Scalable Exploration of Functional Dependency by Interpolation and Incremental SAT Solving

Chih-Chun Lee, Jie-Hong R. Jiang, Chung-Yang Huang
EE Dept./Graduate Inst. of Electronics Engineering
National Taiwan University

Alan Mishchenko
Dept. of EECS
University of California, Berkeley

ABSTRACT

Functional dependency is concerned with rewriting a Boolean function f as a function h over a set of base functions $\{g_1, \ldots, g_n\}$, i.e. $f = h(g_1, \ldots, g_n)$. It plays an important role in many aspects of electronics design automation (EDA), ranging from logic synthesis to formal verification. Prior approaches to the exploration of functional dependency are based on binary decision diagrams (BDDs), which may not be easily scalable to large designs. This paper proposes a novel reformulation that extensively exploits the capability of modern satisfiability (SAT) solvers. Thereby, functional dependency is detected effectively through incremental SAT solving and the dependency function h, if exists, is obtained through Craig interpolation. The main strengths of the proposed approach include: (1) fast detection of functional dependency with small memory consumption and thus scalable to large designs, (2) a full capacity to handle a large set of base functions and thus discovering dependency whenever exists, and (3) potential application to large-scale logic optimization with different design constraints. Experimental results show the proposed method is far superior to prior work and scales well in dealing with the largest ISCAS89 and ITC99 benchmark circuits with up to 200K gates.

Categories and Subject Descriptors

B.6.3 [**Logic Design**]: Design Aids -- Automatic Synthesis, Optimization, Verification; J.6 [**Computer-Aided Engineering**]: Computer-aided design (CAD).

General Terms

Algorithms, Optimization, Synthesis, Verification

Keywords

BDD, Boolean Satisfiability, Craig Interpolation, Functional Dependency, Refutation Proof

1. INTRODUCTION

Functional dependency [1] appears commonly among a set of Boolean functions $\{f_1, ..., f_n\}$ in VLSI circuit design as a function f_i (called the *target function*) can often be reexpressed as some function h (called the *dependency function*) over a subset of the functions (called the *base functions*). The exploration of functional dependency plays an important role in many aspects of EDA, ranging from logic synthesis to formal verification. For instance, it leads to the identification of redundant registers in RTL synthesis [2][3], resubstitution and simplification of Boolean functions in both technology-independent and technology-dependent logic synthesis [4], BDD minimization [5] and state space reduction [1][6][7] in formal verification, search space reduction in SAT solving [8], etc. Advances on the exploration of functional dependency may benefit a wide range of applications.

Given a set of Boolean functions $\{f_1, \ldots, f_n\}$, we would like to know if any f_i can be written as $h(f_1, \ldots, f_{i-1}, f_{i+1}, \ldots, f_n)$. Conventional approaches [1] to the exploration of functional dependency rely mostly on BDDs [9]. Unfortunately computation using BDDs suffers from the memory explosion problem and thus is not scalable to manipulate large designs. In contrast, SAT solving consumes little memory (linear in the input size) at the cost of time resources and thus is more robust at least in representing large designs. Recent advances, see e.g. [10][11], in SAT solving have made it a very efficient Boolean reasoning engine and a viable alternative to BDD. More and more logic synthesis and verification algorithms shift their computation paradigm from BDD to SAT, e.g. [12][13]. However, formulating the computation of functional dependency as pure SAT solving is not straightforward due to the difficulty in deriving the dependency function h, whose derivation in BDD-based computation is in contrast immediate.

This paper demonstrates, for the first time, that the exploration of functional dependency (including efficient derivation of dependency function) can be achieved with pure SAT solving. In particular, a dependency function, if exists, can be obtained through the construction of interpolants from a refutation proof of a SAT solver. Essentially, Craig interpolation theorem [14] lays the foundation. Moreover, to detect functional dependency for different target functions and to obtain different dependency functions for a target function, incremental SAT solving is adopted to reuse learned clauses. Experimental results show encouraging improvements over BDD-based approaches.

The main results of the paper include (1) a new SAT-based derivation of dependency function using Craig interpolation, which enables a pure SAT solution to the exploration of functional dependency, and (2) an incremental SAT-based enumeration of target and base functions, which effectively reduces the search space for solving similar SAT instances. Practical experience shows that a pure SAT formulation of functional dependency avoids the BDD memory explosion problem and is scalable to large designs. It is powerful in detecting functional dependency even among a large set of base functions.

The paper is organized as follows. After preliminaries are introduced in Section 2, our SAT formulation of functional dependency is detailed in Section 3. The proposed approach is evaluated with experimental results in Section 4. Section 5 concludes this paper and outlines some future research directions.

2. PRELIMINARIES

As a notational convention, in the sequel symbols "$\wedge$", "$\vee$", and "$\neg$" denote Boolean AND, OR, and COMPLEMENT operations, respectively. The cardinality (or size) of a set S is denoted as |S|. The problem formulation of functional dependency and some background on SAT solving are given as follows.

2.1 Functional Dependency

Functional dependency is defined as follows.

