Hardware-Accelerated Formal Verification

Hiroaki Yoshida[1,2] Satoshi Morishita[3] Masahiro Fujita[1,2]

1. VLSI Design and Education Center (VDEC), University of Tokyo
2. CREST, Japan Science and Technology Agency
3. Dept. of Electronic Engineering, University of Tokyo

Abstract

A semi-formal verification technique, which performs a brute-force compiled simulation with a sophisticated search space pruning, has been proposed and shown to be competitive with the state-of-the-art SAT-based verification techniques [3]. This paper presents a novel approach for accelerating the semi-formal verification by utilizing hardware/software co-execution. To maximize the gain from hardware acceleration, we propose two novel techniques such as hardwired conflict analysis for learning and speculative input pattern generation. Experimental results demonstrate that our FPGA-based prototype system achieved about 7x speedup compared against the software implementation of the semi-formal verifier. Also, to reduce the compilation time overhead, we propose a novel reconfigurable architecture in which every logic block has an embedded conflict analysis logic.

1 Introduction

With the increasing importance and substantial cost, efficient formal verification of a large-scale design has become a major challenge. One of the most promising approaches for overcoming this challenge is the use of hardware accelerator. A number of studies on hardware acceleration of SAT solver, a core engine of formal verification, have been available [6, 1]. However, most hardware-accelerated SAT solvers have not integrated advanced techniques used in the state-of-the-art SAT solvers, such as the conflict learning and non-chronological backtracking. Recently a semi-formal verification technique, which performs a brute-force compiled simulation with a sophisticated search space pruning, has been proposed and shown to be competitive with the state-of-the-art SAT-based verification techniques.

This paper presents a novel approach for accelerating the semi-formal verification by utilizing hardware/software co-execution. To maximize the gain from hardware acceleration, we propose several novel techniques such as hardwired conflict analysis and speculative input pattern generation. Experimental results demonstrate that our FPGA-based prototype system achieved about 7x speedup compared against the software-only implementation of the semi-formal verifier.

In our approach, like other hardware/software co-execution approaches, the hardware is customized for each input problem instance and hence the compilation time is considerable for a large-scale design. Therefore, not only the execution time but also the compilation time should be reduced. To reduce the compilation time overhead, we propose a novel reconfigurable architecture in which every logic block has an embedded conflict analysis logic.

2 Semi-Formal Verification Technique

Bingham and Hu [3] proposed a semi-formal bounded model checking method[1] based on compiled simulation, as a replacement of SAT-based model checkers. The main disadvantage of a simulation-based approach is the exponential number of possible input vectors. The semi-formal method overcomes this disadvantage by pruning the search space analogously to the advanced techniques used in the state-of-the-art SAT solvers, such as the conflict learning and non-chronological backtracking. An important advantage of this method is that, even if the full verification is infeasible, one can obtain the verification coverage, *i.e.* the fraction of the search space which has been verified.

In the remainder of this section, we will explain the details of the semi-formal verification method. Procedure 1 describes the procedure of the semi-formal verification. In the procedure, V maintains the set of input vectors which have been verified. First, a new input vector which has not been verified is picked up as a random minterm from $\overline{V}$ and a compiled simulation for the input vector is performed by calling a function `simulate_circuit`. The function body in `simulate_circuit` is generated according to the target circuit structure. Each statement in the generated code corresponds to the evaluation of a gate in the circuit. For in-

[1] We omit the details of bounded model checking here since it is irrelevant to the main argument. For details please refer to [2]

Procedure 1 Semi-Formal Verification

```
1: V ← ∅
2: repeat
3:    v ← A random minterm in V̄
4:    (f, c_skip) ← simulate_circuit(v)
5:    if f = TRUE then
6:       return SATISFIABLE
7:    end if
8:    V ← V ∪ {c_skip}
9: until V = 1
10: return UNSATISFIABLE
```

stance, if a circuit consists of a 2-input AND gate, and then the function body should be as follows:

```
circuit.c.value =
    circuit.a.value & circuit.b.value;
```

Since the generated simulator does not require any expensive computation such as graph traversal, an efficient simulation can be achieved.

