mk_adder`, `mk_add`, `mk_mul`, `mk_udiv`, and so on. The problem with this is that one ends up with over 1000 lines of unintellible code with no clear way to test the generated circuits, or to quickly change implementations.

Finally after bit-blasting, we use ABC to do some simple Boolean simplifications. Then the formula is to be passed to a SAT solver. If it is a circuit-based SAT solver, we directly write out ABC's AIG. Otherwise for CNF-based SAT solvers, we use ABC's CNF generation methods to write out a CNF formula. We experimented with two methods of CNF generation in ABC – 1) Tseitin encoding, and 2) technology-mapping-based CNF generation. The latter generates smaller CNF formulas, and does provide some speed-up for some formulas, but it is not a uniform improvement of very large factor for all formulas. These comparisons are available in our

technical report [24]. Both the methods are available through command-line options, and their effects should be evaluated for specific applications.

Chapter 3

Experiments using SMT-LIB benchmarks

We evaluated our solver and other bit-vector solvers on QF_BV benchmarks in SMT-LIB [31]. Currently, this library has over 30000 benchmarks, but about 28000 are from single source, Sage, and are of similar nature. Hence we narrowed down to 3678 benchmarks for our experiments. These contain all the benchmarks other than those from Sage, and a few of Sage benchmarks.

We used the PSI cluster of the Millennium clusters [1] to conduct the experiments. Each workstation had an Intel(R) Xeon(TM) 3.00 GHz CPU, 3 GB RAM, and running 64-bit Debian "etch" Linux 2.6.18. We enforced memory limit of 1 GB and timeout of 1 hour on all runs. Versions of the solvers used are listed in Table 3.1. For Beaver, we used NFLSAT [21, 22] as the backend SAT solver. In our previous technical report, we make comparisons of using NFLSAT vs. other CNF-based SAT solvers [23, 24].

Basic summary of the runs is as shown in Table 3.2. The scatter plots of total runtimes are shown in Figure 3.1, Figure 3.2, Figure 3.3. In these scatter plots, x-axis is time taken by Beaver, y-axis is time taken by the other solver. Timeouts or errors make the time 3600s, i.e. the maximum. All axes are in log scale.

Beaver is not the best solver, but has comparable performance among the group. MathSAT, Boolector are the fastest, with ability to solve most number of formulas, and in shorter times. OpenSMT gives syntax errors on a large number of formulas (this is not entirely surprising, because many of the SMT-LIB formulas deviate from the standard a little, e.g. having **and** connective with only one input). Curiously, MathSAT and Z3 reported

Solver	Version	Options
Beaver	svn version r864	NFLSAT
Z3	2.6	Default
OpenSMT	svn version r29	Default
MathSAT	4.3 of SMT-COMP'09	`-smt-comp` option
Yices	yices2smt09	Default
Boolector	1.2	Default
STP	svn version r709	Cryptominisat2

Table 3.1: Versions of solvers and other info. The svn versions correspond to the official svn repositories of the respective solvers.

Exit status	Beaver	Boolector	MathSAT	Z3	Yices	STP	OpenSMT
0	3222	3554	3577	3191	3193	3423	1714
1	0	43	0	0	0	0	1743
139	0	0	5	0	0	0	11
101	0	0	0	14	0	0	0
2	2	0	0	0	0	0	0
37	13	0	0	0	0	0	0
NOLOG	0	0	0	0	2	2	0
UNFINISHED	441	81	93	469	483	253	210
WRONG	0	0	3	4	0	0	0
total	3678	3678	3678	3678	3678	3678	3678

Table 3.2: Basic summary of runs using different solvers on SMT-LIB benchmarks. Exit status 0 means that solver ended successfully, and returned the expected result. Other non-zero exit statuses are the ones reported by the solvers on hitting some kind of failure. UNFINISHED, NOLOG mean that solver was not able to finish within time/memory limit. WRONG means that solver ended successfully, but reported unexpected result.

