Proc. of Fifth International Conference on Principles and Practice of Constraint Programming 1999 (CP-99)

Solving Satisfiability Problems on FPGAs using Experimental Unit Propagation

Takayuki Suyama, Makoto Yokoo, and Akira Nagoya

NTT Communication Science Laboratories
2-4 Hikaridai, Seika-cho
Soraku-gun, Kyoto 619-0237 Japan
e-mail: suyama/yokoo/nagoya@cslab.kecl.ntt.co.jp

Abstract. This paper presents new results on an innovative approach for solving satisfiability problems (SAT), that is, creating a logic circuit that is specialized to solve each problem instance on Field Programmable Gate Arrays (FPGAs). This approach has become feasible due to recent advances in Reconfigurable Computing, and has opened up an exciting new research field in algorithm design.
We have developed an algorithm that is suitable for a logic circuit implementation. This algorithm is basically equivalent to the Davis-Putnam procedure with Experimental Unit Propagation. The algorithm requires fewer hardware resources than previous approaches. Simulation results show that this method can solve a hard random 3-SAT problem with 400 variables within 1.6 minutes at a clock rate of 10MHz. Faster speeds can be obtained by increasing the clock rate. Furthermore, we have actually implemented a 128-variable, 256-clause problem instance on FPGAs.

1 Introduction

Recently, due to advances in Field Programmable Gate Array (FPGA) technologies [3], users can create original logic circuits and electronically reconfigure them. Furthermore, users are able to describe their designs in a Hardware Description Language (HDL) and obtain logic circuits by using current high level logic synthesis technologies [4, 12]. These recent hardware technologies enable users to rapidly create logic circuits specialized to solve each problem instance. We have chosen satisfiability problems to examine of the effectiveness of this approach.

A constraint satisfaction problem (CSP) is a general framework that can formalize various problems, and many theoretical and experimental studies have been performed on these problems [10]. In particular, a satisfiability problem for propositional formulas in conjunctive normal form (SAT) is an important subclass of CSP. This problem was the first computational task shown to be NP-hard.

In [17], we presented an initial report on innovative approach for solving satisfiability problems. In this approach, a logic circuit that is specialized to solve each problem instance is created on FPGAs. After this pioneer work, various researches following this line have been carried out [1, 8, 13, 16, 18]. As such, solving SAT using FPGAs is becoming a very vital research area.

The performance of the method described in [17] is equivalent to the basic Davis-Putnam procedure [5], and the performance is not very efficient since the variable ordering is static. In [16], we introduced dynamic variable ordering with Maximum Occurrences in clauses of Minimum Size (Mom's) heuristic [6, 7]. However, we were not able to implement large-scale problems since implementing Mom's heuristic on FPGAs require too many hardware resources. In this paper, we present a new method for implementing a more efficient algorithm that requires less hardware resources. This algorithm is basically equivalent to the Davis-Putnam procedure with Experimental Unit Propagation [9].

In the remainder of the paper, we describe Reconfigurable Computing approach (Section 2), and briefly describe the problem definition (Section 3). Then, we present in detail the developed algorithm, which is suitable for implementation on a logic circuit (Section 4). We show an implementation of this algorithm on FPGAs (Section 5). Finally, we show evaluation results obtained with the software simulation and actual implementation (Section 6).

2 Reconfigurable Computing Approach

In this section, we describe our Reconfigurable Computing (RC) approach. RC systems are hardware systems with logical configurations that can be changed to solve some problem or to quickly carry out some application. These systems are realized with FPGAs and logic synthesis systems. Ordinary RC systems are reconfigured to a target application. On the other hand, our RC system features reconfiguration for an instance of the application problem.

One might argue that it is natural to increase a system's speed by implementing an algorithm on hardware, and there is no significant reason for research on such an approach. This argument is not correct, since the operations that can be directly performed by hardware are rather limited. If we perform a complicated operation by iterating a number of simple operations, the performance is similar to that of general-purpose computers.

To obtain reasonable speed increases utilizing hardware, we need new and quite different methodologies for designing/implementing algorithms. For example, when implementing an efficient search algorithm using general-purpose computers, we often need to avoid duplicated computations by using a carefully designed data structure to represent a state. More specifically, we can maintain an integer vector for clauses (the initial value of each vector element is 3), when solving a 3-SAT problem. In determining a variable value, we subtract one from the element where the clause is reduced by this operation. With such a data structure, radically changing a state is not desirable, because its book-keeping would be too costly.

On the other hand, if we have enough hardware resources, all clauses can be checked in one clock cycle without using such a data structure. Since some kinds of computations can be performed very quickly using hardware, we can get more freedom in the design of algorithms. We believe that this approach brings a very exciting new dimension to algorithm design.

