An Efficient Finite-Domain Constraint Solver for Circuits

G. Parthasarathy M. K. Iyer K.-T. Cheng Li-C. Wang
{gpartha, madiyer, timcheng, licwang}@ece.ucsb.edu
Department of Electrical and Computer Engineering
University of California, Santa Barbara

ABSTRACT

This paper presents a novel hybrid finite-domain constraint solving engine for RTL circuits. We describe how DPLL search is modified for search in combined integer and Boolean domains by using efficient finite-domain constraint propagation. This enables efficient combination of Boolean SAT and linear integer arithmetic solving techniques. We automatically use control and data-path abstraction in RTL descriptions. We use conflict-based learning using the variables on the boundary of control and data-path for additional performance benefits. Finally, we analyze the hybrid constraint solver experimentally using some example circuits.

Categories and Subject Descriptors: F.4.1 Mathematical Logic: Mechanical theorem proving; I.1 Symbolic and Algebraic Manipulation; T.2.2 Verification: Functional and formal verification.
General Terms: Algorithms, Constraints, Circuits
Keywords: Design Verification, Decision Procedures, Boolean Satisfiability, Integer Linear Programming, Bit-vector arithmetic

1. INTRODUCTION

Numerous *electronic design automation* (EDA) problems can be efficiently represented by a combination of Boolean and integer constraints – like formal verification and functional test generation for RTL circuits. A combined decision procedure (CDP) that integrates decision procedures for Boolean and integer domains should be ideal for solving such problems.

Boolean Satisfiability (SAT) solvers have improved significantly over the last few years [13, 17]. SAT solvers are now frequently applied to EDA problems that are expressible as propositional SAT instances. However, they still face problems of scalability for large RTL designs. Decision procedures for integer domains such as the Omega test [11], based on *Fourier-motzkin elimination* [6](FME) are very efficient for checking satisfiability of large sets of integer constraints. However, they ignore the distinction between Boolean and integer domains in the problem for efficiency in the general case. EDA problems on RTL circuits appear to be such that neither a SAT solver nor an integer constraint solver can solve them individually in a reasonable time.

RTL circuit descriptions typically have well defined sets of Boolean and integer variables. This property allows automatic partitioning of Boolean control and word-level data-path. We use such a partition to integrate specialized methods for Boolean domains and integer domains without sacrificing the efficiency of each solver.

The main contribution of this paper is an attempt to generalize the key elements of efficient DPLL search to an efficient FD constraint solver for RTL circuits. In our approach, we systematically modify DPLL [7], a well-known branch-and-bound algorithm into a hybrid DPLL (HDPLL) algorithm, that integrates solvers for Boolean and integer domains into a cohesive whole. We augment this with *conflict-based learning* to drive search into a solution space efficiently.

The rest of this paper is organized as follows: In Section 2, we describe the relevant prior work. In Section 3, we describe some concepts that are used to explain our approach. In Section 4, we describe a novel *hybrid DPLL constraint-solving* (HDPLL) algorithm, which integrates Boolean search and fourier-motzkin elimination using FDCP. We also describe modeling issues and our efficient implementation of the FDCP. In Section 6, we describe how we use conflict-based learning to bound hybrid search. In Section 7, we describe our experiments to show proof-of-concept of our approach. Finally, we discuss the results of the experiments in Section 8.

DAC – 2004, June 7–11, 2004, San Diego, California, USA.

2. PRIOR WORK

There has been a significant amount of work on algorithms that use a structural description (*net-list*) of a circuit to improve the performance of satisfiability checking [1]. SAT based models of EDA problems are compact and can be adapted to various logic algebra [16]. A modern SAT solver can solve these problems efficiently. Various authors have tried to integrate Boolean SAT and BDDs [5]. Iyer proposed constraint solving for generating simulation test-benches [9].

Attempts have been made to combine Boolean and integer arithmetic solving as two cooperating decision procedures. Barrett *et al.*, proposed integrating a *presburger arithmetic* solver and SAT based on equality propagation and expression rewriting. They also extended their solver to handle bit-vector arithmetic [2, 3]. These solvers are designed as generalized decision procedures; not optimized for circuits. Hence, they are currently impractical for EDA problems.

Liu *et al.*, proposed an elegant approach – CAMA, to solve satisfiability of Boolean logic formulas on multi-valued variables [12]. CAMA is quite useful for applications with purely Boolean operations on word variables, like multi-valued synthesis. CAMA represents formulas in *conjunctive normal form* (CNF), with a set-based representation of the values for each literal. CAMA uses a DPLL style procedure, with multi-valued implication and resolution on the clauses in the formula. However, their technique models arithmetic operations like addition or comparison as Boolean operations on word variables, which poses scalability problems on general RTL circuits.

We have approached the problem differently from earlier work. We try to optimize our approach for applications on RTL circuits. We automatically partition the circuit at points where Boolean and integer domains interface. The proposed solver based on this partition is a branch-and-bound algorithm that is easily extensible to other solvers. We use 3-valued search in the control part to enable implicit enumeration on decision variables. We build on established work to develop models and data-structures for efficient FDCP. We also use conflict-based learning to improve the search. Learned relations are Boolean relations between control signals, which naturally abstract data-path constraints. Our approach works well, since the integration maintains performance of the component engines.