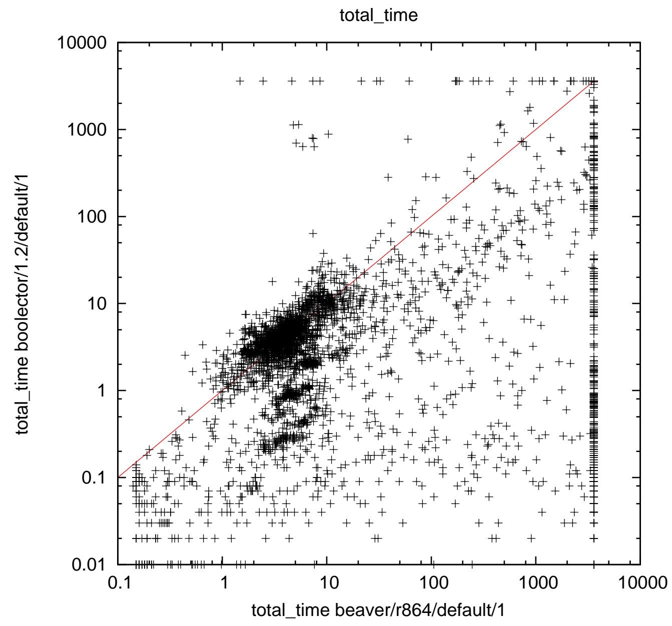
unexpected result for a few formulas. For MathSAT, these formulas are: `calypto/problem_7`, `calypto/problem_8`, `calypto/problem_9`. MathSAT reports UNSAT, when expected status is SAT (at least three other solvers report SAT). Z3 seems to go wrong on `spear/wget_v1.10.2/src_wget_-vc18196`, `spear/wget_v1.10.2/src_wget_vc18197`, `spear/wget_v1.10.2/src_-wget_vc18755`, `spear/wget_v1.10.2/src_wget_vc18756` – all other solvers report the expected result UNSAT, while Z3 reports SAT.

Looking in detail at the comparison of Beaver vs. MathSAT, we observe that for about 320 formulas, Beaver times out after 3600 seconds, but MathSAT is able to finish successfully within 10 seconds. For additional 220 formulas, both the solvers finish, but MathSAT gives speed-up of more than 1000x over Beaver. Most of these formulas are from bruttomesso family, with the remaining few from brummayerbiere*, wienand-cav2008, calypto, and VS3. In our past experience, we have observed that such a great performance gap usually means missing rewrite rules, in absence of which the bit-blasted formula is too complex for the SAT solver to solve. To look for these missing rewrites is an item for future work.

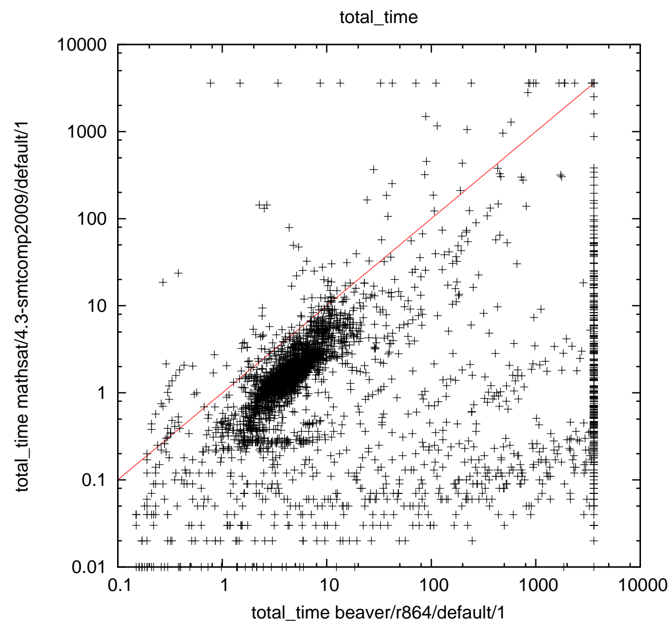
3.1 Effectiveness of simplifications in Beaver

As reported earlier in [23, 24], we evaluated the effectiveness of various simplifications performed in Beaver. Figure 3.4 is a scatter plot of runtimes with and without word-level simplifications. Big cluster of points above the diagonal shows the effectiveness of these simplifications. In addition, the path feasibility queries (marked by $\times$ in the plot) benefit greatly from word-level rewrites – especially equality propagation.