A logic circuit that solves a specific SAT problem is synthesized by the following procedure. First, a text file that describes a SAT problem is analyzed by an SFL generator written in the C language. This program generates a behavioral description specific to the given problem with an HDL called SFL. Then, a CAD system analyzes the description and synthesizes a netlist, which describes the logic circuit structure. Finally, the FPGA Mapper of the FPGA system generates FPGA mapping data from the netlist. We use a system called PARTHENON [12] developed at NTT. PARTHENON is a highly practical system that integrates a description language SFL, simulator, and logic synthesizer. Furthermore, the FPGA Mapper generates FPGA mapping data from the netlist. Only the SFL generator is newly developed for this research. Other parts are commercial software.

3 Problem Definition

A satisfiability problem for propositional formulas in conjunctive normal form (SAT) can be defined as follows. A boolean *variable* x_i is a variable that takes a true or false value (represented as 1 or 0, respectively). We call the value assignment of one variable a *literal.* A *clause* is a disjunction of literals, e.g., $(x_1 + \overline{x_2} + x_3)$. Given a set of clauses $C_1, C_2, \ldots, C_m$ and variables $x_1, x_2, \ldots, x_n$, the satisfiability problem is to determine if the formula $C_1 \cdot C_2 \cdot \ldots \cdot C_m$ is satisfiable, that is, to determine whether an assignment of values to the variables exists so that the above formula is true.

In this paper, if the formula is satisfiable, we assume that we need to find all or a fixed number of solutions, i.e., the combinations of variable values that satisfy the formula. Most of the existing algorithms for solving SAT problems aim to find only one solution. Although this setting corresponds to the original problem definition, some application problems, such as visual interpretation tasks [15] and diagnosis tasks [14] require finding all or multiple solutions. Furthermore, since finding all or multiple solutions is usually much more difficult than finding only one solution, it is worthwhile to solve the problem with special-purpose hardware.

For simplicity, we restrict our attention to 3-SAT problems, i.e., the number of literals in each clause is 3. Relaxing this assumption is rather straightforward.

4 Algorithm

4.1 Basic Algorithm

The algorithm used in this paper is basically equivalent to the Davis-Putnam procedure that introduces a dynamic variable ordering with *Experimental Unit Propagation* (EUP) [9]. The outline of the algorithm can be described as follows.

Let I be the input SAT problem instance, and L be the working list containing problem instances, where L is initialized as $\{I\}$. A problem instance is represented as a pair of a set of value assignments and a set of clauses that are not yet satisfied.

1. If L is empty, then I is unsatisfiable, so stop the algorithm. Otherwise, select the first element S from L and remove S from L.
2. If S contains an empty clause, then S is unsatisfiable, go to 1.
3. If all clauses in S are satisfied, print the value assignments of S as one solution, go to 1.
4. If S contains a unit clause (a clause with only one variable), then set this variable to the value which satisfies the clause, simplify the clauses of S and go to 2.
5. **Branching:** To select a variable to be assigned at the next step, we experimentally set each unassigned variable's value to 0 and 1. This procedure is called Experimental Unit Propagation. If some clause becomes unsatisfiable by setting a variable to 0, set it to 1, and vice versa, then go to 2. Otherwise, we select the variable x that causes the maximum number of unit propagation[1]. Add to the top of L the simplified instance S' obtained by setting x to 1. Set x to 0 and simplify S. Go to 2.

4.2 Differences between the Previous Approach

In [16], we developed a method for implementing a backtracking tree search algorithm without using a stack. In this approach, the overhead of sequential accesses to a large memory is avoided by using separate registers assigned for each variable. More specifically, there exists a register for each variable, which records the depth of the search tree where the variable value is determined. This information is used for backtracking.

We follow our previous method to implement backtracking tree search procedures, but use a different branching heuristic. In [16], Maximum Occurrences in the clauses of the Minimum Size (Mom's) heuristic [6, 7] is used for branching. However, we need a logic circuit that counts the number of occurrences of all variables in binary clauses to implement MOM's heuristic. This logic circuit tends to be very large, so it is very difficult to implement large-scale problems using currently available hardware resources. Actually, only a very small-scale problem with 30 variables was implemented in [16].

In this paper, we introduce another branching heuristic called Experimental Unit Propagation, which was shown to be at least as efficient as MOM's heuristic [9]. One advantage of using this heuristic is that the logic circuit for implementing this heuristic is very similar to the logic circuit for the main tree search procedures; thus, we can share the hardware resources between these routines.

[1] The detail of the branching rule is described later.

4.3 Details of the Algorithm

In this section, we are going to describe the detail of the algorithm. First, we are going to define concepts and terms used in the algorithm. We represent the fact that x_i is true as $(x_i, 1)$.