In addition to the compiled simulation, the function `simulate_circuit` computes the skip cube of the primary output (verification target). A skip cube is a subset of the input values which determines the value of the output independent of the other input values. In other words, any input not contained in the skip cube can take any value without changing the output, *i.e.* can be treated as *don't care*. The skip cube $A_v(w)$ of a wire w with respect to an input vector v is defined as follows:

Definition 1 *The skip cube $A_v(w)$ of a wire w with respect to an input vector v is defined depending on the type:*

1. *If w is the output of a two-input gate with controlling value μ (e.g. $\mu = 0$ for AND gates) and input wires a and b,*

$$A_v(w) = \begin{cases} A_v(a) \cap A_v(b) & \text{if } a_v = \overline{\mu} \wedge b_v = \overline{\mu} \\ A_v(b) & \text{if } a_v = \overline{\mu} \wedge b_v = \mu \\ A_v(a) & \text{if } a_v = \mu \wedge b_v = \overline{\mu} \\ \max(A_v(a), A_v(b)) & \text{if } a_v = \mu \wedge b_v = \mu \end{cases}$$

   where max() *returns the set of greater cardinality.*

2. *If w is the output of an inverter with input a, $A_v(w) = A_v(a)$.*

3. *If w is an input x_i, $A_v(w) = (B_1, \ldots, B_N)$ where $B_j = w_v$ if $j = i$, or $B_j =$ " $-$ " (don't care) otherwise.*

4. *If w is a two-input gate without a controlling value (i.e. exclusive-OR or exclusive-NOR) with input wires a and b, $A_v(w) = A_v(a) \cap A_v(b)$.*

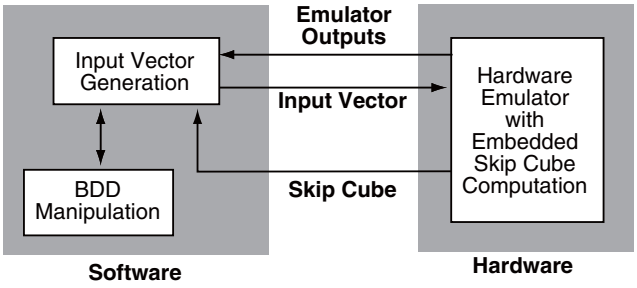


Figure 1. System organization.

The skip cube of a gate with more than 2 inputs can be defined similarly. In the procedure, the skip cube $A_v(f)$ of the primary output f is computed for each input vector v in the function `simulate_circuit` as well as the simulation of the target circuit. Since any input vector contained in the skip cube leads to the same output, we can skip all input vectors in the skip cube from the next iterations by adding the skip cube to V. Thus, the number of input vectors which are actually simulated is significantly small.

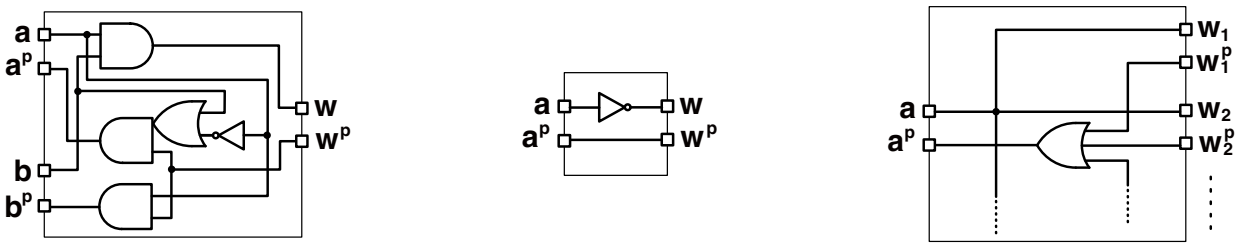
To represent the set of input vectors V which have been verified, *i.e.* the input vectors which have been explicitly simulated and covered via skip cubes, a BDD [5] is maintained. For each iteration, a new input vector is generated by picking up a random minterm not contained in the BDD. The main advantage of using a BDD is that picking up a minterm not in a BDD can be computed in linear time to the BDD size.

As mentioned earlier, this semi-formal approach can be viewed as an analogy to the advanced techniques used in the state-of-the-art SAT solvers. A skip cube can be viewed as a conflict clause, and BDD-based pattern generation as the non-chronological backtracking. The experimental results showed that, in some cases, the semi-formal approach performs competitive with the state-of-the-art model checker based on SAT solvers.