3. BACKGROUND

A *range* of integers $[l \ldots u]$, is the set of integers $\{l, l+1, \ldots, u\}$, or $\emptyset$ if $l < u$. A *domain* $\mathcal{D}$, is a complete mapping from a fixed set of variables $\mathcal{V}$ to finite sets of integers. A variable is called a *ground* variable if there is only value in its domain. A Boolean variable b_i has a domain of $\{0,1\}$, and a word variable w_i, has a domain of $\{0 \ldots N\}$, where N is a finite integer.

Given a finite set of variables $\mathcal{V}$, over the set of Boolean values $\mathbf{B} \in \{0,1\}$, a *literal*, $l/\overline{l}$ is a variable, $v/\neg v \in \mathcal{V}$. A *clause* c_i, is a disjunction of literals. Given a finite set of variables $x_i \in \mathcal{W}$ over finite integer domains, an $n-term$ *finite-domain* constraint is an inequality of the form:

$$\sum_i a_i \cdot x_i \geq r,\ a_i, r \in I, x_i \in \{0 \ldots n\}$$

A *primitive* constraint is a finite-domain constraint with at most three terms. We denote the set of primary inputs of the RTL circuit by **PI**, and the set of primary outputs by **PO**. Present state variables or *pseudo-primary inputs* are denoted by **PPI** and next state variables or *pseudo-primary outputs* by **PPO**. We may now view satisfiability on circuits with Boolean and arithmetic operations with Boolean and word-level variables as a *FD constraint solving* problem.

3.1 DPLL Search

The most effective SAT solvers use a DPLL style branch-and-bound algorithm [7], shown in Figure 1 to systematically search the possible solution space.

```
procedure dpll()
while (Decide() ≠ Done) do
  while (bcp() == conflict) do
    blevel = analyzeconflicts();
    if (blevel == 0) then
      return UNSATISFIABLE;
    else
      backtrack(blevel);
    end if
  end while
end while
return SATISFIABLE;
```

Figure 1: Boolean DPLL with Conflict-based learning.

Initially, the assignment corresponding to the proposition is implied on the formula. This corresponds to **`blevel = 0`** in Figure 1. If the procedure **`bcp`**(), cannot find a conflict, then the procedure **`Decide`**() makes additional *decisions* on variables using procedure **`Decide`**(). The procedure **`bcp`**(), is then called to generate a set of implied assignments (*implications*) using BCP. These implications *must* hold for the proposition to be **true** under the current partial assignment. An assignment may be inconsistent under two conditions. The first condition occurs when a variable is implied to have two different values at the same time – a *conflict*. This is easy to check when a new value is set on a variable. The second condition occurs when all the literals in a clause evaluate to **false**. This can be checked efficiently by checking only those clauses that have a new implication on its variables during BCP. This process continues until the search space is empty (UNSAT) or no more decision variables remain (SAT).

If a conflict is detected, the procedure **`analyzeconflicts`**() performs *conflict-based learning* to identify the value assignments that led to the conflict. This is done by selectively applying resolution to the clauses that implied values during the current partial assignment. The procedure returns the correct decision to backtrack to. This is denoted by the variable **`blevel`** in Figure 1. A proof for satisfiability or unsatisfiability of a set of Boolean clauses is a series of inference steps using resolution on individual clauses, until an empty clause is derived. If the collection of conflict clauses implies a conflict with the proposition to be proved, they constitute a proof of unsatisfiability. Hence, the instance is classified as UNSAT.

The main points of interest in the Figure 1 are:

1. The decision variables are a set of variables, $V \subseteq \mathcal{V}$, where $\mathcal{V}$ comprises all the variables of the propositional formula C.
2. **`bcp`**() is the single most commonly used procedure in **`dpll`**(). Its efficiency lower-bounds the efficiency of **`dpll`**().
3. The bounding of the search space is highly dependent on the size and nature of the conflict clauses [17]. The conflict clauses are resolvents on the set of clauses that led to conflicts during the decision procedure.

In the following section, we describe how we model RTL circuits as a set of Boolean and arithmetic constraints. We then explain the details of the hybrid DPLL algorithm and its components.

4. HYBRID DPLL

A Boolean gate in a circuit can be modeled as a conjunction of clauses. In general, circuits can be modeled as set of constraints – either entirely in CNF for bit-level circuits or pure arithmetic constraints for RTL data-path, or a mixture of both. HDPLL represents the arithmetic data-path as a set of *primitive constraints* and the Boolean control as a net-list of Boolean-clause based data-structures. This retains the efficiency of each solver in its domain.

The circuit is automatically partitioned into control and data-path along the *control-data-path interface*. This partition includes all bit-level lines, which are inputs from the control to the data-path and outputs from the data-path to the control. RTL operator outputs, which are used as inputs to the Boolean control logic, are called *interface primary outputs* (IPO) (*e.g.* outputs of comparators). Outputs from the Boolean control to a data-path operator are called *interface primary inputs* (IPI) (*e.g.* control lines to mux selects). The set of IPOs and IPIs are called *interface points*. We describe the details of how we model RTL operators in the next section.