Definition 1. *Given a Boolean function f:* $\boldsymbol{B}^m \rightarrow \boldsymbol{B}$ *and a vector of Boolean functions* $G = (g_1(X), ..., g_n(X))$ *with* g_i*:* $\boldsymbol{B}^m \rightarrow \boldsymbol{B}$ *for* $i = 1, ..., n$*, over the same set of variable vector* $X = (x_1, ..., x_m)$*, we say that f* **functionally depends** *on G if there exists a Boolean function h:* $\boldsymbol{B}^n \rightarrow \boldsymbol{B}$*, called the* **dependency function***, such that* $f(X) = h(g_1(X), ..., g_n(X))$*. We call functions f, G, and h the* **target function***,* **base functions***, and* **dependency function***, respectively.*

Note that functions f and G are over the same domain in the definition. Moreover, h needs not depend on all of the functions in G.

The necessary and sufficient condition of the existence of the dependency function h is given as follows.

Proposition 1. [1] *Given a target function f and base functions G, let* $h^0 = \{y \in \boldsymbol{B}^n: y = G(x) \text{ and } f(x) = 0, x \in \boldsymbol{B}^m\}$ *and* $h^1 = \{y \in \boldsymbol{B}^n: y = G(x) \text{ and } f(x) = 1, x \in \boldsymbol{B}^m\}$*. Then h is a feasible dependency function if and only if* $\{h^0 \cap h^1\}$ *is empty. In this case,* h^0*,* h^1*, and* $\boldsymbol{B}^n \backslash \{h^0 \cup h^1\}$ *are the off-set, on-set, and don't-care set of h, respectively.*

By Proposition 1, one can not only determine the existence of a dependency function, but also deduce a feasible one.

To explore functional dependency for a given circuit netlist, there are many choices of f and G. One may ask how to effectively choose G for a specific f.

Definition 2. *For a Boolean function f with input variables* $X = (x_1, ..., x_m)$*, variable* x_i *is a* **support variable** *of f if* $f(x_1, ..., x_{i-1}, 0, x_{i+1}, ..., x_m) \neq f(x_1, ..., x_{i-1}, 1, x_{i+1}, ..., x_m)$*.*

For a functional vector $G = (g_1, g_2, \ldots, g_n)$, its support variables are the union of the supports of g_i for $i = 1, \ldots, n$.

Proposition 2. [1] *There exists a feasible dependency function of f with respect to G only if, for every support variable v of f, v is also a support variable of G.*

Proposition 2 can be used for fast screening in selecting the base functions G.

2.1.1 BDD-based Exploration of Functional Dependency

Conventional BDD-based exploration of functional dependency is reviewed in order to contrast with the novel SAT-based approach.

Proposition 1 suggests a way of determining the existence of a dependency function and its derivation. Essentially standard image computation applies. Let y_i be the output variable of g_i. Then the on-set, off-set, and dc-set of h can be derive by

$$h^0(Y) = \exists X\, [\, R(X, Y) \wedge (f(X) \equiv 0)\,],$$

$$h^1(Y) = \exists X\, [\, R(X, Y) \wedge (f(X) \equiv 1)\,], \text{ and}$$

$$h^{dc}(Y) = \neg(h^0 \vee h^1),$$

respectively, where relation $R(X, Y) = (y_1 \equiv g_1(X)) \wedge (y_2 \equiv g_2(X)) \wedge \cdots \wedge (y_n \equiv g_n(X))$. The dependency function $h(Y)$ exists if $h^0(Y) \wedge h^1(Y)$ = false. All of the above operations can be done using BDDs.

Note that constructing the relation $R(X, Y)$ along with the image computation may suffer from memory explosion especially when $|G|$ is large even though the final BDDs of h^0 and h^1 can be small. Therefore, it is necessary to restrict the size of the set of base functions at the cost of losing completeness. Keeping $|G|$ small may often result in the nonexistence of the dependency function. Once the search for a feasible dependency function with respect to a set of base functions is failed, another set of base functions is selected and the computation of functional dependency repeats. Consequently, although some fast filtering techniques, e.g. by Proposition 2, are available [1], BDD-based computation is inefficient in that there may be too many selections of G tested before functional dependency is discovered. As will be seen later, the deficiency can be overcome in SAT-based exploration of functional dependency.

2.2 Propositional Satisfiability

Let $V = \{v_1, \ldots, v_k\}$ be a finite set of Boolean variables. A **literal** l is either a Boolean variable v_i or its negated form $\neg v_i$. A **clause** c is a disjunction of literals. Without loss of generality, we shall assume there are no repeated or complementary literals in the same clause. A **SAT instance** is a conjunction of clauses, i.e., in the so-called **conjunctive normal form** (**CNF**). In the sequel, a clause set $C = \{c_1, ..., c_k\}$ shall mean to be the CNF $(c_1 \wedge \cdots \wedge c_k)$. An **assignment** over V gives every variable v_i a Boolean value either true or false. A SAT instance is **satisfiable** if there exists a satisfying assignment such that the CNF valuates to true. Otherwise it is **unsatisfiable**. Given a SAT instance, the **satisfiability** (**SAT**) **problem** asks whether it is satisfiable or not. A SAT solver is designated to solve the SAT problem.