3 Hardware/Software Co-Execution System for Formal Verification

3.1 System Overview

In this paper, we assume that a given verification problem can be transformed into a problem of checking the satisfiability of a circuit. For instance, suppose a given problem is the equivalence checking between two circuits. First a new circuit is generated by adding a gate which computes the exclusive-OR of the primary outputs of two circuits, then the equivalence is proved by checking that the output of the exclusive-OR gate is false for all possible input vectors. Similarly, a property checking problem can be transformed



(a) 2-input AND block (AND) (b) Inverter block (INV) (c) Distribution block (DIST)

Figure 2. Basic logic gate blocks for embedded skip cube computation.

by adding an extra circuit which checks the property where the output of the extra circuit is true *iff* the property is true. There are several methods proposed for generating such an extra circuit [4]. Thus, under the assumption that a target circuit has one and only primary output f which is the verification target, our verification system focuses only on the problem of checking if f is true for all input vectors.

Initially, we implemented a software semi-formal verifier described in Procedure 1 and performed a runtime profiling analysis using an example circuit used in Section 5. The profiling result, as presented in Table 1, shows that the circuit simulation and the skip cube computation spent most of the execution time. Based on the analysis, our system performs Procedure 1 in a hardware/software co-execution fashion, as illustrated in Figure 1. The circuit simulation and skip cube computation in the procedure are executed using hardware. Table 1 presents a runtime profiling result We first describe how to efficiently implement the skip cube computation. Next, to resolve the communication bottleneck, we propose a speculative input pattern generation technique.

3.2 Hardware Emulation with Embedded Skip Cube Computation

In our verification system, we accelerate Procedure 1 by executing the circuit simulation and the skip cube computation (*i.e.* `simulate_circuit`) using hardware. Obviously, the circuit simulation can be easily implemented in hardware. If the skip cube computation is implemented using hardware just the way described in Def. 1, it may require N-bit Boolean operations for each gate in the circuit where N is the number of primary inputs. The size of skip cube computation circuit increases $O(G \cdot N)$ where G is the number of gates, and critically limits the capacity of a target design. To avoid this large overhead, we propose a novel circuit where every gate embeds a fixed skip cube computation logic. The size of our circuit is proportional to the original circuit size regardless of the number of inputs.

First, we slightly modify the definition of the skip cube for efficient hardware implementation as follows:

Table 1. Runtime profiling result of software semi-formal verifier.

Process	Percentage of Execution Time
Circuit simulation	49.2%
Skip cube computation	36.0%
Update of input vector set (BDD)	13.1%
Input vector generation (BDD)	1.7%

Definition 2 *The skip cube $A_v(w)$ of a wire w with respect to an input vector v is defined depending on the type:*

1. *If w is the output of a two-input gate with controlling value μ (e.g. $\mu = 0$ for AND gates) and input wires a and b,*

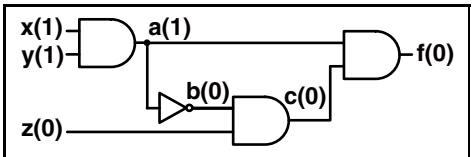
$$A_v(w) = \begin{cases} A_v(a) \cap A_v(b) & \text{if } a_v = \overline{\mu} \wedge b_v = \overline{\mu} \\ A_v(b) & \text{if } a_v = \overline{\mu} \wedge b_v = \mu \\ A_v(a) & \text{if } a_v = \mu \wedge b_v = \overline{\mu} \\ A_v(a) & \text{if } a_v = \mu \wedge b_v = \mu \end{cases}$$

2. 3. 4. Equivalent to Definition 1.

The difference is the skip cube of a two-input gate if both inputs have the controlling value. In such a case, either one of two skip cubes $A_v(a)$ and $A_v(b)$ can be chosen arbitrary. In the original definition, max() function is used in order to pick up (potentially) more efficient skip cube. Since it is not trivial to implement max() using hardware, the skip cube $A_v(a)$ of the first input is always chosen in such a case in our definition. From our experiences, this slight modification of the skip cube computation has little impact on the overall performance. In the remainder of this paper, a skip cube is computed as defined in Definition 2.