4.1 Data-Path Modeling

All linear arithmetic data-path operations such as addition, subtraction, comparison, and multiplexer/case statements can be easily represented using primitive constraints. In the following, we describe how we normalize RTL data-path operators into a single type, which aids the design of our efficient constraint propagation engine. All word-level variables, w_i, are assumed to have a known bit-width and a known maximum range, L. Therefore, the maximum range of an arithmetic primitive with bit-width n, is $L = 2^n - 1$.

RTL Operator		Model Definition
$w_1 \geq w_2$	GEQ	$(w_1 - w_2 + L \cdot \overline{b} \geq 0) \wedge (w_2 - w_1 + L \cdot b \geq 1)$
$w_1 \leq w_2$	LEQ	$(w_2 - w_1 + L \cdot \overline{b} \geq 0) \wedge (w_1 - w_2 + L \cdot b \geq 1)$
$w_1 < w_2$	LT	$(w_2 - w_1 + L \cdot \overline{b} \geq 1) \wedge (w_1 - w_2 + L \cdot b \geq 0)$
$w_1 > w_2$	GT	$(w_1 - w_2 + L \cdot \overline{b} \geq 1) \wedge (w_2 - w_1 + L \cdot b \geq 0)$
$w_1 \equiv w_2$	EQ	$b_1 \models w_1 \geq w_2$, $b_2 \models w_1 \leq w_2$, and $(b_1 + b_2) \wedge (b_1 + \overline{b}) \wedge (b_2 + \overline{b}) \wedge (\overline{b_1} + \overline{b_2} + b)$ is True
if (b) $w_o = w_2$ **else** $w_o = w_1$	MUX or ITE	$(w_1 - w_o + L \cdot b \geq 0) \wedge (w_o - w_1 + L \cdot b \geq 0) \wedge (w_2 - w_o + L \cdot \overline{b} \geq 0) \wedge (w_o - w_2 + L \cdot \overline{b} \geq 0)$
$w_o = w_1 + w_2$	ADD	$(w_1 + w_2 - w_o \geq 0) \wedge (w_o - w_1 - w_2 \geq 0)$

Table 1: Modeling of RTL Arithmetic Operations with GEQs

We assume that the evaluation of a comparison operator *cmp* is a Boolean variable b, where $b = 1$, *only if* $b \models w_1$ *cmp* w_2 holds. The basic GEQ ($\geq$) operator is modeled with a pair of constraints, as shown in Table 1. All other comparison operations, $\{\leq, <, >, \equiv\}$, can be modeled using the basic GEQ operator. For the equality comparator, the result is a Boolean variable b, predicated on two auxiliary

boolean variables b_1 and b_2, such that they satisfy the relation shown in Table 1. The relation is similar to $b = b_1 \wedge b_2$, except that the case of $\{b_1, b_2\} = \{0,0\}$ is forbidden. Hence, $b = 0$, implies that either $b_1 = 1$ or $b_2 = 1$.

Multiplexers, addition and subtraction are represented using basic 2-input constructs, shown in Table 1. Multi-input operations are converted to a flattened representation of these basic constructs. RTL state machines typically use **`case`** statements. These are modeled as multiplexers with 1-hot encoded select signals. Non-linear constraints such as bit-vector operations, shifts, and mod-2 arithmetic, can be handled by adding Boolean variables with constant coefficients. The interested reader is referred to [3, 4] for details and background. Our implementation currently supports all major non-linear operations except multiplication and division.

4.2 Hybrid DPLL Algorithm

DPLL-based ATPG algorithms like FAN or PODEM [1] use a subset of Boolean variables in a circuit to check for satisfiability. Thus, the problem of finding a satisfying assignment in an RTL net-list with Boolean and arithmetic operators, is analogous to checking satisfiability of an objective in a Boolean circuit with the decision variables restricted to the set $V \equiv \{\mathbf{PI} \cup \{\mathbf{IPI} \cup \mathbf{IPO}\}$. If the assignments on $v|v \in \mathbf{IP} = \{\mathbf{IPI} \cup \mathbf{IPO}\}$ are simultaneously satisfiable in the data-path, then the overall problem is satisfiable.

```
procedure hdpll()
loop
  while (Decide() ≠ Done) do
    while (FdcpFme() == conflict) do
      blevel = hybrid_analyzeconflicts();
      if (blevel == 0) then
        return UNSATISFIABLE;
      else
        backtrack(blevel);
      end if
    end while
  end while
end loop
```

Figure 2: Modified DPLL for Hybrid Search.

The main elements of the hybrid DPLL algorithm are shown in Figure 2. As we can see, it is similar to the **`dpll`**() procedure in Figure 1. The procedure **`Decide`**() makes decisions similar to the **`Decide`**() in Figure 1. Decisions are made only on Boolean variables. Completeness is ensured by the criterion discussed above.

The Boolean solver finds values, which need to be set on **IP** in order to satisfy the objective. We use 3-valued search combined with a *reduction algorithm* [10], in the Boolean solver. This reduces the number of assignments on the interface points, which are needed to satisfy the objective. We view the don't-cares and Boolean assignments on the interface points as an *objective cube*, which encodes multiple satisfying assignments in a single solution. The objective cube increases search efficiency by implicit enumeration on **IP**.