Definition 3. *Assume literal v is in clause* c_1 *and* $\neg v$ *in* c_2*. A* **resolution** *of clauses* c_1 *and* c_2 *on variable v yields a new clause c containing all literals in* c_1 *and* c_2 *except for v and* $\neg v$*. The clause c is called the* **resolvent** *of* c_1 *and* c_2*, and variable v the* **pivot variable**.

Proposition 3. *A resolvent c of* c_1 *and* c_2 *is a logical consequence of* $c_1 \wedge c_2$*, that is,* $c_1 \wedge c_2$ *implies c.*

Theorem 1. [15] *For an unsatisfiable SAT instance, there exists a sequence of resolution steps leading to an empty clause.*

Theorem 1 can be easily proved by Proposition 3 since an unsatisfiable SAT instance must imply a contradiction. Often only a subset of the clauses of the SAT instance participates in the resolution steps leading to an empty clause.

Definition 4. *A* **refutation proof** *Π of an unsatisfiable SAT instance C is a directed acyclic graph (DAG) Γ = (N, A), where every node in N represents a clause which is either a root clause in C or a resolvent clause having exactly two predecessor nodes, and every arc in A connects a node to its ancestor node. The unique leaf of Π corresponds the empty clause.*

Modern SAT solvers, such as Chaff [10] and MiniSat [11], are capable of producing a refutation proof from an unsatisfiable SAT instance.

2.2.1 Refutation Proof and Craig's Interpolation

Theorem 2. (Craig Interpolation Theorem) [14] *Given two inconsistent clause sets A and B (i.e. the clause set* $A \cup B$ *is*

unsatisfiable), then there exists a Boolean formula $A^{\#}$ referring only to the common input variables of A and B such that $A \Rightarrow A^{\#}$ and $A^{\#} \Rightarrow \neg B$.

The Boolean formula $A^{\#}$ is referred to as the **interpolant** of A and B. Detailed exposition on how to construct an interpolant from a refutation proof in linear time can be found in [16][17][18]. Note that the so-derived interpolant is in a circuit structure, which can then be converted into CNF as discussed below.

2.2.2 Circuit to CNF Conversion

Given a circuit netlist, it can be converted to a CNF in such a way that the satisfiability is preserved. The conversion is achievable in linear time by introducing some intermediate variables [19][20].

3. SAT-BASED EXPLORATION OF FUNCTIONAL DEPENDENCY

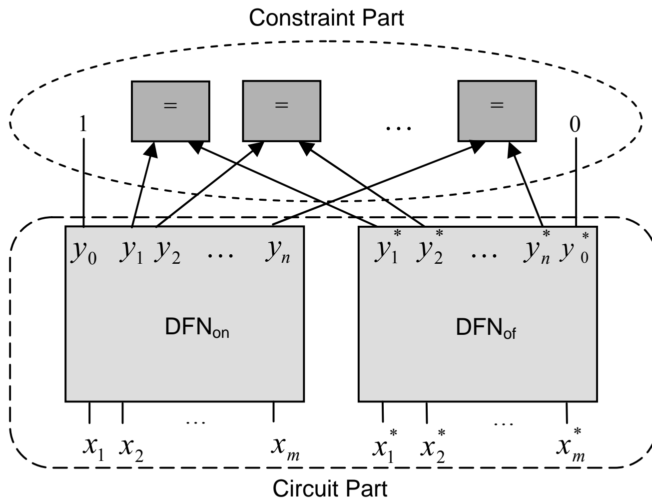

Figure 1. Dependency Function Network.

3.1 The Primary Construct

To formulate the exploration of functional dependency as SAT solving, we introduce the **dependency function network** (**DFN**) as shown in Fig. 1. For a given circuit netlist consisting of $n + 1$ Boolean functions $\{f_0, \ldots, f_n\}$, suppose function f_0 and the others are identified to be the target function and base functions, respectively. Then, in the notation of Section 2, f_0 corresponds to f and f_i corresponds to g_i for $i = 1, \ldots, n$. The circuit netlist is instantiated into two copies, identified as DFN$_{on}$ and DFN$_{off}$, in the DFN. For every variable v in DFN$_{on}$, there is starred counterpart v^* in DFN$_{off}$. Let y_i and y_i^* be the output variables of f_i and f_i^*, respectively. The circuits DFN$_{on}$ and DFN$_{off}$ can be converted to CNFs C_{on} and C_{off}, respectively, in linear time. In addition, the output of the target function f in DFN$_{on}$ is asserted to true, i.e., $y_0 = 1$; that of f^* in DFN$_{off}$ is asserted to false, i.e., $y_0^* = 0$. Furthermore, equality constraints $(y_i \equiv y_i^*)$ are imposed for $i = 1, \ldots, n$. Thereby the entire CNF C_{DFN} is

$$C_{on} \wedge C_{off} \wedge y_0 \wedge \neg y_0^* \wedge (y_1 \equiv y_1^*) \wedge \cdots \wedge (y_n \equiv y_n^*), \quad (1.1)$$

where $(y_i \equiv y_i^*)$ is the shorthand for $(y_i \vee \neg y_i^*) \wedge (\neg y_i \vee y_i^*)$.

