Enhancing Iterative Layering with SAT Solvers

Ana Petkovska*, David Novo*, Ajay K. Verma*, Alan Mishchenko‡ and Paolo Ienne*

* Ecole Polytechnique Fédérale de Lausanne (EPFL)
School of Computer and Communication Sciences
CH–1015 Lausanne, Switzerland
{ana.petkovska, david.novobruna, ajaykumar.verma, paolo.ienne}@epfl.ch

‡ University of California, Berkeley
Department of EECS
alanmi@eecs.berkeley.edu

***Abstract*—Iterative Layering is an existing pre-synthesis technique that can be used for constructing a near-optimal representation of the input circuit. Originally, the Iterative Layering algorithm uses SAT instances extensively. However, in the original implementation, they are solved either by exhaustive enumeration of all possible assignments of input variables, or by using BDDs. Both approaches perform poorly, which constrains the size of the circuits that can be processed. In this paper, we propose a new implementation of the algorithm in which all SAT problems are reformulated and solved by a modern SAT solver. Moreover, we use the unsatisfiability proofs produced by the SAT solver and Craig interpolants to fundamentally change the way Iterative Layering handles circuit reconstruction. As a result, our enhanced Iterative Layering promises to scale to larger circuits than the original one. Unfortunately, this preliminary work still does not show evidence of superior overall runtime; accordingly, some parts of the current heuristic will need to be reviewed in future work.**

***Index Terms*—Iterative Layering, SAT, Interpolation, Logic Synthesis, Logic Design, ABC.**

I. INTRODUCTION

THE complexity of circuits has grown with transistor density, to the point that humans can no longer reason about them. Thus, it is crucial to automatically optimize those circuits during logic synthesis, and to minimize their delay and area. Existing heuristics for logic optimization try to find an equivalent near-optimal representation of the input circuit. One such pre-synthesis technique is the Iterative Layering technique presented by Verma et al. [1], which serves as reference for this work.

Iterative Layering is an iterative algorithm, which does not try to do local optimizations; rather, it performs more global optimizations and eagerly seeks for useful computations, namely bricks. These *bricks* are small pieces of computation, with a few inputs and a single output bit. They are judged by their ability to reduce further computation later on and are placed as early as possible to maximize parallelism. In the last iteration, Iterative Layering produces a new circuit composed of a set of bricks that compute the same information as the input circuit with a new, smaller structure.

A. K. Verma is currently affiliated with Google Inc., Germany.

The Iterative Layering algorithm consists of two main phases, namely brick generation and brick selection, which are repeated iteratively. The process of *brick generation* is the first phase of the Iterative Layering algorithm. In each iteration, it computes candidate bricks for the corresponding layer. Before it delivers the set of candidate bricks to the brick selection phase of the algorithm, it ensures that the generated set contains all required information about the input bits of the circuit. The set of generated bricks contains redundant bricks. The *brick selection* phase, selects only the subset of the received candidate bricks that minimize resource utilization. Once the chosen set of bricks contains enough information about the circuit function, a layer is formed from the bricks, and the dependency function is computed. Afterwards, the algorithm uses this dependency function as an input to the next iteration, which generates and selects the bricks for the next layer. Figure 1 shows an example of a function synthesized with the Iterative Layering algorithm.

In this paper we present a new implementation of Iterative Layering, which makes extensive use of modern SAT solvers, contrary to the reference implementation. We propose using SAT-based functional dependency checks instead of BDDs that turns out to be significantly faster, especially for large circuits. Moreover, we improve the calculation of the brick selection metric. Our metric is computed based on the area of the logically optimized circuit, obtained with Craig interpolation, and not on the size of the partially complete BDD. In most situations, this improves the heuristic responsible for discovering more efficient circuits. Figure 2 presents the flow of the proposed Iterative Layering algorithm.

The rest of the document is organized as follows. First, in Section II, we give background on functional dependency, the satisfiability problem, and the Craig interpolation. Next, in Section III and Section IV, we describe in detail the brick generation and the brick selection phase of the algorithm, respectively. Section V presents the experimental results. Finally, Section VI draws conclusions and proposes some future work.

II. BACKGROUND INFORMATION

In this section, we give the definitions of functional dependency, satisfiability problem, and interpolant.

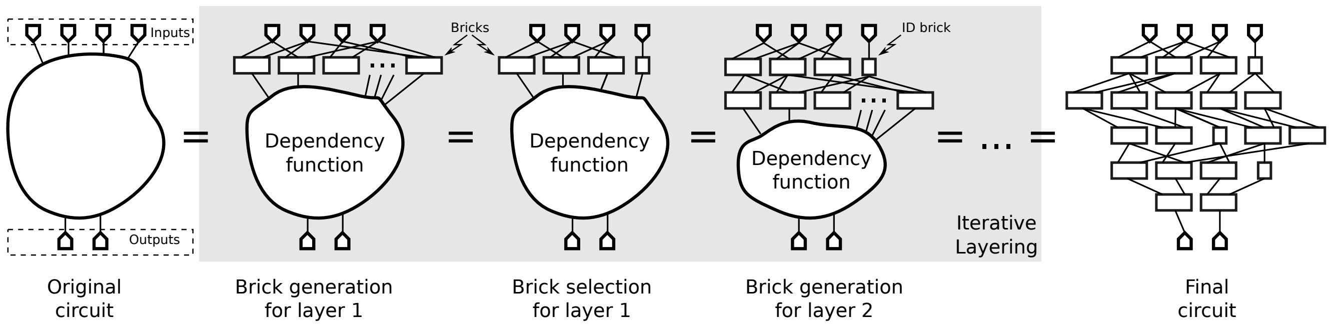


Fig. 1: Reconstruction of a function with four primary inputs, and two primary outputs, by applying the Iterative Layering algorithm. The final circuit, composed from five layers of bricks, is functionally equivalent to the original circuit.