Similar to DPLL, the procedure implies every decision using the procedure **`FdcpFme`**(), which uses a combination of FDCP and FME to check for new implications and all inconsistency conditions. If a conflict is found, then the procedure **`hybrid_analyzeconflicts`**() is called to perform conflict-based learning. This procedure is a modified version of the conflict analysis routine in Figure 1 to include FDCP. If FME returns a conflict, we use an iterative procedure to select a subset of assignments on the interface points that are sufficient to cause the conflict. This is used as a learned clause to bound the search. This is further described in Section 6. In the next section, we describe the details of the procedure **`FdcpFme`**() in greater detail

5. EFFICIENT FDCP

FDCP is a subject that has been extensively studied in the theory and implementation of *constraint logic programming*. It is beyond the scope of this paper to detail the subtleties involved in general FDCP. However, we shall explain the basic ideas behind our implementation by likening it to a generalized version of BCP. The algorithm for FDCP and consistency checking is shown in Figure 3.

```
procedure FdcpFme()
for all (a_i ∈ impqueue) do
  if (fdcp (a_i) == conflict) then
    return conflict;
  end if
end for
for all (a_i ∈ impqueue and a_i ∈ {IPO ∪ IPI }) do
  if (fme () == conflict) then
    return conflict;
  end if
end for
```

Figure 3: Combined FDCP and FME consistency checking

A constraint is satisfied under some assignment of values to the variables x_i, if the inequality (equality) holds. A *finite domain constraint propagation* (FDCP) procedure maps a set of constraints, C, and an initial domain $\mathcal{D}$, to a new domain $\mathcal{D}'$. Formally:

$$C \wedge \bigwedge_{x \in vars(C)} x \in \mathcal{D}(x) \leftrightarrow C \wedge \bigwedge_{x \in vars(C)} x \in \mathcal{D}'(x)$$

The procedure **`FdcpFme`**() in Figure 3 determines that the constraint set C and domain $\mathcal{D}$ are unsatisfiable when it returns a null domain $\mathcal{D}'$. The set of assignments **`impqueue`** in Figure 3 can be literals appearing purely in Boolean clauses, or in integer constraints, or domain changes on word-level variables. If all events in **`impqueue`** are processed without suppressing any implication, then all value changes in the control, which can affect the data-path and vice-versa are implied. This is further discussed in Section 7. The procedure **`FdcpFme`**(), consists of two procedures – **`fdcp`**() that does FDCP, and **`fme`**() that does a final hybrid consistency check using FME. We shall explain why we need to use FME in the following.

A constraint $\kappa \in C$ is domain consistent if, for each variable $v \in c$, no value in the domain of the variable v, is known to violate the constraint. By extension, the entire set C is consistent, if no such violation is known to exist for every value assignment in the domains of every variable in $\kappa \in C$. However, this implies that the entire constraint set may have to be considered for every domain change during FDCP, which is computationally very expensive. Therefore, we consider bounds consistency per constraint for FDCP. The procedure **`fdcp`**() in Figure 3 ensures that the bounds of all variables in a constraint respect the constraint by making bounds changes on variables. In general, neither BCP nor FDCP can deduce all possible implications caused by a set of assignments. In Boolean DPLL any inconsistency in the propositional formula from assignments can be detected by checking if any clause evaluates to *false*. However, FD consistency checking poses additional challenges.

This is further illustrated in the Figure 4. The variables $\{w_1, w_2, w_3\}$ in the example, have bit-width 3, and variables $\{w_4, w_5\}$ have bit-width 4. Given the Boolean objective $b_3 = 1$, FDCP using bounds propagation, will imply that $\{b_1, b_2, w_3\} = \{1, 1, 3\}$ and $\{w_4, w_5\} = [5, \ldots, 10], [5, \ldots, 10]\}$. However, $w_5 = w_2 + 5$ and $w_4 = w_2 + 3$ when $b_1 = 1$, which implies that $w_5 = w_4 + 2$. This in turns implies that $b_2 = 0$, which conflicts with the value assignment $b_2 = 1$. This conflict can be detected only by finding the relation between w_4 and w_5.

From the discussion above, it is clear that we cannot check for global consistency of integer constraints by individually checking each constraint. We also cannot implement a complete consistency check-

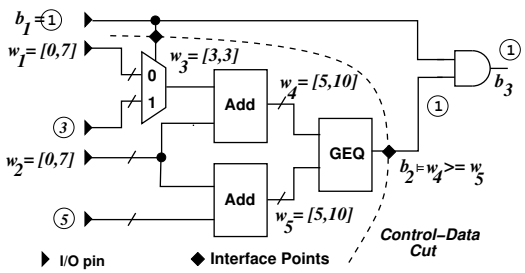


Figure 4: Limitations of Constraint Propagation

ing routine based on constraint propagation alone, due to efficiency reasons. Therefore, we need a final consistency check of the entire set of arithmetic data-path constraints when FDCP cannot detect a conflict and the Boolean control is consistent. One of the most efficient methods of checking consistency on FD constraints is fourier-motzkin elimination [6], which uses resolution on constraints to evaluate the size of the solution set for the problem. We use the *Omega-library*'s [11] implementation of FME to check consistency of the FD constraint set if FDCP cannot detect a conflict. This is done by the procedure **fme**() in Figure 3, which iteratively checks the satisfiability of each assignment on the data-path using FME. In general, this can be very expensive since FME has a worst-case exponential complexity. However, this does not appear in the average case.