The intuition behind this construct is that formula $C_{on} \wedge y_0$ (respectively $C_{off} \wedge \neg y_0^*$) imposes the constraint that the valuations over input variables $(x_1, \ldots, x_m)$ (respectively $(x_1^*, \ldots, x_m^*)$) must be the on-set of f (respectively off-set of f^*). By Proposition 1, we can thus test if h^0 and h^1 are disjoint. Precisely speaking, we conclude the following.

Theorem 3. *Given a target function f and a set of base functions g_i for i=1, ..., n, a dependency function h exists if and only if the CNF C_{DFN} of the corresponding DFN is unsatisfiable.*

Proof: ($\Rightarrow$) By Definition 1, f can be expressed as $h(g_1(X), \ldots, g_n(X))$. Proposition 1 asserts that the onset h^1 and offset h^0 of h must be disjoint. Observe that h^0 and h^1 are essentially the sets of satisfying assignments of variables y_i of DFN$_{on}$ and y_i^* of DFN$_{off}$, respectively. Hence CNF C_{DFN}, which is the conjunction of C_{on} and C_{off}, is unsatisfiable.

($\Leftarrow$) For C_{DFN} is unsatisfiable, there are two cases. For the first case, C_{on} or C_{off} is unsatisfiable. It happens only when f is a constant function. C_{on} (C_{off}) is unsatisfiable if and only if f is constant zero (one). In this case, we may express f as a function over any base functions. For the second case, C_{on} and C_{off} are both satisfiable. Then unsatisfiable C_{DFN} implies its clauses of the equality constraints $(y_i \equiv y_i^*)$ are violated. That is, the sets of images of the onset and offset of f under the base functions are disjoint. By Proposition 1, we know that h must exist. ■

In the sequel we shall assume a target function is non-constant.

Remark 1. Note that, although DFN is similar to the miter structure used in combinational equivalence checking, the underlying principle is completely different. The DFN construct differs from the miter structure in that: Firstly, the sets of input variables of DFN$_{on}$ and DFN$_{off}$ are disjoint. Secondly, the output variables of the target functions of DFN$_{on}$ and DFN$_{off}$ are asserted to true and false, respectively. Thirdly, the equality constraints are imposed only on the corresponding pairs of base functions.

We show how the dependency function can be derived using interpolation provided that the clause set C_{DFN} is unsatisfiable. To apply Theorem 2, consider partitioning the clause set C_{DFN} into two subsets A and B. We claim the following.

Corollary 1. *For unsatisfiable $C_{DFN} = A \wedge B$ with $A = C_{on} \wedge y_0$ and $B = C_{off} \wedge \neg y_0^* \wedge (y_1 \equiv y_1^*) \wedge \cdots \wedge (y_n \equiv y_n^*)$, the resultant interpolant $A^{\#}$ derived from a refutation proof yields a desired dependency function h.*

Proof: Observe that the common variables of A and B are $Y = (y_1, \ldots, y_n)$, which is desirable for the dependency function. Since $A \wedge B$ is unsatisfiable, by Theorem 2 there exists an interpolant $A^{\#}$ which refers only to Y. In addition, conditions $A \Rightarrow A^{\#}$ and $A^{\#} \Rightarrow \neg B$ suggest that the set of valuations over variables Y satisfying $A^{\#}$ must be an over-approximation of $h^1(Y)$ and must be disjoint from $h^0(Y)$. Hence, $A^{\#}(Y)$ is a valid implementation of the dependency function $h(Y)$ with respect to the underlying target and base functions. ■

Therefore, as long as a SAT solver can produce an interpolant from a refutation proof, it can be exploited to generate the dependency function. The overall algorithm of the exploration of functional dependency is sketched in Fig. 2.

```
FunctionalDependencyBySAT
 input: target function f and base functions {g1, …, gn}
 output: a dependency function h
 begin
 01  Construct clause set C_DFN
 02  if (C_DFN is UNSAT)
 03    Partition C_DFN into clause sets A and B
 04    Derive an interpolant A# from refutation proof
 05    return A#
 06  return no solution
 end
```

Figure 2. Algorithm: Functional dependency by SAT.

Note that the choice of clause sets A and B is not unique. For instance, letting $A = C_{on} \wedge y_0 \wedge (y_1 \equiv y_1^*) \wedge \cdots \wedge (y_n \equiv y_n^*)$ and $B = C_{off} \wedge \neg y_0^*$ is valid as well. In fact, different refutation proofs and different choices of A and B can be exploited to obtain the flexibilities implementing the dependency function.

Remark 2. One reason making our SAT-based approach outperform BDD-based ones is due to the efficiency in getting base functions. In our method, we can simply include all candidate base functions rather than carefully select a subset of the candidate functions. Therefore, our method can detect functional dependency in one run. However, BDD-based methods may require multiple runs.