We evaluated two CNF generation techniques available in ABC – 1) standard Tseitin encoding, with some minimal optimizations like detection of multi-input ANDs, ORs and muxes, and 2) technology mapping based CNF generation [18], which uses optimization techniques commonly found in the technology mapping phase of logic synthesis to produce more compact CNF. Comparison of these two techniques using ABC's internal Minisat 1.14 is shown in Figure 3.5a. The TM-based CNF generation significantly reduced the SAT run-time for the `spear` benchmarks, but actually increased the runtime for the `brummayerbiere` family of benchmarks. For other SAT solvers, the two techniques didn't result in appreciable difference. Also, neither CNF generation technique improved much on NFLSAT.

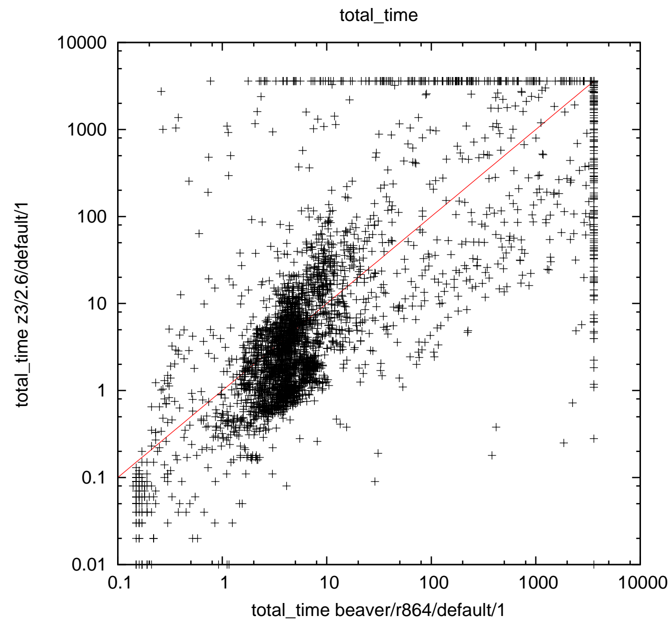


(a) Beaver vs. Boolector

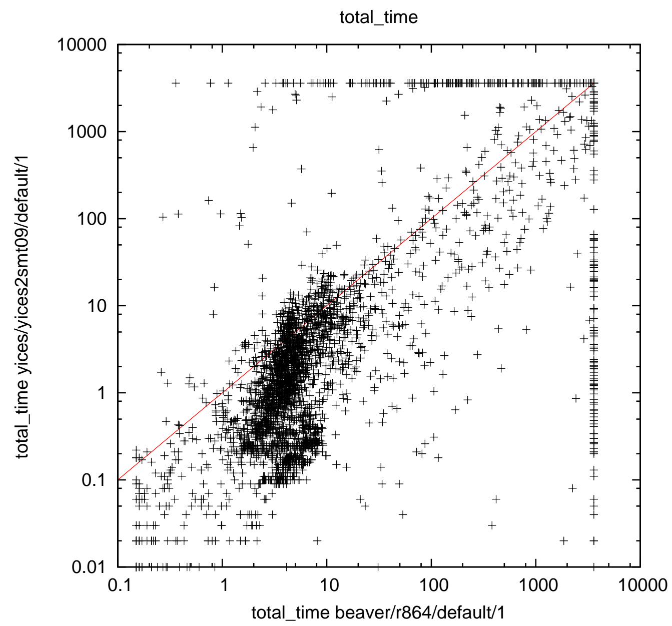


(b) Beaver vs. MathSAT.

Figure 3.1: Run-time comparison of Beaver with Boolector and MathSAT.