Clearly, we have a trade-off between the calls to the FME solver and the expense of constraint propagation. In practice, we find that FDCP manages to detect most of the conflicts in the hybrid decision procedure, as we shall see in the experimental results in Section 7.

5.1 Implementation of FDCP

We describe the implementation details of our constraint propagation procedure in this section. A clause can be viewed as a constraint,

$$\sum_i v_i \geq 1, \mathcal{D}(v_i) \in \mathbf{B}$$

An *implication* is determined when all variables except 1 in the clause are implied to values, which do not add to the minimum value of the LHS of the constraint. An inconsistency is detected when the maximum value of the LHS is strictly less than the RHS.

The costliest part of any implication procedure is deciding which variable in a constraint should be implied. BCP can be highly optimized since the Boolean domain has only two values. The *unit clause rule*, states that an unassigned literal is implied to a value *if-and-only-if* all other literals in the clause evaluate to 0. As a corollary, the clause cannot imply if two or more literals are unassigned. BCP implementations using data-structures like *watched literals* [14] exploit this corollary to evaluate a clause only when an implication is necessary for consistency.

The LHS of a constraint corresponding to a clause is always equal to the number of the literals in the clause with assigned values. BCP *watches* the current value of the LHS, by *implicitly* counting the number of literals, which have been set. This corresponds to the basic idea in domain propagation in generalized FDCP. The key idea in our approach is to recognize that watching the sum of the LHS is exactly the same as using *watched literals* in BCP. FD constraints have the added problem of finding the new domain on an implied variable. While BCP has a 2-valued domain, finite-domains have a large, albeit finite set of values, which a variable can take. These are governed by implication rules [8]. The performance of the constraint propagation is directly proportional to the number of times that the rule-base is applied. Therefore, we still have the problem of finding *when* to apply these rules.

In order to mitigate this problem, we use a generalized version of *watching* for FD constraints, which is based on the idea that a term

$wsum_c$	=	$\sum_i v_g$
$wsum_{lb}$	=	$\sum_i min_{\mathcal{D}(v_i^+)} - \sum_j max_{\mathcal{D}(v_j^-)}$
$wsum_{ub}$	=	$\sum_i max_{\mathcal{D}(v_i^+)} - \sum_j min_{\mathcal{D}(v_j^-)}$

Table 2: Definition of Watched Sums for each constraint κ

is implied as soon as its bound *must* be changed for satisfying the constraint. The *current watched-sum* ($wsum_c$), *watched lower-bound* ($wsum_{lb}$), and *watched upper-bound* ($wsum_{ub}$) of a constraint are formally defined in the Table 2.

The quantity $\sum v_g$ in Table 2, represents the sum of all the variables in a constraint κ, that have been bound to a single value. v_i^+ indicates the non-ground variables with positive signs and v_j^- indicates the non-ground variables with negative signs. $wsum_c + wsum_{lb}$ represents the current minimum possible value of the LHS of the constraint. If it is greater than the RHS, then the constraint is satisfied. $wsum_c + wsum_{ub}$ represents the current maximum possible value of the LHS. Since it is calculated over the variables, which can have bound changes, this quantity can be used to check for inconsistency of the constraint and for the necessity for implications to maintain consistency of the constraint. The following rules can be deduced for each constraint based on $wsum_c$, $wsum_{lb}$, and $wsum_{ub}$:

1. κ is satisfied if $wsum_c + wsum_{lb} \geq r$.
2. κ is conflicting if $wsum_c + wsum_{ub} < r$
3. Bound change(s) need to be made on at least one variable in κ, if $wsum_c < r$ and $wsum_{ub} - wsum_{lb} - wsum_c \geq r$. κ is currently in a inconsistent state. It can be made consistent through bound changes on the domains of the free variables.

The form of the primitive clauses that we use are particularly suited for variable substitution, since variable substitution in constraints of 3 or less variables are guaranteed to be tighter [8]. Hence, we use variable substitution for equality propagation. This enables FDCP to find more implications, especially in circuits with large mux structures. Detection of equality of two variables during constraint propagation is done by detecting conditions of the form, $(a_i.w_i - a_j.w_j \geq 0) \wedge (a_j.w_j - a_i.w_i \geq 0)$. If the constant coefficients a_i, a_j are the same, then the two variables w_i, w_j are equivalent.