3.2 Incremental SAT Solving

The above discussion assumes the target function is given. However, for a given circuit netlist, there may be many different choices of the target function. Often we need to detect functional dependency for different target functions one at a time. Consider we have explored the functional dependency for target function f_0 and base functions $\{f_1, \ldots, f_n\}$. Suppose now we want to switch the target function to f_1 and add f_0 to the base functions. Only slight modification is needed migrating from the original SAT instance, C_{DFN0}, to the new one, C_{DFN1}, because the sets of base functions are almost the same. Since the search spaces for the two SAT instances are very similar, incremental SAT solving [21] is helpful in amortizing the computation overhead. We investigate how to incorporating incremental SAT solving in our framework by reusing helpful clauses learned from solving previous SAT instances in subsequent computation.

Since not all previously learned clauses are valid to inherit in solving the current SAT instance, invalid clauses need to be disabled. To avoid sophisticated clause removal, we adopt the MiniSat [11] interface, where *unit assumptions* [11] can be made on a list of literals such that the subsequent SAT solving is restricted to the specified solution subspace and the assumptions are discharged upon return. We introduce auxiliary variables and make unit assumptions on them to enable or disable "dynamic clauses." Let the auxiliary variable controlling clause c be α_c. We replace c in the original SAT instance with the new clause $(\alpha_c \Rightarrow c)$ such that c is enabled (disabled) when $\alpha_c = 1$ (0). Because in our case these "dynamic clauses" are resulted from the equality constraints of C_{DFN}, we introduce auxiliary variables α_i as the switches of the conditional satisfaction of equality constraints $(y_i \equiv y_i^*)$. For $\alpha_i = 1$ (0), equality constraint $y_i \equiv y_i^*$ is turned on (off). For f_i to be the target function and the others the base functions, the entire CNF C_{DFN} of Eq. (1.1) now becomes

$$C_{on} \wedge C_{off} \wedge y_i \wedge \neg y_i^* \wedge (\alpha_0 \Rightarrow (y_0 \equiv y_0^*)) \wedge (\alpha_1 \Rightarrow (y_1 \equiv y_1^*)) \wedge \cdots \wedge (\alpha_n \Rightarrow (y_n \equiv y_n^*)) \wedge \alpha_0 \wedge \cdots \wedge \alpha_{i-1} \wedge \neg\alpha_i \wedge \alpha_{i+1} \wedge \cdots \wedge \alpha_n.$$

Again to compute interpolants, we partition the above clause set into two subsets A and B with

$A = C_{on} \wedge y_i$ and

$B = C_{off} \wedge \neg y_i^* \wedge (\alpha_0 \Rightarrow (y_0 \equiv y_0^*)) \wedge (\alpha_1 \Rightarrow (y_1 \equiv y_1^*)) \wedge \cdots \wedge (\alpha_n \Rightarrow (y_n \equiv y_n^*)) \wedge \alpha_0 \wedge \cdots \wedge \alpha_{i-1} \wedge \neg\alpha_i \wedge \alpha_{i+1} \wedge \cdots \wedge \alpha_n$.

To check the satisfiability of the above C_{DFN}, the unit clauses $\{y_i, \neg y_i^*, \alpha_0, \cdots, \alpha_{i-1}, \neg\alpha_i, \alpha_{i+1}, \cdots, \alpha_n\}$ will be on the unit assumptions of MiniSat. Effectively, the SAT solving is restricted to the solution subspace with $y_i = 1$, $y_i^* = 0$, $\alpha_i = 0$ and $\alpha_j = 1$ for $j \neq i$. If the result is satisfiable, no functional dependency exists under the target function. Otherwise, a conflict clause is returned, which refers only to a subset of the auxiliary variables in addition to the output variables y_i and y_i^* of the target function. It indicates some of the unit assumptions are self-contradicting. Hence there exists a dependency function that depends only on the corresponding base functions. This property gives a quick answer if we are interested only in the input size of the dependency function. On the other hand, from this clause we may construct an interpolant under the contradicting solution subspace and thus derive the dependency function.

3.3 Enumeration of Different Dependency Functions

For a fixed target function f functionally depending on a set of base functions, it is often the case that the don't care set $\boldsymbol{B}^n \backslash \{h^0 \cup h^1\}$ for the dependency function h is not empty. Hence there is flexibility implementing h differently. Obtaining these don't cares is preferable. However, the capability of SAT solvers is limited in this respect as they tend to find "a" satisfying assignment or "a" refutation proof. A refutation proof uniquely determines an interpolant and, thus, an implementation of the dependency function. To overcome this deficiency, we propose two methods identifying two different types of alternatives of a dependency function implementation: those with the same support variables and those with different ones. For the former, we reorder the resolution sequence of a refutation proof to obtain different interpolants and, thus, different implementations of the dependency function. For instance, the approach in [22] is one way of doing it. By proper strengthening and weakening the interpolants, we may obtain a subset of the don't cares. However, practical experience suggests that the so obtained don't care set may not be large. For the latter, we block the SAT solver from searching the same instance and enforce it to search a new refutation proof with a different set of support variables. It can be done by making proper unit assumptions under the MiniSat interface.