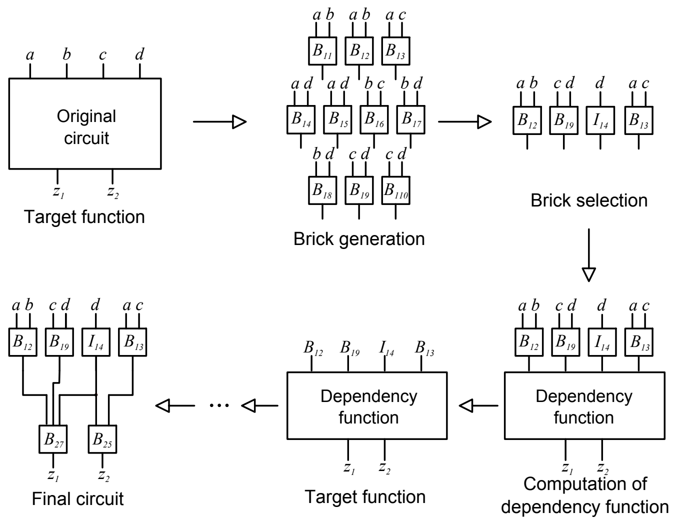


Fig. 2: The new flow between the main parts of the Iterative Layering algorithm. As an additional step, we propose computing the dependency function once the bricks for the current layer are selected. In the next iteration, the dependency function is considered as a target function.

A. Functional Dependency

The check for functional dependency is used extensively in the two phases of the Iterative Layering algorithm. Primarily, it ensures that a layer can be formed from a given set of bricks, which is true if and only if the input circuit *functionally depends* on the set of bricks. On the other hand, the necessary and sufficient condition for existence of functional dependency helps in defining the construction of the circuits from which the interpolant is built. We use the definition of functional dependency proposed by Jiang et al. [2].

Definition 1: Given a Boolean function $f : B^m \to B$ and a vector of Boolean functions $G = (g_1(X), ..., g_n(X))$ with $g_i : B^m \to B$ 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 $h : B^n \to B$, called the *dependency function*, such that $f(X) = h(g_1(X), ..., g_n(X))$.

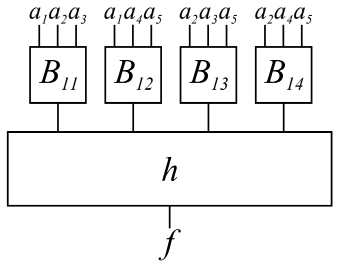


Fig. 3: Representation of the target function $f(a_1, a_2, a_3, a_4, a_5)$ as a function h of the set of bricks selected for the first layer $G = \{B_{11}, B_{12}, B_{13}, B_{14}\}$. If the function f functionally depends on the set of bricks G, then the dependency function h can be computed.

In Iterative Layering, the functions f, g_i, and h correspond to the *target function*, a *brick*, and the *dependency function*, respectively. Figure 3 shows the target function f with its dependency function $h(B_{11}, B_{12}, B_{13}, B_{14})$.

The following example explains functional dependency. Consider the target function $f = a \oplus b$, the set of inputs $X = (a, b)$, and the set of bricks $G = (g_1(X), g_2(X))$, which correspond to

$$g_1(X) = a + b, \text{ and } g_2(X) = ab.$$

Since the target function f can be written as

$$f = h(g_1(X), g_2(X)) = g_1(X)\overline{g_2(X)},$$

it follows that the dependency function $h(G)$ exists, and thus, f functionally depends on the set of bricks G.

The set of bricks G, on which the target function is functionally dependent, may include some *redundant bricks*. For example, a third brick $g_3(X) = a\bar{b}$ can exist in the set G, but it can be excluded from the dependency function h.

Next, we define what an assignment is, in order to set the necessary and sufficient condition for functional dependency used later.

Definition 2: An assignment to a finite set of Boolean variables, $V = (v_1, ..., v_k)$, is the mapping $V \to \{0, 1\}$.

The necessary and sufficient conditions for functional dependency are defined with the following theorem [3].

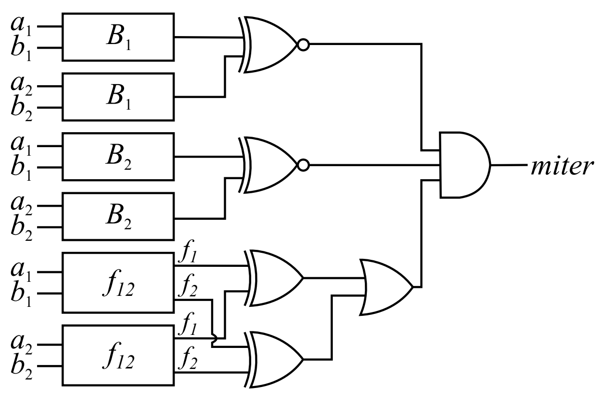


Fig. 4: The circuit constructed for checking if the function f_{12}, with two outputs f_1 and f_2, and two inputs a and b, functionally depends on the set of bricks $G = \{B_1, B_2\}$. Functional dependency exist if and only if the miter evaluates to 0 for any two assignments of the primary inputs.