6. LEARNING AND CONFLICT ANALYSIS

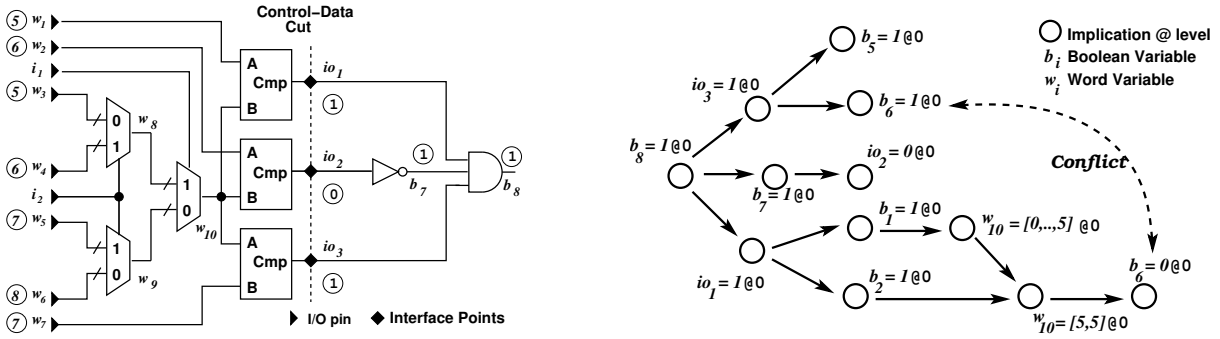
In this section, we present a learning scheme for hybrid DPLL, that drives the search into the solution space by constraining the Boolean space alone. Conflict-based learning was proposed by Stallman *et al.*, [15] as a method to learn partial assignments, which are guaranteed to cause a conflict. This was further extended by Marques-Silva *et al.*, for a CNF-based SAT solver, GRASP [13] by tracing the implication graph. The learned information is stored as a clause called a *learned* clause that prevents the assignments, which caused the conflict.

We do not trace the implication graph as in the above techniques due to complexities introduced by word-variable implications and equality propagation during FDCP. Instead, we learn conflict clauses on the interface points. We shall explain the problems associated with implication-graph-based methods of conflict analysis in HDPLL, using the example shown in Figure 5(a). We shall then describe our current method of conflict-based learning.

$io_1 \models (w_1 \equiv w_{10})$	where,	$(b_1 \models w_1 \geq w_{10}), (b_2 \models w_{10} \geq w_1)$
$io_2 \models (w_2 \equiv w_{10})$	where,	$(b_3 \models w_2 \geq w_{10}), (b_4 \models w_{10} \geq w_2)$
$io_3 \models (w_{10} \equiv w_7)$	where,	$(b_5 \models w_{10} \geq w_7), (b_6 \models w_7 \geq w_{10})$

Table 3: Modeling of Comparators in Figure 5(a).

The interface points in the circuit in Figure 5(a), are ib_1, ib_2, which are the *input* interface points (IPIs) and io_1, io_2, io_3, which are the *output* interface points (IPOs). Extra Boolean variables $b_1, \ldots, b_6$ are introduced, as described in Section 4, since the comparators are modeled as inequalities as shown in the Table 3. The circuit has the prop-



(a) Learning at the Control-Data Interface.

(b) Implications for $b_8 = 1$.

Figure 5: Example circuit and implication graph

erty that, with the value assignment $\{w_1, \ldots, w_7\} = \{5,6,5,6,7,8,7\}$, the IPOs are *one-hot* encoded.

Figure 5(b) shows the implication graph for the proposition $b_8 = 1$. We find $b_8 = 1$ implies that $io_1 = 1, io_2 = 0$, and $io_3 = 1$. Further, $io_1 = 1$ implies that $b_1 = 1, b_2 = 1$, and $io_3 = 1$ implies that $b_5 = 1, b_6 = 1$. The implications on b_1 and b_2 bind the domain of w_{10} to the value 5. However, this in turn implies that $b_6 = 0$, which conflicts with the earlier implication of $b_6 = 1$. We see that a value assignment in the Boolean logic, caused a chain of implications through the data-path, which implied a value in the Boolean logic. Ideally, we would trace through the implication graph from the conflict site, and find that the cause of the conflict, is $b_8 = 1$. This contradicts the proposition and hence, it is UNSAT.

We can see from the implication graph, that w_{10} had two value changes, one of which caused the second value change. In addition, if a value is set on the mux select line, i_1 before i_2, then equality propagation would take place. w_{10} would be replaced by w_8 or w_9. Therefore, we need to distinguish between multiple value changes (or *incarnations*) of a word-variable, and the immediate cause of the change. This complicates conflict-analysis in the implication graph. Therefore, we avoid the problem, by taking a different cut in the implication graph. This is described in the next section.

6.1 Hybrid Conflict Analysis

If some objective in the Boolean control can be satisfied by an assignment of $\{io_1, io_2, io_3\} = \{1,0,1\}$, a naive way of checking whether this is satisfiable in the data-path is to use an FME solver to check satisfiability of the data-path with this constraint. If this is unsatisfiable, then we have to back-track and repeat the process, by learning a conflict clause that avoids the assignments on the interface – a *blocking clause* [10].

We can use all Boolean values on the interface points as a *learned clause* in SAT, to drive the solver into the solution space. However, this leads to enumeration on the the interface points. So they are called *loose* learned clauses. It would be more efficient to find *tight* conflict clauses that restrict the search space in the arithmetic domain more than loose clauses. To this end, our current method finds *unique implication points* (UIPs) [13] if possible. However, while tracing paths backward from a conflict in the implication graph, we stop at assignments on variables in the set **IP**. Intuitively, the learned clause is a cut in the implication graph. We can find a "good" cut by using the methods in [13]. However, we can still use other cuts, while maintaining correctness.