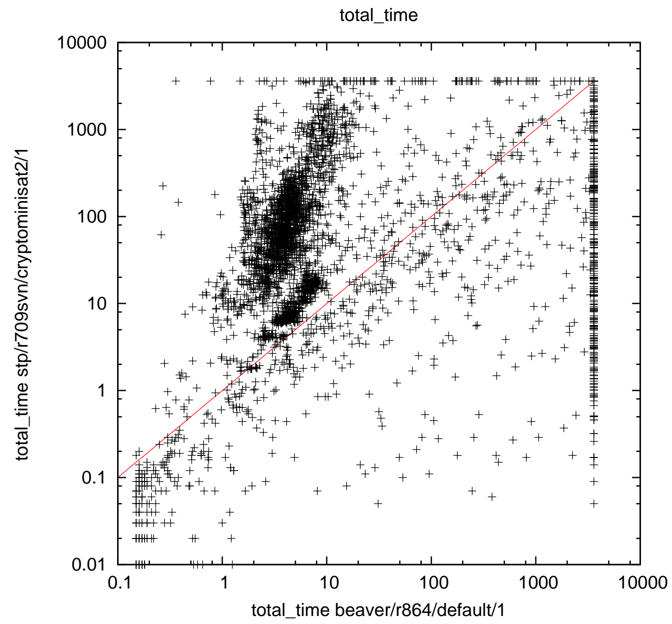


(a) Beaver vs. Z3

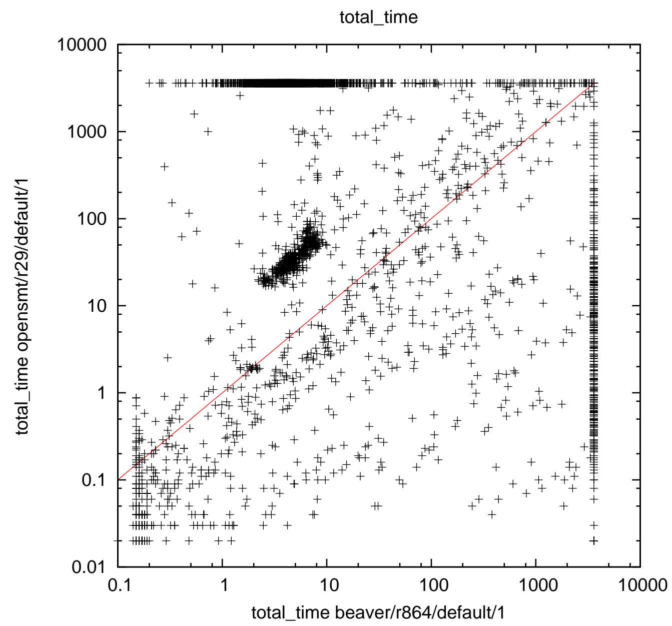


(b) Beaver vs. Yices2.

Figure 3.2: Run-time comparison of Beaver with Z3 and Yices2.



(a) Beaver vs. STP



(b) Beaver vs. OpenSMT.

Figure 3.3: Run-time comparison of Beaver with STP and OpenSMT.