Theorem 1: For any two assignments $P = (p_1,..., p_m)$ and $Q = (q_1, ..., q_m)$ of the variable vector $X = (x_1,..., x_m)$, if the output value of $f(P)$ differs from the output value of $f(Q)$, then in the vector of Boolean functions $G = (g_1(X),..., g_n(X))$ exist at least one brick $g_i(X)$, for $i = 1, ..., n$, such that the output value of $g_i(P)$ differs from the output value of $g_i(Q)$.

To check for functional dependency, we construct a circuit called *miter* and we give it to a SAT solver. As example, Figure 4 shows a miter constructed for a multiple-output target circuit. The miter evaluates to 1 if and only if two assignments for the primary inputs P and Q exist, for which each brick evaluates to the same value and at least one output of the function f evaluates to a different value. If an output of the function evaluates to a different value, then it can not be represented as a function of the given set of bricks G, and there is no functional dependency. Otherwise, if the miter evaluates to 0 for all possible assignments of the primary inputs, then f functionally depends on the set of bricks G.

B. Satisfiability Problem

J.-H. R. Jiang et al. [2] have shown that the exploration of functional dependency can be done efficiently, even for large circuits, by formulating it as a satisfiability (SAT) problem of a linear size and using modern SAT solvers. In this subsection, the SAT problem is defined, and the terms needed for its definition are introduced.

A *literal*, l, is either a variable $l = v$ or its negation $l = \overline{v}$. A disjunction (OR, $+$) of literals form a *clause*, $c = l_1 + ... + l_k$. A *propositional formula* or a *Boolean formula* is a logic expression defined over variables that take value in the set $\{0, 1\}$. To solve a SAT problem, the propositional formula is converted into its *conjunctive normal form* (CNF) as a conjunction (AND, $\cdot$) of clauses, $F = c_1 \cdot ... \cdot c_k$. [5]

Definition 3: The *satisfiability (SAT) problem* is a decision problem that as input receives a propositional formula in CNF form and decides whether there exists a *satisfying assignment* of the variables from the formula, for which the CNF evaluates to 1. If a satisfying assignment exists, then it is said the propositional formula is *satisfiable* (*SAT*). Otherwise, it is *unsatisfiable* (*UNSAT*).

The algorithms that solve SAT problems are called *SAT solvers*. SAT solvers produce either a *satisfying assignment* when the problem is satisfiable, or a *proof of unsatisfiability* when the problem is unsatisfiable. The proof of unsatisfiability is also called a *refutation proof* and is used for building the interpolant.

C. Interpolant

The interpolation theorem for first-order logic was first proved by W. Craig [6] in 1957.

Theorem 2: An *interpolant* for the pair of subsets of clauses (A, B), such that $A \cdot B$ is unsatisfiable, is a formula P with the following properties:

- A implies P;
- $P \cdot B$ is unsatisfiable;
- P refers only to the common variables of A and B.

An *interpolation system* is a procedure for constructing an interpolant for a pair of subsets of clauses (A, B) from their refutation proof. Given the pair (A, B) and their refutation proof, the interpolant may be constructed in linear time. Today, modern SAT solvers, such as MiniSat [7], can compute a refutation proof from an unsatisfiable instance and build an interpolant out of it. Next, we define what a refutation proof is and we explain the construction of the interpolant through McMillan's system [8].

Definition 4: A *refutation proof* Π of a set of clauses C is a directed acyclic graph (V_Π, E_Π), where E_Π is set of edges connecting the vertices with their predecessor vertices, and V_Π, the set of vertices, is a set of clauses such that

- for every vertex $c \in V_\Pi$, c is either a *root clause*, such that $c \in C$, or c is an *intermediate clause* and represents the resolvent of its two predecessors c_1 and c_2;
- the unique *leaf vertex* is an empty clause.

Definition 5: [8] Let (A, B) be a pair of clause sets and let Π be their refutation proof. For all vertices $c \in V_\Pi$, let $p(c)$ be a Boolean formula, such that

- if c is a root clause, and
  - if $c \in A$, then $p(c) = g(c)$, where $g(c)$ is the disjunction of the global literals in c, or
  - if $c \notin A$, then $p(c) = 1$;
- else, if c is an intermediate clause, then let c_1 and c_2 be the predecessors of c, and let x be their pivot variable. Then,
  - if $x \in B$, then $p(c) = p(c_1) \cdot p(c_2)$, and
  - if $x \notin B$, then $p(c) = p(c_1) + p(c_2)$,

the Π-interpolant of (A, B) is $p(0)$.

The Boolean circuit representing the interpolant is constructed by substituting the intermediate vertices and the leaf with gates corresponding to the executed operation between their predecessors.

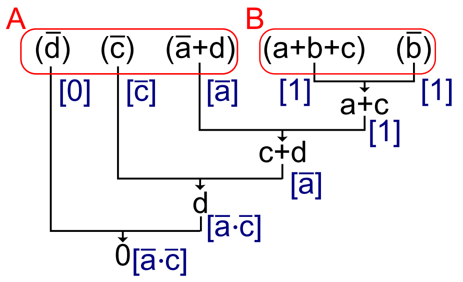


Fig. 5: Computing the interpolant of $A = (\bar{d}) \cdot (\bar{c}) \cdot (\bar{a} + d)$ and $B = (a + b + c) \cdot (\bar{b})$ using McMillan's algorithm. The interpolant is equal to the Boolean formula assigned to the leaf clause $p(0) = \bar{a} \cdot \bar{c}$.