In our method, the cut lies on the interface points. In this example, we would find the clause $(\overline{io_1} + \overline{io_3})$, which is part of the restriction that the interface points are 1-hot encoded. In general, we find resolvents that are larger than those corresponding to UIPs. The method is applicable only when the conflict is found through FDCP. However, if FDCP does not find a conflict, then FME is used for a final consistency check. If FME returns a conflict, then we should find as tight a clause as possible, on the interface points. This is done by the iterative loop on FME in the Figure 3. Since the assignments are processed iteratively, we find a subset of the assignments on the interface points as a learned clause.

The most significant drawback of these techniques, is that performance is now predicated on the nature of the conflict clauses found. In general, the current conflict analysis in HDPLL may not find the best clause. Considerable experimentation will have be to done before we can definitively conclude what the best strategy would be for classes of RTL circuits.

7. EXPERIMENTAL EVALUATION

In this section, we discuss some experiments on the hybrid DPLL solver. Our benchmarks are safety properties on RTL circuits drawn from the ITC'02 benchmarks (b01, b02, b04 and b11). The various test-cases are logic cones unrolled over time by 10, 15 and 20 timeframes each. The average word-size was 3-8 bits.

The experiments, shown in the Table 4, are performed on a Pentium-IV 2.0 GHz with 1GB of RD-RAM. Column 1 shows the test case name. Columns 2 and 3 shows the number of Boolean and Word level operators in each test-case. Column 4 shows the number of interface points, which is the size of the control-data partition. Column 5 shows the ratio of arithmetic to Boolean operators in each test-case. The remainder are the results of the experiments described below.

Experiment 1 (HDPLL[1]): First, we ran HDPLL with no constraint propagation, in order to evaluate the cost of a very naive algorithm. As we can see from Columns 6-8 in Table 4, the cumulative cost of FME is prohibitive. The number of calls to the FME solver is so high, since we do not efficiently bound the search space on the interface points.

Experiment 2 (HDPLL[2]): We then ran HDPLL with a limited version of constraint propagation, in order to evaluate how useful constraint propagation is in getting tighter clauses on the interface points. We expect to find conflicts in the data-path due to value assignments on the interface points and also use the implication graph to get tighter clauses on the interface points. As we can see from Columns 9-11 in Table 4, the number of calls to FME go down dramatically, with a corresponding performance improvement. However, the Boolean solver still cannot bound the search space very well since it generates a lot of assignments on the interface points, which are not satisfiable. This is because the values, which are implied on interface points due to value assignments on other interface points are not implied in the Boolean logic. Hence the Boolean solver makes a

Ckt	# Bool Ops	# Word Ops	Interface Points	Arith/Bool Ratio	HDPLL[1] Without FDCP			HDPLL[2] With limited FDCP			HDPLL[3] With unrestricted FDCP			SVC Time
					Btrks	FME	Time	Btrks	FME	Time	Btrks	FME	Time	
test1	257	186	121	0.72	29131	7780	31.62	665	1	0.71	173	1	0.05	0.83
test2	417	301	196	0.72	123672	22122	244.94	1309	1	3.23	546	1	0.19	3.8
test3	577	416	271	0.72	–	–	-to-	2024	1	11.54	1216	1	0.49	-to-
test4	241	340	226	1.41	7786	710	28.02	1776	1	5.44	224	1	0.09	11.66
test5	391	550	366	1.40	31197	6285	1176.17	3844	1	26.27	680	1	0.34	38.81
test6	541	760	506	1.40	–	–	-to-	17963	1	720.89	1506	1	0.95	-to-
test7	178	252	215	1.41	729	198	0.4	136	1	0.40	26	2	0.2	16.29
test8	283	412	355	1.45	607	214	0.6	154	2	0.70	87	4	0.62	34.17
test9	388	572	495	1.47	1404	465	1.3	218	2	2.10	123	4	1.21	54.86
test10	561	471	325	0.83	–	–	-to-	2134	1	6.55	713	1	0.49	-to-
test11	876	841	505	0.96	–	–	-to-	2879	1	20.37	1124	1	1.04	-to-
test12	1191	991	685	0.83	–	–	-to-	5181	1	102.57	1590	1	1.62	-to-
FME : Number calls to Omega library					Btrks: Number backtracks			-to- : Aborted after 3600 cpu secs (Time is in CPU seconds)						

Table 4: Performance Comparison of Hybrid DPLL with SVC

significant number of assignments, which led to data-path conflicts.

Experiment 3 (HDPLL[3]): Next, we allowed implications on the interface points forward into the Boolean control logic. Columns 12-14 in Table 4 shows that this adds a significant performance improvement over the other two approaches (HDPLL[1] and (HDPLL[2]). This clearly demonstrates the power of constraint propagation combined with conflict-based learning. The results show that the hybrid constraint solver can handle non-trivial RTL test-cases.

Experiment 4: Finally, we compared HDPLL with SVC [2], which is an implementation of a cooperating decision procedure, based on congruence closure. The results for SVC on the same benchmarks are shown in Column 15 in Table 4. SVC shows a rapid decline in performance when the size of the problem increases. The hybrid DPLL procedure can complete on all the test cases, where SVC times-out after 3600 CPU seconds. This clearly demonstrates that HDPLL is both efficient and scalable as compared to state-of-the art. SVC could complete only on those test-cases, which had more arithmetic than Boolean operators (entries > 1 in Column 4). This bears out the intuition that SVC would not scale with increasing complexity in the control. However, varying relative sizes of the control and data-path has little effect on the performance of HDPLL. Hence HDPLL appears to be more robust than SVC.