– Each variable x_i is associated with the value $depth(x_i)$, which represents the depth of the search tree where the variable value is determined.
– Each variable x_i is associated with the value $determined(x_i)$.
  If $determined(x_i) = 1$, this means that x_i's value is determined.
  If $determined(x_i) = 0$, x_i's value is not yet determined. The initial value of $determined(x_i)$ is 0.
– Each variable x_i is associated with the value $branch(x_i)$. If $branch(x_i) = 0$, this means that x_i's value is determined by unit resolutions. If $branch(x_i) = 1$, this means that x_i's value is determined by branching.
– A global variable *current_depth* is defined with an initial value of 1.
– For each clause $(x_i, v_i) \vee (x_j, v_j) \vee (x_k, v_k)$, we classify the condition of the clause into one of the following five cases.
  not-satisfied: for each variable x_i, x_j, x_k, the variable value is determined, and the value is not equal to v_i, v_j, v_k, respectively.
  satisfied: at least one of the following conditions is satisfied; 1) $determined(x_i) = 1$ and x_i's value is v_i, 2) $determined(x_j) = 1$ and x_j's value is v_j, 3) $determined(x_k) = 1$ and x_k's value is v_k.
  unit: for any two of x_i, x_j, x_k, the variable value is determined, and the value of the remaining one variable is not determined. The value is not equal to v_i, v_j, v_k, respectively, for the variable whose value is determined.
  other: the other condition above.
– If multiple unit clauses exist, and these unit clauses assign different values to the same variable, we call these unit clauses *inconsistent*.

Here, we show the details of the algorithm. The algorithm consists of "Main Procedure" and "Experimental Unit Propagation Procedure". The Experimental Unit Propagation Procedure is used in the **Branching** state for searching a variable that should be assigned at the next step. In the initial state, *current_depth* is 1, and for each variable x_i, $determined(x_i)$ and $branch(x_i)$ are 0.

Main Procedure

1. **Not Satisfied:** If a *not-satisfied* clause or inconsistent *unit* clauses exists, go to 5.
2. **Satisfied:** If all clauses are *satisfied*, print the current value assignments as a solution. Go to 5.
3. **Unit:** If *unit* clauses exist and are not inconsistent, for all unit clauses, for each variable x_i where $determinead(x_i)$ is 0, set the value to v_i, which is specified by the clause. Set $determined(x_i)$ to 1, $branch(x_i)$ to 0, and $depth(x_i)$ to *current_depth*. Go to 1.

4. **Branching:** Otherwise, set max_eval_sum to 0, and max_eval_min to 0, and apply the conditions below for each variable x_i where $determined(x_i)$ is 0

   (a) Set *eval_0* to the return value of Experimental Unit Propagation(x_i, 0) procedure. If *eval_0* is *not-satisfied*, set $determined(x_i)$ to 1, x_i to 1, $branch(x_i)$ to 0 and $depth(x_i)$ to *current_depth*. Abort procedures for the other variables. Go to 1.

   (b) Set *eval_1* to the return value of Experimental Unit Propagation(x_i, 1) procedure. If *eval_0* is *not-satisfied*, set $determined(x_i)$ to 1, x_i to 0, $branch(x_i)$ to 0 and $depth(x_i)$ to *current_depth*. Abort procedures for the other variables. Go to 1.

   (c) If $max\_eval\_min < min(eval\_0, eval\_1)$, or $max\_eval\_min = min(eval\_0, eval\_1)$ and $max\_eval\_sum < eval\_0 + eval\_1$, set *best_pos* to x_i, max_eval_min to $min(eval\_0, eval\_1)$ and max_eval_sum to $eval\_0 + eval\_1$.

   Branching end: Set x_i that is specified by *best_pos* to 0, $determined(x_i)$ to 1, $branch(x_i)$ to 1 and $depth(x_i)$ to $current\_depth + 1$. Set *current_depth* to $current\_depth + 1$. Go to 1.

5. **Backtracking:**

   5.1 If *current_depth* is 1, stop the algorithm.

   5.2 Otherwise, for each variable x_i,
   If $depth(x_i) = current\_depth$ and $branch(x_i) = 0$, set $determined(x_i)$ to 0, and set $depth(x_i)$ to 0.
   If $depth(x_i) = current\_depth$ and $branch(x_i) = 1$, set x_i to 1, $branch(x_i)$ to 0, and set $depth(x_i)$ to $current\_depth - 1$.

   5.3 set *current_depth* to $current\_depth - 1$, go to 1.

Experimental Unit Propagation(x_i, *value*) Procedure

Set $count = 0$, $x_i = value$, $branch(x_i) = 1$, $determined(x_i) = 1$, $depth(x_i) = current\_depth + 1$ and $current\_depth = current\_depth + 1$.