Figure 5 shows how the interpolant for $A = (\bar{d})\cdot(\bar{c})\cdot(\bar{a}+d)$ and $B = (a+b+c)\cdot(\bar{b})$ is constructed by following McMillan's algorithm.

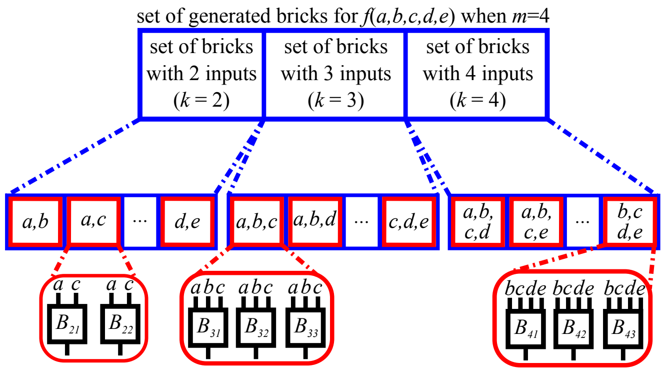
III. Brick Generation

The brick generation is the first phase of the Iterative Layering algorithm. In each iteration, it computes the candidate bricks for the corresponding layer. In this section, we explain the whole process in detail.

Assume that the input circuit has n primary inputs. The Iterative Layering algorithm receives as input the maximal number of primary inputs for the bricks m, such that $m < n$ and $m \leq 6$. The final set of generated bricks is composed of $m - 1$ subsets, G_k, where $2 \leq k \leq m$. G_1 corresponds to the set of *identity bricks*, which have only one bypassed input and have no logic, thus it is not generated explicitly. Each G_k subset provides a complete set of bricks with different number of primary inputs. Although the set G_k is generated for bricks with k inputs, after performing logic optimisation on the bricks, it may also contain bricks with less than k inputs. Figure 6 shows the contents of the set of bricks generated for a function f with five primary inputs: a, b, c, d, and e, when the maximum number of primary inputs m is 4. The G_k sets are represented in the first row and below each set its subsets are shown.

If we want to generate bricks for the subset with k inputs, then for each possible combination of k inputs, we generate a set of bricks as cofactors in the following way.

1) First, we start with an empty set, and we generate a given number of bricks by giving a random assignment on the remaining $(n - k)$ primary inputs. By propagating these values, we obtain a *cofactor* circuit. For a multiple-output input circuit, we generate a multiple-output cofactor. Since the bricks have only one output, the cofactor function of each output is saved as a separate brick. Thus, the number of generated bricks from one assignment equals the number of outputs of the input circuit.
2) Next, since the sampling is random, it is easy to miss some irredundant cofactors. Therefore, after a set of bricks is formed, we ensure that the generated set contains all required information about the circuit. Thus, we check if the input circuit functionally depends on the given bricks joined with the identity brick for the primary inputs that are not used as inputs for the bricks. For this check, we construct a miter as the one in Figure 4 and we give it to the SAT solver.
3) If the target function functionally depends on the set of bricks, then the algorithm saves the set and proceeds with the generation of bricks for the next combination of primary inputs. Otherwise, from the satisfying assignment received from the SAT solver, we extract an assignment for the generation of an additional brick. Since the miter includes two copies of the primary inputs of the target circuit, the received satisfying assignment contains two assignments P and Q. The assignment, from which we generate an additional brick, contains the values for the primary inputs that are identical in both P and Q. For example, if the returned satisfying assignment for a function $f(a, b, c, d, e)$ contains the assignments $P = (1, 0, 0, 0, 1)$ and $Q = (1, 1, 0, 1, 1)$, then by assigning and propagating $a = 1$, $c = 0$, and $d = 1$ we generate a brick with two inputs b and d. Steps 2 and 3 are repeated until there is a functional dependency from the generated set of bricks.



set of generated bricks for $f(a,b,c,d,e)$ when m=4
set of bricks with 2 inputs (k = 2)
set of bricks with 3 inputs (k = 3)
set of bricks with 4 inputs (k = 4)
a,b
a,c
...
d,e
a,b,c
a,b,d
...
c,d,e
a,b, c,d
a,b, c,e
...
b,c d,e
a c
a c
B_{21}
B_{22}
abc abc abc
B_{31}
B_{32}
B_{33}
bcde bcde bcde
B_{41}
B_{42}
B_{43}

Fig. 6: The subsets of generated bricks for one layer for the function f with five primary inputs a, b, c, d, and e, when $m = 4$. The three sets with different numbers of primary inputs are indicated with squares. Each of these sets is composed from complete sets of bricks for all combinations of primary inputs, which are indicated with ovals. The function f functionally depends on each of the oval sets.

Figure 7 shows the flowchart of the described brick generation process for one combination of inputs.

This process guarantees that the target function functionally depends on the union of the set of bricks generated for one combination of the primary inputs and the unused primary inputs. Since all sets generated for different combinations of the inputs are complete, the union of the sets for all combinations is also a complete set.