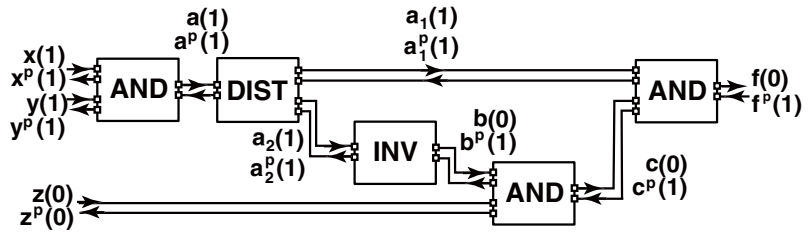
Next, we define a propagation flag:

Definition 3 *The propagation flag (p-flag) $w_v^p \in \{0, 1\}$ of a wire w with respect to an input vector v is 1 if the skip cube $A_v(w)$ of the wire w is propagated to the primary output, and 0 otherwise.*



Wire	Value	Skip Cube x	y	z
x	1	1	-	-
y	1	-	1	-
z	0	-	-	0
a	1	1	1	-
b	0	1	1	-
c	0	1	1	-
f	0	1	1	-

(a) Conventional skip cube computation. The value in the parentheses is the value of a signal. The skip cube of the output f is $(1, 1, -)$.



(b) Embedded skip cube computation. The value in the parentheses is the value of a signal or a p-flag. First the p-flag f^p is set to 1 and all other p-flags are computed from the output. The p-flags of the primary inputs are $(x^p, y^p, z^p) = (1, 1, 0)$ and hence the skip cube is $(1, 1, -)$.

Figure 3. Examples of two skip cube computation methods. In this example, an input vector $(x, y, z) = (1, 1, 0)$ is simulated and the computed skip cube $A_v(f)$ is $(1, 1, -)$. Thus, an input vector $(1, 1, 1)$ can be skipped from the simulation.

Table 2. Computation of p-flag for 2-input gate where a and b are the inputs, w is the output, μ_a and μ_b are the controlling value of the inputs.

a	b	a^p	b^p
μ_a	$-$	w_v^p	0
$\overline{\mu_a}$	μ_b	0	w_v^p
$\overline{\mu_a}$	$\overline{\mu_b}$	w_v^p	w_v^p

In the proposed circuit, we replace each logic gate with a logic block which not only performs the original logic operation but also computes the p-flags. Each wire in the original circuit is replaced with two wires, one is for the signal and the other is for the p-flag. The direction of the p-flag computation is opposite to the logic operation, that is, the p-flags of the gate inputs are computed from the p-flag of the gate output. After setting the p-flag of the primary output to 1, the p-flag of each wire is computed in a topological order from the primary output to the primary inputs. Table 2 shows how the p-flags of the inputs of a two-input gate are computed, where a, b, w are the inputs and the output of the two-input gate. Figure 2 (a)-(c) show basic logic gate blocks with embedded p-flag computation. *For ease of explanation*, it is assumed that a given circuit consists only of 2-input AND gates and inverters. An extension to more general circuits should be straightforward and also an extension to FPGA-like devices is explained in the next section. A circuit for hardware emulation with embedded skip cube computation can be obtained by simply replacing each 2-input AND gate with the 2-input AND block in Figure 2 (a), each inverter with the inverter block in Figure 2 (b) and each multiple fanout point with the distribution block in Figure 2 (c). Figure 3 illustrates the example of the skip cube computation by the conventional method (Figure 3 (a)) and the proposed hardwired method (Figure 3 (b)). The circuit in Figure 3 (a) is transformed into the circuit in Figure 3 (b) by applying the rules described above.

After computing the p-flags of all wires, we can compute the skip cube $A_v(f)$ of the primary output f by the following theorem:

Theorem 1 *Let C be a cube $(B_1, \ldots, B_N)$ such that*

$$B_i = \begin{cases} v_i & \text{if } P_i = 1 \\ \text{"}-\text{"} & \text{if } P_i = 0 \end{cases}$$

where v_i is the value of the i-th input and P_i is the p-flag of i-th primary input. Then, C is equivalent to the skip cube $A_v(f)$ of the primary output f.

Proof *We omit the proof due to the limit of space.*

Thus, we can easily compute the skip cube from the p-flags of the primary inputs.