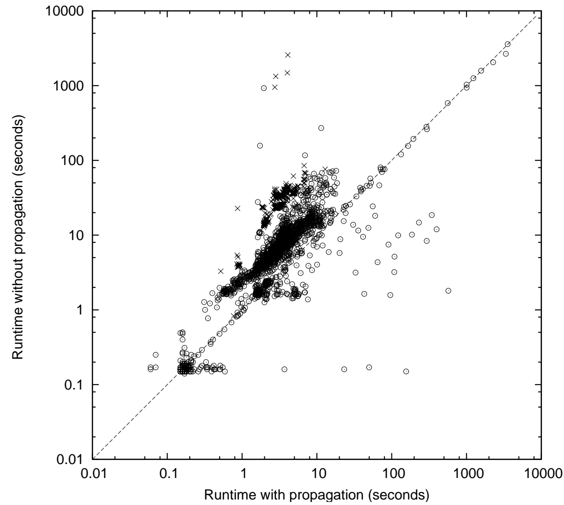
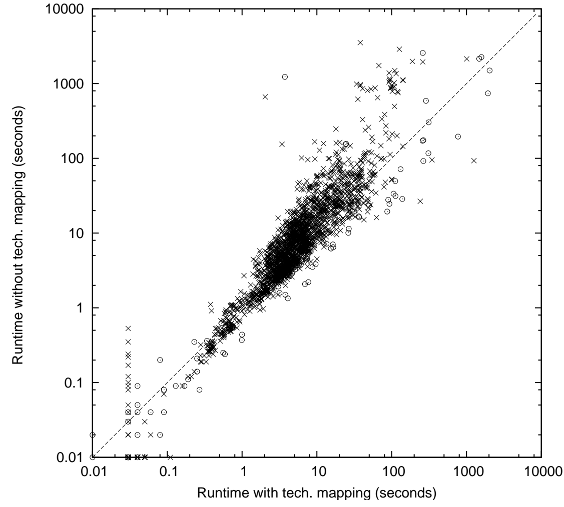
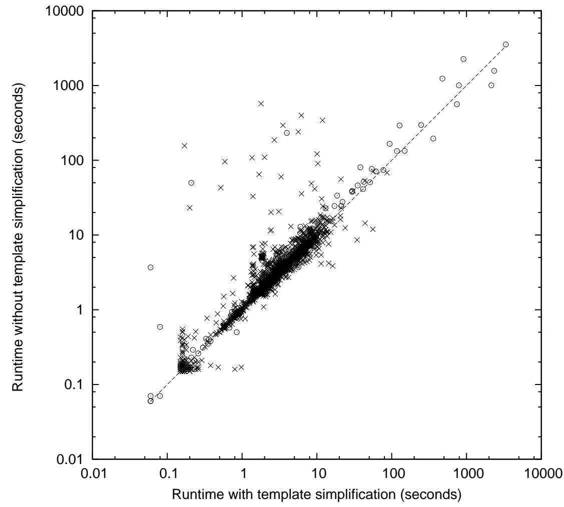


Figure 3.4: Impact of word-level simplifications (the path feasibility queries are marked with $\times$). x-axis is the run-time with simplifications, and y-axis is without simplifications. Both axes are in log-scale.

The use of logic optimization techniques on the template circuits for arithmetic operators was beneficial in general, as can be seen from Figure 3.5b. The speed-up obtained from optimizing the templates was especially large for the `spear` benchmarks. We hypothesize that this is because that family of benchmarks has a wide variety of arithmetic operations, including several non-linear operations, which are the ones for which template optimization helps the most.



(a) Impact of tech-mapping-based CNF generation.



(b) Impact of template optimizations.

Figure 3.5: Impact of Boolean simplifications. In both the plots, x-axis is run-time with simplification, and y-axis is without it. All axes use log-scale. Satisfiable benchmarks are marked using $\times$ and unsatisfiable benchmarks as $\circ$.

Chapter 4

Conclusion

We presented the details of design of our SMT solver, Beaver, for quantifier-free bit-vector logic. Like its contemporaries, Beaver uses the obvious way of solving bit-vector formulas – word-level simplifications, followed by bit-blasting to SAT. However, a few sound design decisions seem to have helped making it a decent solver for QF_BV logic. At word-level, generic formula graph data structure, and event-drien, on-the-fly simplification scheme can be utilized to construct SMT solvers for theories beyond bit-vectors. In addition, the choice of using ABC as the backend is a systematic, modular approach to implementing bit-blasting. Some key features are still missing in our solver, and they are listed below. They might be items of future work.