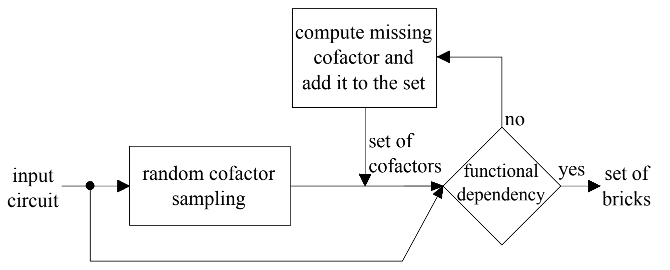


Fig. 7: Brick generation process for one combination of inputs. By using the SAT solver to check for functional dependency between the generated bricks and the input circuit, we ensure that we generate a complete set of bricks for each input combination. Until there is no functional dependency, the returned satisfying assignment from the SAT solver is used for computing the missing irredundant cofactors. These cofactors are added into the set of sampled cofactors, and the complete set of cofactors is saved as a set of bricks.

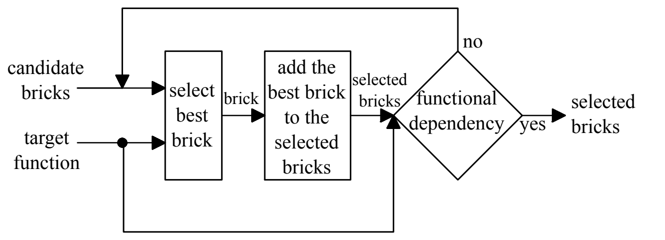


Fig. 8: In the brick selection phase of the Iterative Layering algorithm, the bricks used for the current layer are selected one by one. The brick selection algorithm stops adding bricks once the target function is functionally dependent on the set of selected bricks.

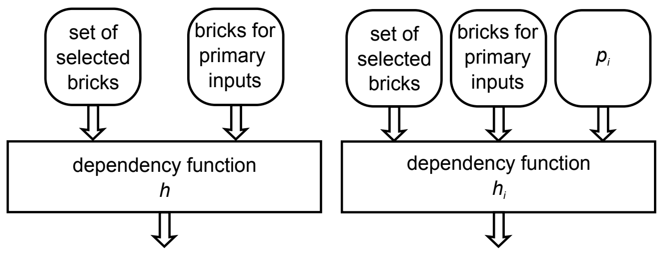
A. Differences from the Original Brick Generation Algorithm

In the original Iterative Layering algorithm, instead of generating sets of cofactors with different number of primary inputs, only one complete set of cofactors is generated for each combination of primary inputs. Then, in order to find smaller bricks from the cofactors, a DAG implementing all cofactors for one set of inputs is constructed and several cuts are found. As output is given the set of bricks that correspond to the signals above the best cut in terms of reduction of resources. This process is faster than our brick generation, however, it may fail to generate some good bricks or it may discard them with the chosen cut.

IV. Brick Selection

The set of generated bricks contains redundant bricks. Thus, in the brick selection phase, a subset of the generated bricks is chosen that is sufficient to represent the function and that brings maximal reduction of resources.

The brick selection phase begins with the set of bricks received from the brick generation phase. It chooses the best brick from the set and adds it to the set of selected bricks that will be returned. Once a brick is added, it is marked as



(a) (b)

Fig. 9: The two dependency functions constructed for computing the information fitness of the brick p_i. The dependency function h is same for all bricks in the layer. The information fitness of p_i is computed as $\text{IF}(p_i) = \alpha - (\beta + \gamma)$, where $\alpha = \text{size}(h)$, $\beta = \text{size}(h_i)$ and $\gamma = \text{size}(p_i)$. The function size$(x)$ returns the number of AND nodes of the logically optimised circuit x.

used and is ignored in the next iteration of the brick selection algorithm. After choosing a brick, we check whether the target circuit for which the bricks were generated functionally depends on the chosen set of bricks. If it depends, then the target circuit can be represented as a function of the bricks, and we return the chosen set of bricks. Otherwise, at least one brick is missing, so we perform another iteration of the brick selection algorithm, which searches for the next best brick.

A. Selecting Good Bricks

For selecting the best brick, we use an *information fitness* metric. The *information fitness* of a brick represents the reduction in resource utilization when this brick is used to compute the original circuit. The information fitness of a brick p_i is computed using the formula $\alpha - (\beta + \gamma)$ where:

- The parameter α is the size of the dependency function h that as inputs has the outputs of the set of already selected bricks and the primary inputs of the input circuit.
- The parameter β is the size of the dependency function h_i that as inputs has the outputs of the set of already selected bricks, the primary inputs of the input circuit, plus the new brick p_i.
- The parameter γ is the size of the brick p_i.

Since the information fitness depends on the bricks contained in the set of selected bricks, it is recomputed for each brick once a brick is selected and added to the set. Figure 9 shows the dependency functions generated for computing the information fitness of one brick.

Bricks with higher information fitness lead to larger resource reductions. Thus, the best brick is the one with the highest information fitness. However, sometimes in the set of generated bricks all unused bricks have information fitness less than or equal to 0. If the information fitness of the best brick is exactly 0 and if its size is bigger than 0, then the brick is not an identity brick, and it does not increase the size of the dependency function, so we add it to the selected set.

However, if the best brick has both size and information fitness of 0, then we have encountered an identity brick.