4. EXPERIMENTAL RESULTS

The proposed algorithm was implemented in ABC [23] modified to equip with the proof-logging version of MiniSat [11]. All the experiments are conducted on a 3.2GHz Linux machine with 2GB memory. The experiments are designed so as to demonstrate

1. the efficiency and scalability of SAT-based in contrast to BDD-based computation [1],
2. the benefit incremental SAT formulation, and
3. the characteristics of the derived dependency functions.

Table 1. SAT- vs. BDD-based Exploration of Functional Dependency.

		Original			Retimed			SAT (original)		BDD (original)		SAT (retimed)		BDD (retimed)	
Circuit	#Nodes	#FF.	#Dep-S	#Dep-B	#FF.	#Dep-S	#Dep-B	Time	Mem	Time	Mem	Time	Mem	Time	Mem
s5378	2794	179	52	25	398	283	173	1.2s	18m	1.6s	20m	0.6s	18m	7s	51m
s9234.1	5597	211	46	x	459	301	201	4.1s	19m	x	x	1.7s	19m	194.6s	149m
s13207.1	8022	638	190	136	1930	802	x	15.6s	22m	31.4s	78m	15.3s	22m	x	x
s15850.1	9785	534	18	9	907	402	x	23.3s	22m	82.6s	94m	7.9s	22m	x	x
s35932	16065	1728	0	--	2026	1170	--	176.7s	27m	1117s	164m	78.1	27m	--	--
s38417	22397	1636	95	--	5016	243	--	270.3s	30m	--	--	123.1	32m	--	--
s38584	19407	1452	24	--	4350	2569	--	166.5s	21m	--	--	99.4s	30m	1117s	164m
b12	946	121	4	2	170	66	33	0.15s	17m	12.8s	38m	0.13s	17m	2.5s	42m
b14	9847	245	2	--	245	2	--	3.3s	22m	--	--	5.2s	22m	--	--
b15	8367	449	0	--	1134	793	--	5.8s	22m	--	--	5.8s	22m	--	--
b17	30777	1415	0	--	3967	2350	--	119.1s	28m	--	--	161.7s	42m	--	--
b18	111241	3320	5	--	9254	5723	--	1414.9s	100m	--	--	2842.6s	100m	--	--
b19	224624	6642	0	--	*	*	*	8184.8s	217m	--	--	*	*	*	*
b20	19682	490	4	--	1604	1167	--	25.7s	28m	--	--	36	30m	--	--
b21	20027	490	4	--	1950	1434	--	24.6s	29m	--	--	36.3	31m	--	--
b22	29162	735	6	--	3013	2217	--	73.4s	36m	--	--	90.6	37m	--	--

("--": memory usage exceeds 1Gb. "x": runtime exceeds 10000 seconds. "*": circuit cannot be retimed using ABC [23].)

Large circuits from the ISCAS89 and ITC99 benchmark suits are chosen. To have fair comparison with [1], functional dependency among the transition functions of a circuit is explored. Among the transition functions of a given circuit, each of them is specified in turn as the target function and all others as base functions. We then explore the corresponding functional dependency and compute dependency functions if exist.

Table 1 compares our approach with the prior work [1]. Columns 1 and 2 respectively list the name and the number of nodes of each circuit. The numbers of flip-flops, denoted #FF, of a circuit and its retimed version are listed in Columns 3 and 6, respectively. Among the flip-flops of an original circuit (respectively a retimed circuit), those whose transition functions possess functional dependency are counted in Columns 4 and 5 (respectively Columns 7 and 8), denoted as #Dep. In particular, #Dep-S and #Dep-B are obtained by the SAT- and BDD-based methods, respectively. In fact, #Dep-S data are exact and complete except for retimed b19 circuit, which is unavailable from ABC. In comparison, the BDD-based method only succeeded in a few circuits and detected only a subset of the dependency over a few support variables.) The runtime (in seconds) and memory (in Megabytes) usage are shown in the following columns. Note that the reported memory usage includes the underlying system memory whereas the prior work was built on VIS [24] and ours on ABC. Despite the uneven comparison, the scalability of our approach is evident and outperforms the prior work.

The strength of incremental SAT solving is shown in Fig. 3. The x-axis and y-axis, respectively, represent the iteration number and the runtime of solving a SAT instance at that particular iteration. The y-axis is log-scaled. Five sample circuits of different sizes from Table 1 are plotted for the first 100 iterations. As can be seen in all of the plots, the runtimes of the first iterations are the maximum of among their first 100 iterations. In fact, all the circuits of Table 1 exhibit the same behavior. After the first iteration, the runtimes for SAT solving decrease rapidly and become relatively short and stable within about 10 iterations. It demonstrates the effectiveness for incremental SAT solving.