- Backtracking: currently Beaver works like a single-pass compiler, and there is no support for adding and removing assertions (which will be a nice-to-have feature once Beaver is made into a library).
- UNSAT core: support for generation of UNSAT core would be a good additional feature.
- More rewrites: as mentioned in Chapter 3, for several benchmarks in SMT-LIB, Beaver times out after 1 hour, while other solvers finish within a few seconds. This usually indicates one or two missing rewrite rules. In general, it would be great to have an easy way to manually hunt for such missed simplifications, and add the corresponding rewrite rules.

Acknowledgments

I thank my advisor, Professor Sanjit Seshia, and a fellow colleague student, Susmit Jha, with whom I conducted most of this work. Part of this report appears in a paper jointly written with Susmit Jha and Prof. Seshia – *Susmit Jha, Rhishikesh Limaye, and Sanjit Seshia. Beaver: Engineering an efficient SMT solver for bit-vector arithmetic. In Computer Aided Verification, pages 668–674. 2009* [23].

Bibliography

[1] UC berkeley millennium clusters. http://www.millennium.berkeley.edu/.

[2] Domagoj Babić. *Exploiting Structure for Scalable Software Verification.* PhD thesis, University of British Columbia, Vancouver, Canada, 2008.

[3] Domagoj Babić and Frank Hutter. Spear theorem prover. In *Proc. of the SAT 2008 Race*, 2008.

[4] Sébastien Bardin, Philippe Herrmann, and Florian Perroud. An alternative to SAT-based approaches for bit-vectors. In Javier Esparza and Rupak Majumdar, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 6015, chapter 7, pages 84–98. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[5] Clark Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. *Satisfiability Modulo Theories*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, chapter 26, pages 825–886. IOS Press, February 2009.

[6] Clark Barrett and Cesare Tinelli. CVC3. In Werner Damm and Holger Hermanns, editors, *Computer Aided Verification*, volume 4590 of *Lecture Notes in Computer Science*, chapter 34, pages 298–302. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[7] Armin Biere. PicoSAT essentials. *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*, 4(2-4):75–97, 2008.

[8] Robert Brummayer and Armin Biere. Boolector: An efficient SMT solver for bit-vectors and arrays. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 174–177. 2009.

[9] Roberto Bruttomesso, Alessandro Cimatti, Anders Franzén, Alberto Griggio, Ziyad Hanna, Alexander Nadel, Amit Palti, and Roberto Sebastiani. A lazy and layered SMT(BV) solver for hard industrial verification problems. In *Computer Aided Verification (CAV '07)*, volume 4590/2007 of *Lecture Notes in Computer Science*, pages 547–560, Berlin, Germany, July 2007. Springer Berlin / Heidelberg.

[10] Roberto Bruttomesso, Alessandro Cimatti, Anders Franzén, Alberto Griggio, and Roberto Sebastiani. The MathSAT 4 SMT solver. In Aarti Gupta and Sharad Malik, editors, *Computer Aided Verification*, volume 5123 of *Lecture Notes in Computer Science*, chapter 28, pages 299–303. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[11] Roberto Bruttomesso, Edgar Pek, Natasha Sharygina, and Aliaksei Tsitovich. The OpenSMT solver. In Javier Esparza and Rupak Majumdar, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 6015, chapter 12, pages 150–153. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[12] Roberto Bruttomesso and Natasha Sharygina. A scalable decision procedure for fixed-width bit-vectors. In *ICCAD '09: Proceedings of the 2009 International Conference on Computer-Aided Design*, pages 13–20, New York, NY, USA, 2009. ACM.

[13] Randal Bryant, Daniel Kroening, Joël Ouaknine, Sanjit Seshia, Ofer Strichman, and Bryan Brady. An abstraction-based decision procedure for bit-vector arithmetic. *International Journal on Software Tools for Technology Transfer (STTT)*, 11(2):95–104, April 2009.

[14] Randal E. Bryant, Daniel Kroening, Joël Ouaknine, Sanjit Seshia, Ofer Strichman, and Bryan Brady. Deciding bit-vector arithmetic with abstraction. In *Tools and Algorithms for the Construction and Analysis of Systems: 13th International Conference, TACAS 2007 Proceedings*, volume 4424/2007 of *Lecture Notes in Computer Science*, pages 358–372. Springer Berlin / Heidelberg, March 2007.