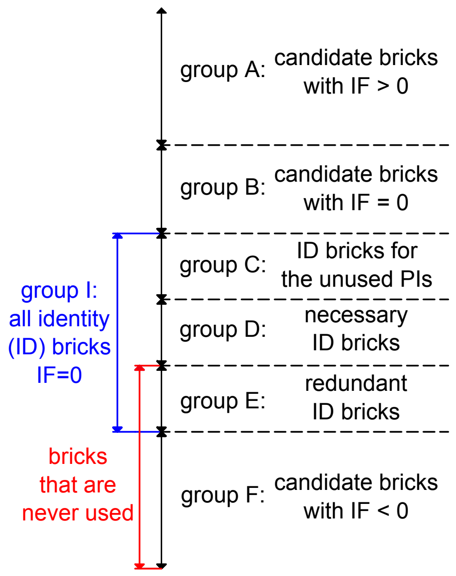

group A: candidate bricks with IF > 0
group B: candidate bricks with IF = 0
group C: ID bricks for the unused PIs
group D: necessary ID bricks
group E: redundant ID bricks
group F: candidate bricks with IF < 0
group I: all identity (ID) bricks IF=0
bricks that are never used

Fig. 10: The bricks are ranked by their information fitness and priority in the brick selection algorithm. The selected bricks belong in one of the groups A, B, C, or D. The bricks from the group E are never selected because the bricks from the group D have priority, and once we add them, we have a functional dependency. The bricks from the group F are never selected because they increase the size of the dependency function.

In cases when we have already selected bricks with higher information fitness, we want to ensure that these bricks are used by the dependency function. Thus, we add just the identity bricks which are necessary for achieving a functional dependency, in the following way.

1) First, add an identity brick for each primary input that is not a primary input of any of the selected bricks.
2) If all primary inputs are used, we add the necessary identity bricks. To check whether a brick i is necessary, we construct a miter for checking functional dependency that as bricks has the set of selected bricks and all identity bricks, except the brick i. If there is a functional dependency, the information about the brick i is already contained in the bricks from the selected set, thus i is discarded. Else, if there is no functional dependency, brick i is necessary, and it is added to the set. Once all necessary identity bricks are added, the set is complete.

Adding the identity bricks ensures that bad bricks, which have size greater than 0 and information fitness less than 0, are never added. Since the identity bricks have information fitness equal to 0, they have priority over the bad bricks. If all bricks in the set are bad bricks, then all identity bricks are added to the selected set. The target function always functionally depends on the set with all identity bricks, so this guarantees that bad bricks are never added.

Figure 10 shows the different types of bricks, ranked according to their priority in the brick selection algorithm.

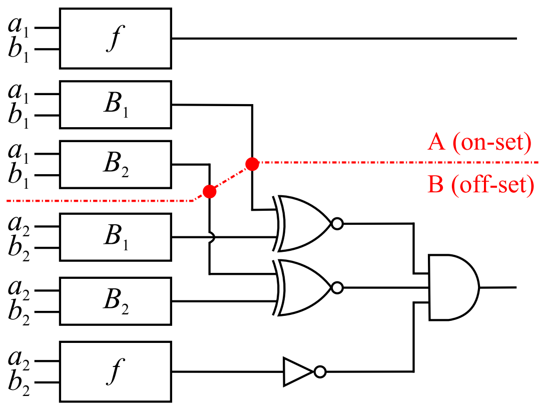


Fig. 11: The circuit used for computing single-output dependency function $h(B_1, B_2)$, such as $f(a,b) = h(B_1, B_2)$. The dots on the division line depict the outputs of the bricks which are common variables of the on-set A and the off-set B. Thus, they represent candidate inputs for the dependency function.

B. Multiple-Output Dependency Function via Interpolation

In this subsection we describe the process for generating a dependency function, which is used in the brick selection algorithm for computing the information fitnesses. After the bricks for one layer are selected, we also compute the dependency function used as target function in the next iteration.

J.-H. R. Jiang et al. [2] suggested using interpolation for re-expressing a target function with some dependency function over other base functions, which is also used by C.-H. Lin et al. [9]. For the proposed algorithm, we use a construction similar to the proposed one, by considering the bricks as base functions. Figure 11 shows the construction required for computing the dependency function of the target function f in terms of the set of bricks $G = \{B_1, B_2\}$. This circuit has the same property as the miter shown in Figure 4, and can be used for the functional dependency check of a single-output function. Thus, if f functionally depends on the set of bricks G, we give the circuit to a SAT solver which produces a refutation proof, from which we derive the interpolant. Since the outputs of the bricks belong both to the on-set A and to the off-set B, they represent candidate inputs for the interpolant, and implicatively for the dependency function.

However, as shown in Section II-C, the interpolant can only be computed for single-output functions. Thus, for multiple-output functions, we create a separate circuit for each output. Then, we compute the corresponding interpolants and we finally combine them in one circuit, which is returned as output.

C. Differences from the Original Brick Selection Algorithm

In the original Iterative Layering algorithm, the brick selection, besides information fitness, uses also another metric called information coverage, which measures the information from the target circuit that is contained in each brick. This metric will be implemented in future work.

On the other hand, the information fitness was computed by constructing a partial BDD of the dependency function and measuring its size. This BDD was based on a partial random sampling of the assignment space, which cannot guarantee that the BDD is faithfully expressing the dependency function. Instead, in this paper we propose to use the Craig interpolant to derive the exact dependency function, which enables a more accurate computation of the information fitness of a brick.

Another significant difference in the brick selection phase with respect to the original algorithm is the addition of the identity bricks to ensure that bad and unnecessary bricks are never included in the set of selected bricks.