These experiments clearly demonstrate the power of constraint propagation in HDPLL. Based on the current test-cases, the hybrid search algorithm is considerably more scalable and faster than SVC [2].

8. CONCLUSIONS

We present an efficient modified DPLL constraint solver for RTL circuits. We show that FDCP can mitigate the problems involved in solving combined Boolean and arithmetic constraints. We describe a strategy of driving the hybrid search into the solution space efficiently by conflict-based learning on the control-data interface. We show that the integration of constraint propagation in arithmetic and Boolean domains provides considerable performance improvement on our benchmarks. However, the approach requires extensive testing on a variety of benchmarks, before we make any broad conclusions. The EDA applications of such a solver are considerable. We shall improve some fundamental aspects of the hybrid solver and applying it to EDA problems in future. The current investigation into HDPLL raises some issues, which we shall address in our future work.

Proof Production: The experiments demonstrates that performance depends on effective conflict-based learning across control and data-path. The conflict analysis implemented does not support unified analysis for UIP learning on a hybrid implication graph. It also does not use information from the FME procedure for better learning. We shall rectify this in our future work.

Decision Ordering: Currently, we do not make full use of structure of the data-path to guide the decision strategy. We shall investigate methods for improving this in the future.

Static Constraint Extraction: It is intuitive that control imposes most of the constraints on RTL data-path. Hence, effective static learning across control and data-path using our techniques can enable efficient constraint extraction for test and verification.

Bit-vector Logic: HDPLL currently uses bit-slicing to implement logic operations on bit-vectors. We shall improve this in future.

Sequential Search: HDPLL is currently a combinational solver. We shall extend HDPLL to sequential SAT [10] on RTL circuits.

9. REFERENCES

[1] ABRAMOVICI, M., BREUER, M. A., AND FRIEDMAN, A. D. *Digital Systems Testing and Testable Design*, $1^s t$ ed. CS Press, 1990.

[2] C. BARETT, D. L. DILL, AND J. LEVITT. Validity Checking for Combinations of Theories with Equality. In *Proc. of FMCAD* (Nov. 1996), vol. 1166 of *LNCS*, 187–201.

[3] C. W. BARRETT, D. L. DILL, AND J. R. LEVITT. A Decision Procedure for Bit-Vector Arithmetic. In 35^{th} *DAC* (1998), pp. 522–527.

[4] BRINKMANN, R., AND DRESCHLER, R. RTL-Datapath Verification using Integer Linear Programming . In *Proc. of* 15^{th} *VLSI Design Conf.* (Jan. 2001), pp. 741–746.

[5] R.E. BRYANT. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. Computers C-35*, 8 (Aug. 1986), 677–691.

[6] DANTZIG, G., AND EAVES, B. Fourier-motzkin Elimination and its Dual. *Journal of Combinatorial Theory A*, 14 (1973), 288–297.

[7] M. DAVIS, G. LOGELAND, AND D. LOVELAND. A Machine Program for Theorem Proving. *Communications of the ACM 5*, 7 (1962).

[8] W. HARVEY, AND P.J. STUCKEY. Improving Linear Constraint Propagation by Changing Constraint Representation. *Constraints 8*, 2 (2003), 173–207.

[9] IYER, M. A. RACE: A Word-Level ATPG-Based Constraints Solver System For Smart Random Simulation In *Proc. of the ITC* (2003), 259–266.

[10] M.K. IYER, G. PARTHASARATHY, AND K.-T. CHENG. SATORI – A Fast Sequential Sat Solver for Circuits. In *Proc. of ICCAD* (2003), 320–325.

[11] W. KELLY, V. MASLOW,W. PUGH, *et al.*, . The Omega Calculator and Library v1.1.0. Tech. rep., Univ. of Maryland–College Park, 1996.

[12] LIU, C., KUEHLMANN, A., AND MOSKEWICZ, M. CAMA: a Multi-Valued Satisfiablity Solver. In *Proc. of ICCAD* (2003), 326 – 333.

[13] J.P M.-SILVA, AND K.A. SAKALLAH. GRASP - A Search Algorithm for Propositional Satisfiability. *IEEE Trans. on Computers 48*, 5 (1999), 506–521.

[14] M. MOSKIEWICZ, C. MADIGAN, Y. ZHAO, L. ZHANG, AND S. MALIK. Engineering an Efficient SAT Solver. In 38^{th} *DAC* (2001).

[15] R.M. STALLMAN, AND G.J. SUSSMAN. Forward Reasoning and Dependency Directed Backtracking in a System for Computer Aided Circuit Analysis". *Artificial Intelligence 9*, 2 (1977), 135–196.

[16] STRICHMAN, O. On Solving Presburger and Linear Arithmetic with SAT. In *Proc. of FMCAD* (2002).

[17] ZHANG, L., MADIGAN, C., MOSKEWICZ, M., AND MALIK, S. Efficient Conflict Driven Learning in a Boolean Satisfiability Solver. In *Proc. of ICCAD* (Nov. 2002).