The experiments tend to suggest that (1) the average runtime for a circuit is linear in the number of its nodes and (2) the solving time for an unsatisfiable SAT instance is often much shorter than that for a satisfiable one. The statistics are plotted in Fig. 4. The x-axis and y-axis, respectively, represent the number of nodes of each circuit and the average runtime of SAT iterations. Both axes are log-scaled. Every circuit in Table 1 is plotted as a spot in Fig. 4. The first tendency can be seen from the two regression lines indicating highly (positive) correlated data set. The second tendency can be seen from the fact that the line for the retimed circuits is well below that for the original circuits. As evident in Columns 4 and 7 of Table 1, more functional dependency exists for the retimed circuits. Effectively, more unsatisfiable SAT instances are there. It reflects the fact that in our experiments a satisfiable instance usually takes longer time to solve than an unsatisfiable one. It seems contradicting with common sense. However, the tendency can be explained as follows. Because the input sizes of interpolants are mostly very small (to be shown in Fig. 5), it suggests that conflicts can be found locally. Also, incremental SAT solving increases *implicativity* [25] and thus enhances early conflict detection. Thus, decisions over only a few variables might be enough to draw an unsatisfiable conclusion. In contrast, in a satisfiable case to obtain a satisfying assignment, decisions must be made over all variables.

We characterize the derived dependency functions in terms of their input sizes in Fig. 5, where a single dependency function, if exists, is derived for each transition function of a given circuit. The x-axis and y-axis indicate the numbers of support variables and of dependency functions, respectively. The y-axis is log-scaled. As can be seen, most of the functions have less than 10 support variables. It demonstrates the fact that the derived interpolants are mostly small. On the other hand, complex functional dependency can also be detected easily by the SAT-based approach. For instance, in the retimed b18 circuit, a

dependency function of input size around 300 is obtained, which is not possible using BDD-based methods.

From practical experience in enumerating different dependency functions for a target function, we note that the number of available dependency functions (with different support sets) varies greatly from function to function. A great amount of trivial dependency exists due to the transitivity of dependency. It results in vast redundant enumeration. How to effectively avoid unnecessary enumeration remains to be done. Nevertheless, if the candidate base functions are specified (e.g. for circuit rewiring), finding a dependency function is easy. On the other hand, we emphasize that the BDD-based method is more effective than the SAT-based one in computing the don't care set for a dependency function. This deficiency of the SAT-based computation is due to the fact that an interpolant (i.e. a dependency function) is with respect to a refutation and contains no don't-care information.

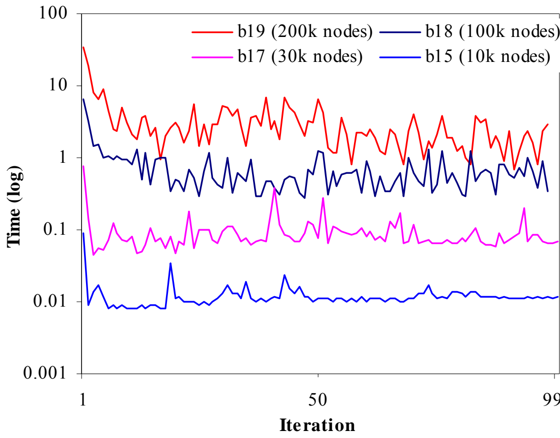

Figure 3. Runtime of the first 100 SAT iterations.

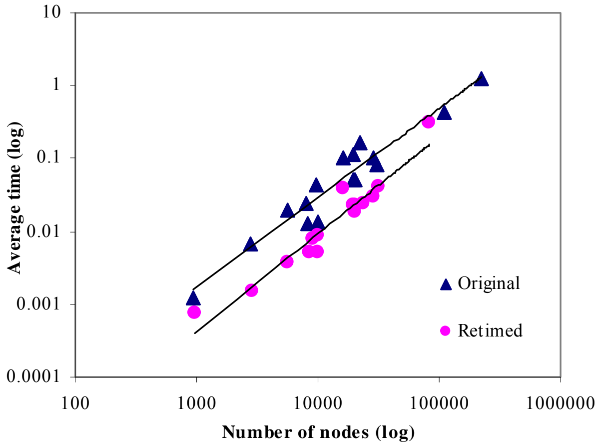

Figure 4. Average runtime for SAT iterations.

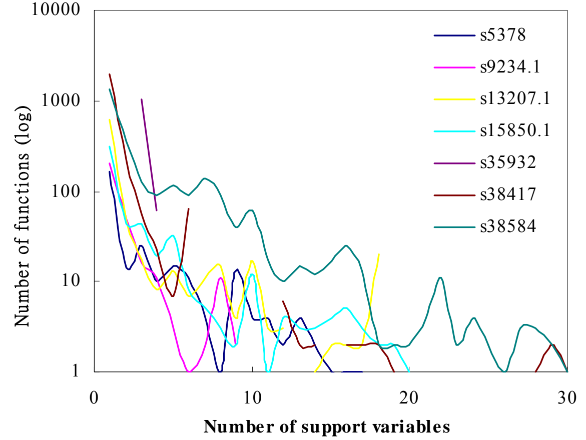

Figure 5. Frequency distribution of different support sizes.

5. RELATED WORK

The previous efforts closest to ours are [1] and [4]. Both of them rely on BDD-based computation. In [1] combinational functional dependency was generalized to sequential dependency. Here we focus on the combinational one. In [4], similar to our enumeration for different dependency functions, a BDD-based technique was used. It allows a more implicit enumeration. However, the size of the set of base functions was limited to no more than 16. Other work [18] applied interpolation in the context of SAT-based model checking for approximated image computation.