V. Results

In this section we introduce our experimental setup, we compare the proposed SAT-based functional dependency check and the reference BDD-based one, and we present the area, delay and runtime results of our algorithm.

A. Experimental Setup

We implemented the enhanced Iterative Layering algorithm described in this paper as a new command into ABC [10]. *ABC* is an open-source software system for synthesis, technology mapping, and formal verification of sequential Boolean logic circuits used in synchronous hardware design. ABC relies heavily on the *And-Inverter Graph* (AIG) data structure, which represents a multi-level logic network composed of two-input AND gates and inverters. AIGs enable short runtimes and high-quality results for synthesis, mapping and verification due to their simplicity and flexibility.

Another advantage is that the modern SAT solver MiniSat [7] is integrated into ABC. MiniSat provides both the satisfying assignment when the problem is SAT, and the proof of unsatisfiability when the problem is UNSAT, and both are essential to Iterative Layering.

B. SAT-based vs. BDD-based Functional Dependency Check

As described in Section III and IV, the functional dependency check is used extensively in both the brick generation and in the brick selection phase of the Iterative Layering algorithm. Thus, an improvement of the running time of the functional dependency check will lead to significant improvement of the running time of the whole Iterative Layering algorithm. In the original implementation of Iterative Layering, the SAT instance derived for the functional dependency miter for small circuits is solved by exhaustive enumeration of all possible assignments of input variables. However, due to scalability problems, BDDs are used to solve big instances.

To compare the proposed SAT-based and the old BDD-based method, we compute the bricks for the first layer of an adder. We modified the brick generation phase to generate all bricks from SAT assignments, and we ensure that the functional dependency check is triggered after each computed brick. We also implemented the BDD-based method which was used in the original implementation of Iterative Layering. We next compute the overall runtime of all checks done with

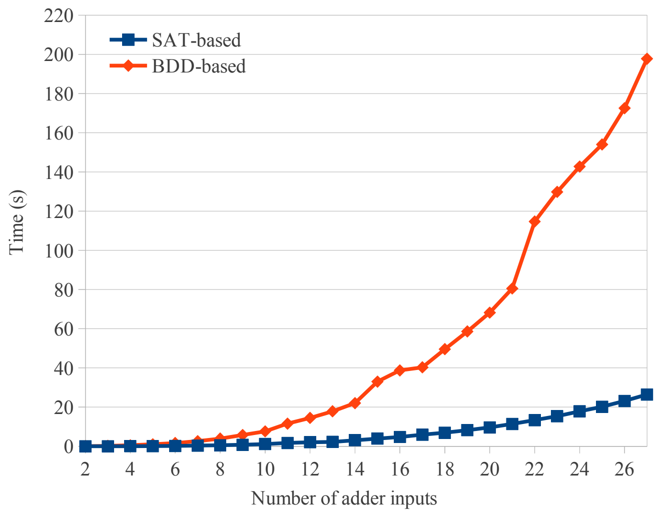


Fig. 12: Comparison of the SAT-based and the BDD-based functional dependency check, which shows the time spent on checking for functional dependency in the brick generation of the first layer of different sized adders.

both the BDD-based method and the SAT-based method for 2-input bricks. To see how the runtime changes depending on the circuit size, we gradually expand the adder size from 2-bit to 27-bit operands. Figure 12 shows that the runtimes of both methods grow approximately exponentially with the circuit size, however, the SAT-based method's runtime grows much slower than the BDD-based method's runtime, resulting in more than 7x speedup for the larger circuits.

C. Preliminary Results

The first results of our Iterative Layering algorithm are shown in Table I. We run the algorithm on ripple-carry adders of different sizes and on the 5-bit majority function for various number of brick inputs. For each run we compute the number of AND nodes (metric related to the circuit area), the maximum number of logic levels (metric related to the circuit delay) and the total runtime divided in brick generation and brick computation time.

The area results show that the proposed implementation, using the new information fitness metric, generally reduces the circuit size—in some cases by up to 50%. In few cases, however, the circuit area grows. Future work will look at these cases and determine the cause of the unwanted behavior.

Instead, the delay of the resulting circuits increases in most of the cases. This is primarily due to the singular focus of the chosen metric function on area reduction and its lack of consideration for the delay of the selected bricks.

Finally, despite having improved the runtime of the functional dependency check, the overall runtime of our algorithm is excessive and only allows the transformation of very small circuits. This is mainly because the brick generation phase produces a number of candidate bricks which grows exponentially with the number of inputs. Table I shows that for bigger bricks