The size of the emulation hardware with the embedded skip cube computation logic is approximately several times bigger than that of the original circuit. The worst delay for computing the skip cube, *i.e.* the p-flags of the primary inputs, is also approximately several times slower than that

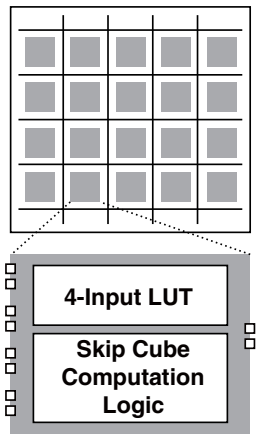


Figure 5. Reconfigurable architecture for accelerating formal verification.

of the original circuit. This is because the p-flags are computed from the primary output to the primary inputs after the values of all logic gates are determined. Using latest FPGAs with a capacity of several million gates, our system can emulate up to one million gate design.

3.3 Speculative Input Vector Generation

In a typical hardware/software co-execution system, the communication between hardware and software could be a major performance bottleneck. Every step in Procedure 1 must be executed sequentially because an input vector for an iteration depends upon a skip cube of the previous iteration and *vice versa*. Therefore, every iteration in the procedure requires at least one distinct hardware/software communication. Since our system uses a generic communication method like UDP/IP via Ethernet, there is a large overhead in every communication. To resolve such a communication overhead, the number of communications is reduced by applying a slight modification to the original procedure. Instead of generating an input vector for each iteration, a certain number of input vectors are randomly picked up from $\overline{V}$. Then, the set of input vectors are transmitted to the hardware at once. The simulation is performed for all input vectors and the set of the outputs and skip cubes are returned. The set of the skip cubes are added into V.

A potential drawback of this speculative vector generation is that many input vectors could be redundant because they are contained in the skip cubes computed for the other input vectors. By using an example, we will show the effectiveness of the speculative generation. As an example, the equivalence of the lower 8 bits of the outputs of two 16×16bit multipliers is checked. Figure 4 (a) presents the number N_C of total communications when a certain number N_S of input vectors are generated speculatively for each

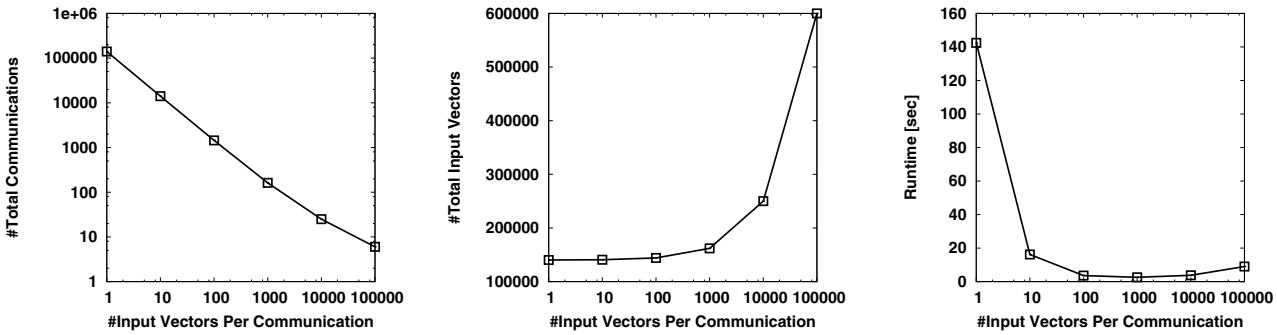
Table 3. Computation of p-flag for 4-input gate.

a	b	c	d	a^p	b^p	c^p	d^p
μ_a	$-$	$-$	$-$	w_v^p	0	0	0
$\overline{\mu_a}$	μ_b	$-$	$-$	0	w_v^p	0	0
$\overline{\mu_a}$	$\overline{\mu_b}$	μ_c	$-$	0	0	w_v^p	0
$\overline{\mu_a}$	$\overline{\mu_b}$	$\overline{\mu_c}$	μ_d	0	0	0	w_v^p
$\overline{\mu_a}$	$\overline{\mu_b}$	$\overline{\mu_c}$	$\overline{\mu_d}$	w_v^p	w_v^p	w_v^p	w_v^p

communication. This graph clearly shows that the number of communications can be reduced dramatically by using the speculative generation. Figure 4 (b) presents the number of total input vectors N_V generated for the full verification, *i.e.* $N_C \times N_S$. As can be seen easily, the speculative generation is effective up to $N_S = 1000$. Figure 4 (c) presents the estimated runtime under the practical assumption that the computation for each iteration requires $25\mu s$ and the single communication requires $1ms$. In this example, we found that $N_S = 1000$ leads to the best. This example demonstrates that the speculative generation can dramatically reduce the communication overhead.