6. CONCLUSIONS AND FUTURE WORK

We have shown that the exploration of functional dependency can be solved by a pure SAT-based formulation. Experimental results demonstrated the great success of the proposed method. The approach is scalable to large designs and discovers much more functional dependency far beyond the capability of prior methods. The success is attributed to several key ingredients including Craig interpolation and incremental SAT solving. We hope that our results may benefit several areas of logic synthesis and formal verification, for example, in finding rewiring and resubstitution candidates for circuit optimization, in identifying redundant registers in RTL synthesis, in reducing state space in formal sequential equivalence checking, etc. Future work includes integrating our technique in logic synthesis and generalizing it for other applications. In addition, it is interesting to explore new applications of Craig interpolation.

ACKNOWLEDGMENTS

The authors would like to thank Prof. Robert Brayton for helpful discussions and comments. This work was supported in part by NSC grants 95-2221-E-002-432 and 95-2218-E-002-064-MY3. AM was supported by the Intel-custom SRC grant 1444.001 "Innovative Sequential Synthesis and Verification."

REFERENCES

[1] J.-H. R. Jiang and R. K. Brayton. Functional dependency for verification reduction. In *Proc. CAV*, pp. 268-280, 2004.

[2] E. Sentovich, H. Toma, and G. Berry. Latch optimization in circuits generated from high-level descriptions. In *Proc. ICCAD*, pp. 428-435, 1996.

[3] B. Lin and A. R. Newton. Exact redundant state registers removal based on binary decision diagrams. In *Proc. Int'l Conf. Very Large Scale Integration*, pp. 277-286, 1991.

[4] V. Kravets and P. Kudva. Implicit enumeration of structural changes in circuit optimization. In *Proc. DAC*, pp. 438-441, 2004.

[5] A. J. Hu and D. L. Dill. Reducing BDD size by exploiting functional dependencies. In *Proc. DAC*, pp. 266-271, 1993.

[6] C. Berthet, O. Coudert, and J.-C. Madre. New ideas on symbolic manipulations of finite state machines. In *Proc. ICCD*, pp. 224-227, 1990.

[7] C. A. J. van Eijk and J. A. G. Jess. Exploiting functional dependencies in finite state machine verification. In *Proc. European Design & Test Conf.*, pp. 9-14, 1996.

[8] E. Gregoire, R. Ostrowski, B. Mazure, and L. Sais. Automatic extraction of functional dependencies. In Proc. SAT, 2004.

[9] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Trans. Computers*, pp. 677--691, August 1986.

[10] M. Moskewicz, C. Madigan, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proc. DAC*, pp. 530-535, 2001.

[11] N. Eén and N. Sörensson. An extensible SAT-solver. In *Proc. SAT*, pp. 502-518, 2003.

[12] A. Mishchenko and R. Brayton. SAT-based complete don't-care computation for network optimization. In *Proc. DATE*, pp. 418-423, 2005.

[13] A. Mishchenko, J. Zhang, S. Sinha, J. Burch, R.K. Brayton, and M. Chrzanowska-Jeske. Using simulation and satisfiability to compute flexibilities in Boolean networks. *IEEE Trans. on CAD*, vol. 25, no. 5, pp. 742-755, 2006.

[14] W. Craig. Linear reasoning: A new form of the Herbrand-Gentzen theorem. *J. Symbolic Logic*, 22(3):250-268, 1957.

[15] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23-41, 1965.

[16] J. Krajicek. Interpolation theorems, lower bounds for proof systems, and independence results for bounded arithmetic. *J. Symbolic Logic*, 62(2):457-486, June 1997.

[17] P. Pudlak. Lower bounds for resolution and cutting plane proofs and monotone computations. *J. Symbolic Logic*, 62(3):981-998, September 1997.

[18] K. L. McMillan. Interpolation and SAT-based model checking. In *Proc. CAV*, pp. 1-13, 2003.

[19] G. Tseitin. On the complexity of derivation in propositional calculus. *Studies in Constructive Mathematics and Mathematical Logic*, pp. 466-483, 1970.

[20] D. Plaisted and S. Greenbaum. A structure preserving clause form translation. *J. Symbolic Computation*, vol. 2, pp. 293-304, 1986.

[21] J. Whittemore, J. Kim, and K. Sakallah. SATIRE: A new incremental satisfiability engine. In *Proc. DAC*, 2001.

[22] R. Jhala and K.L. McMillan, "Interpolant-based transition relation approximation". *Proc. CAV*, pp. 39-51, 2005.

[23] Berkeley Logic Synthesis and Verification Group. ABC: A System for Sequential Synthesis and Verification. Release 51205. http://www.eecs.berkeley.edu/~alanmi/abc/

[24] R. K. Brayton, *et al*. VIS: a system for verification and synthesis. In *Proc. CAV*, pp. 428--432, 1996.

[25] Y. Novikov and R. Brinkmann. Foundations of Hierarchical SAT-Solving. In *Proc. Int'l Workshop on Boolean Problems*, 2004.