Benchmark	# Inputs	# Outputs	Original		# Brick	This Work		Time (s)
			# ANDs	# Levels	Inputs	# ANDs	# Levels	Total (Brick Gen.)
5-bit Majority	5	1	25	11	2	14 **(-44%)**	5 **(-54.55%)**	1.87 (41.71%)
					3	12 **(-52%)**	6 **(-45.45%)**	5.67 (53.26%)
					4	12 **(-52%)**	5 **(-54.55%)**	10.19 (62.12%)
2-bit Adder	4	3	13	4	2	11 **(-15.38%)**	4 **(0%)**	0.99 (63.64%)
					3	11 **(-15.38%)**	4 **(0%)**	1.62 (69.75%)
					4	11 **(-15.38%)**	4 **(0%)**	1.91 (72.77%)
3-bit Adder	6	4	22	6	2	18 **(-18.18%)**	7 **(16.67%)**	4.70 (55.32%)
					3	28 **(27.27%)**	8 **(33.33%)**	9.74 (59.86%)
					4	19 **(-13.64%)**	6 **(0%)**	9.78 (65.54%)
					5	22 **(0%)**	8 **(33.33%)**	12.37 (68.71%)
4-bit Adder	8	5	31	8	2	29 **(-6.45%)**	9 **(12.50%)**	17.33 (40.05%)
					3	27 **(-12.90%)**	9 **(12.50%)**	33.27 (54.16%)
					4	30 **(-3.23%)**	8 **(0%)**	56.68 (62.74%)
					5	30 **(-3.23%)**	8 **(0%)**	82.38 (67.21%)
					6	30 **(-3.23%)**	8 **(0%)**	95.12 (68.84%)
5-bit Adder	10	6	40	10	2	38 **(-5.00%)**	9 **(-10.00%)**	44.42 (34.56%)
					3	40 **(0%)**	11 **(10%)**	98.82 (44.02%)
					4	45 **(12.50%)**	14 **(40.00%)**	168.52 (68.15%)
					5	37 **(-7.50%)**	12 **(20.00%)**	309.38 (67.61%)
					6	37 **(-7.50%)**	12 **(20.00%)**	473.68 (71.28%)
6-bit Adder	12	7	49	12	2	-	-	-

TABLE I: Preliminary results derived by running the proposed Iterative Layering algorithm on adders with different operand sizes, and on the 5-bit majority function.

the time required for brick generation is increases dramatically. Moreover, the high number of generated bricks also impacts the time required for brick selection, as more bricks need to be evaluated. As a result, this first implementation of our new SAT-based formulation of Iterative Layering leaves open plenty of future work which is discussed in the next section.

VI. Conclusions and Future Work

This paper describes a novel implementation of the Iterative Layering algorithm which extensively leverages modern SAT-solvers. We show that SAT-based functional dependency check can potentially improve the runtime of the algorithm; however, other inefficiencies on the current implementation hinders its impact on the overall runtime. Also, we make use of the Craig interpolant to compute a more accurate information fitness metric, which leads to a better selection of bricks and a reduction of the circuit size.

However, the first experimental results indicate different directions to improve the current implementation of our Iterative Layering algorithm. Our main concern is the current runtime, which results from the overzealous brick generation approach that adversely impacts the brick selection phase. Accordingly, we will work on new methods that prioritize the generation of relevant bricks and on the early-pruning of uninteresting bricks.

We also plan to include incremental SAT solving, as proposed by Lee et al. [11], in the brick generation process. Incremental SAT solving should significantly reduce current runtime as many computations will be shared across different calls to the SAT solver.

Finally, to improve the delay of the generated circuits, we will include an additional metric to steer brick selection similar to the information coverage present in the original implementation of Iterative Layering. This metric should prioritize the early selection of bricks that contain most of the information of the input circuit.

References

[1] A. K. Verma, P. Brisk, and P. Ienne, “Iterative Layering: Optimizing arithmetic circuits by structuring the information flow,” in *ICCAD*. IEEE, 2009, pp. 797–804.

[2] J.-H. R. Jiang, C.-C. Lee, A. Mishchenko, and C.-Y. Huang, “To SAT or not to SAT: Scalable exploration of functional dependency,” *IEEE Trans. Computers*, vol. 59, no. 4, pp. 457–467, 2010.

[3] J.-H. R. Jiang and R. K. Brayton, “Functional dependency for verification reduction,” in *CAV*, ser. Lecture Notes in Computer Science, R. Alur and D. Peled, Eds., vol. 3114. Springer, 2004, pp. 268–280.

[4] K. L. McMillan, “Methods for exploiting SAT solvers in unbounded model checking,” in *MEMOCODE*. IEEE Computer Society, 2003, pp. 135–142.

[5] C. P. Gomes, H. Kautz, A. Sabharwal, and B. Selman, “Satisfiability solvers,” in *Handbook of Knowledge Representation*, ser. Foundations of Artificial Intelligence, F. V. Harmelen, V. Lifschitz, and B. Porter, Eds. Elsevier, 2008, vol. 3, ch. 2, pp. 89–134.

[6] W. Craig, “Linear reasoning. A new form of the Herbrand-Gentzen theorem,” *J. Symb. Log.*, vol. 22, no. 3, pp. 250–268, 1957.

[7] N. Eén and N. Sörensson, “An extensible SAT-solver,” in *SAT*, ser. Lecture Notes in Computer Science, E. Giunchiglia and A. Tacchella, Eds., vol. 2919. Springer, 2003, pp. 502–518.

[8] K. L. McMillan, “Interpolation and SAT-based model checking,” in *CAV*, ser. Lecture Notes in Computer Science, W. A. H. Jr. and F. Somenzi, Eds., vol. 2725. Springer, 2003, pp. 1–13.

[9] C.-H. Lin, C.-Y. Wang, and Y.-C. Chen, “Dependent-latch identification in reachable state space,” *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 28, no. 8, pp. 1113–1126, 2009.

[10] “Berkeley logic synthesis and verification group,” *ABC: A System for Sequential Synthesis and Verification*, http://www.eecs.berkeley.edu/alanmi/abc/, Release 20130. January 2012.

[11] C.-C. Lee, J.-H. R. Jiang, C.-Y. Huang, and A. Mishchenko, “Scalable exploration of functional dependency by interpolation and incremental SAT solving,” in *ICCAD*, G. G. E. Gielen, Ed. IEEE, 2007, pp. 227–233.