4 Reconfigurable Architecture for Accelerating Formal Verification

Like many other hardware-acceleration approaches, the hardware is customized for each input problem instance and hence the compilation time is considerable for a large-scale design. In this section, we propose an FPGA-like reconfigurable architecture for accelerating formal verification. A basic idea is to add the skip cube computation logic described in Section 3.2 to each logic block in FPGA, as illustrated in Figure 5. Table 3 shows the function of the skip cube computation logic for generic 4-input gate. The skip cube computation logic contains 8-bit memory where each input corresponds to 2-bit for maintaining the controlling value (0, 1, −). Note that a logic block is not necessarily a 4-input LUT, but it can be 5-input LUT or any other complex logic. An interconnect between logic blocks is replaced with a dual-rail interconnect where one of the rails is for a signal and the other is for a p-flag. The same design methodology as FPGAs can be used with a little modification. Using the conventional method, a circuit is synthesized and mapped into a netlist of 4-input LUTs. For each LUT, the controlling value for each LUT input is analyzed and programmed into the skip cube computation logic. Since the skip cube computation logic is comparably small compared with 4-input LUT, the increase in logic area



(a) Communication reduction by speculative generation

(b) Number of total input vectors

(c) Runtime estimation

Figure 4. Effectiveness of speculative input vector generation.

Table 4. Speedup by hardware acceleration.

Name	Software Only [sec.]	HW/SW Co-Execution [sec.]	Speedup Factor
MULT16-8	5.7	1.0	5.7
MULT16-9	25.9	3.7	7.0
MULT16-10	122.5	16.5	7.4
MULT16-11	575.3	88.0	6.5
Average			6.7

should be small. Due to the doubled interconnect, the area efficiency may be reduced by up to half.

5 Experimental Results

We implemented the proposed hardware/software co-execution verification system described in Section 3. The software portion is implemented on Linux PC system on Core 2 Duo 2.0GHz with 2 GB main memory, and the hardware portion is implemented using a Xilinx Virtex-II Pro FPGA with embedded PowerPC processor. For the communication between the PC and the FPGA, the UDP/IP via Ethernet is used and the embedded processor in the FPGA controls the UDP/IP communication.

We used a 16×16bit multiplier as an example circuit, and our system is used for checking the equivalence of the lower n-bits of the outputs. We also implemented the semi-formal model checker explained in Section 2 and performed the same equivalence checking as a reference. Table 4 compares the runtime of software-only execution and hardware/software co-execution, and also shows the speedup factor. The results show that our system was about 7x faster than the state-of-the-art semi-formal model checker.

6 Conclusions

In this paper, we proposed a novel hardware/software co-execution approach for accelerating the semi-formal verification. To maximize the gain from hardware acceleration, we propose several novel techniques such as hardwired conflict analysis and speculative input pattern generation. Despite the recent progress in formal verification, conventional simulation is still widely used for functional verification. From another perspective, our approach can be regarded as a technique for enhancing the coverage of simulation-based verification. Experimental results demonstrated that our FPGA-based prototype system achieved about 7x speedup compared against the software implementation of the semi-formal verifier. Also, to reduce the compilation time overhead, we also proposed a novel reconfigurable architecture which embeds the conflict analysis capability.

References

[1] M. Abramovici, J. T. de Sousa, and D. Saab. A massively-parallel easily-scalable satisfiability solver using reconfigurable hardware. In *Proceedings of Design Automation Conference*, pages 684–690, 1999.

[2] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. *Lecture Notes in Computer Science*, 1579:193–207, 1999.

[3] J. D. Bingham and A. J. Hu. Semi-formal bounded model checking. In *Proceedings of International Conference on Computer Aided Verification*, pages 280–294, 2002.

[4] M. Boule and Z. Zilic. Incorporating efficient assertion checkers into hardware emulation. In *Proceedings of IEEE International Conference on Computer Design*, pages 221–228, 2005.

[5] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.

[6] P. Zhong, M. Martonosi, P. Ashar, and S. Malik. Solving boolean satisfiability with dynamic hardware configurations. *Lecture Notes In Computer Science*, 1482:326–335, 1998.