[15] Sylvain Conchon, Jean-Christophe Filliâtre, and Julien Signoles. Ocamlgraph: Generic graph library for Objective Caml. http://ocamlgraph.lri.fr/.

[16] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science, chapter 24, pages 337–340. 2008.

[17] Bruno Dutertre and Leonardo de Moura. The YICES SMT solver. Technical report, Computer Science Laboratory, SRI International, 2006.

[18] Niklas Eén, Alan Mishchenko, and Niklas Sörensson. Applying logic synthesis for speeding up SAT. In João Marques-Silva and Karem A. Sakallah, editors, *Theory and Applications of Satisfiability Testing (SAT 2007)*, volume 4501 of *Lecture Notes in Computer Science*, chapter 26, pages 272–286. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[19] Vijay Ganesh and David L. Dill. A decision procedure for bit-vectors and arrays. In *Computer Aided Verification (CAV '07)*, volume 4590/2007 of *Lecture Notes in Computer Science*, pages 519–531, Berlin, Germany, July 2007. Springer Berlin / Heidelberg.

[20] Frank Hutter, Domagoj Babic, Holger H. Hoos, and Alan J. Hu. Boosting verification by automatic tuning of decision procedures. *Formal Methods in Computer Aided Design*, 0:27–34, 2007.

[21] Himanshu Jain. *Verification Using Satisfiability Checking, Predicate Abstraction, and Craig Interpolation*. PhD thesis, Carnegie Mellon University, School of Computer Science, 2008.

[22] Himanshu Jain and Edmund M. Clarke. Efficient SAT solving for non-clausal formulas using DPLL, graphs, and watched cuts. In *DAC '09: Proceedings of the 46th Annual Design Automation Conference*, pages 563–568, New York, NY, USA, 2009. ACM.

[23] Susmit Jha, Rhishikesh Limaye, and Sanjit Seshia. Beaver: Engineering an efficient SMT solver for bit-vector arithmetic. In *Computer Aided Verification*, pages 668–674. 2009.

[24] Susmit K. Jha, Rhishikesh S. Limaye, and Sanjit A. Seshia. Beaver: Engineering an efficient SMT solver for bit-vector arithmetic. Technical Report UCB/EECS-2009-95, EECS Department, University of California, Berkeley, June 2009.

[25] Daniel Kroening and Ofer Strichman. *Decision Procedures: An Algorithmic Point of View*. Texts in Theoretical Computer Science. An EATCS Series. Springer Berlin Heidelberg, 2008.

[26] Panagiotis Manolios, Sudarshan Srinivasan, and Daron Vroon. BAT: The bit-level analysis tool. In Werner Damm and Holger Hermanns, editors, *Computer Aided Verification*, volume 4590 of *Lecture Notes in Computer Science*, chapter 35, pages 303–306. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[27] SMT-COMP Organizers. SMT-COMP'07 results. http://www.smtexec.org/exec/?jobs=20.

[28] SMT-COMP Organizers. SMT-COMP'08 results. http://www.smtexec.org/exec/?jobs=311.

[29] SMT-COMP Organizers. SMT-COMP'09 results. http://www.smtexec.org/exec/?jobs=529.

[30] SMT-LIB Initiative. Quantifier-free bit-vector (QF_BV) logic. http://goedel.cs.uiowa.edu/smtlib/logics/V1/QF_BV.smt.

[31] SMT-LIB Initiative. SMT-LIB. http://www.smtlib.org/.

[32] SMT-LIB Initiative. Theory of Fixed-size Bit-vectors. http://goedel.cs.uiowa.edu/smtlib/theories/V1/BitVectors.smt.

[33] Berkeley Logic Synthesis and Verification Group. ABC: A system for sequential synthesis and verification, release 70930. http://www.eecs.berkeley.edu/˜alanmi/abc/.

[34] Henry S. Warren. *Hacker's Delight*. Addison Wesley, 2003.