1. **Not Satisfied:** If a *not-satisfied* clause or inconsistent *unit* clauses exist, set *result* to *not-satisfied*. Go to 5.
2. **Satisfied:** If all clauses are *satisfied*, set *result* to n (the number of variables). Go to 5.
3. **Unit:** If *unit* clauses exists and are not inconsistent, for all unit clauses, for each variable x_i where $determinead(x_i)$ is 0, set the value to v_i, which is specified by the clause. Set $determined(x_i)$ to 1, $branch(x_i)$ to 0, $depth(x_i)$ to *current_depth*, and *count* to $count + 1$. Go to 1.
4. **Branching:** Otherwise, set *result* to *count*. Go to 5.
5. **Backtracking:** For each variable x_i, if $depth(x_i) = current\_depth$, set $determined(x_i)$ to 0, $branch(x_i)$ to 0 and $depth(x_i)$ to 0. Set $depth(x_i)$ to $current\_depth - 1$. Return *result*.

5 Implementation

The algorithm described in Section 4 can be straightforwardly represented as a finite state machine.

Since there are many similarities between the Main Procedure and the Experimental Unit Propagation Procedure, hardware can be shared between them, that is, at some time the hardware works in the Main Procedure mode, while at other times, it works in the Experimental Unit Propagation Procedure mode. In order to distinguish these two modes, a flag called *eup* is set up. If *eup* is 0, the algorithm is in the Main Procedure, and if *eup* is 1, the algorithm is in the Experimental Unit Propagation Procedure mode. The *evaluation*, *unit* and *backtrack* states are used both in the Main Procedure mode and in the Experimental Unit Propagation Procedure mode. However, the behavior of these states is slightly different according to the value of *eup*.

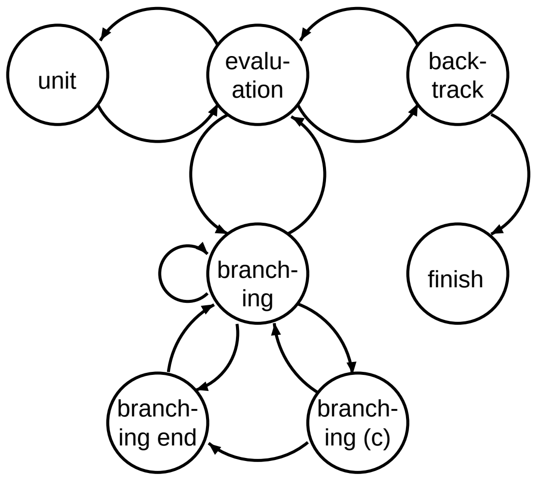

Fig. 1. Finite state machine transitions

The conditions of clauses are calculated and integrated in the *evaluation* state and the next state is determined. The condition of each clause can be calculated in parallel with other clauses. For example, we create a logic circuit that is equivalent to the following logic formula to check whether a clause is *not-satisfied*.

$$(determined(x_i) \cdot determined(x_j) \cdot determined(x_k)) \cdot$$
$$((x_i \oplus v_i) \cdot (x_j \oplus v_j) \cdot (x_k \oplus v_k))$$

where $\oplus$ is Exclusive-OR.

In the *backtracking* state, for each variable x_i, $depth(x_i)$ and *current_depth* are compared, and variable values, *determined*, *depth*, etc. are changed. These procedures can be done in parallel for each variable. If *current_depth* $= 1$, the algorithm is terminated. These procedures require one clock cycle.

In the *unit* state, several variable values are determined by unit clauses. These procedures are simple and can be executed within the same clock cycle of the *evaluation* state.

When the current state moves from the *evaluation* state to the *branching* state, *eup* is set to 1, and *eup* is 1 until it goes back to the Main Procedure mode. In the *branching (c)* state and *branching end* state, **Branching (c)** and **Branching end** in the Main Procedure are carried out, respectively.

The high-level hardware description language SFL can handle the finite state machine representation, and the LSI CAD system PARTHENON can automatically generate a RT-level hardware description. The state transitions of the finite state machine are shown in Figure 1.

6 Evaluation

6.1 Simulation

We first evaluate the efficiency of the developed algorithm with software simulation. We use hard random 3-SAT problems for evaluation. Each clause is generated by randomly selecting three variables, and each of the variables is given the value 0 or 1 (false or true) with a 50% probability. The number of clauses divided by the number of variables is called *clause density*, and the value 4.3 has been identified as a critical value that produces particularly difficult problem instances [11], which are called problems in *phase transition* region.

In Figure 2, we show the log-scale plot of the average required time of over 100 phase-transition problems, assuming the clock rate is 10MHz. Since a randomly generated 3-SAT problem tends to have a very large number of solutions when it is solvable, we terminate each execution after the first 100 solutions are found to finish the simulation within a reasonable amount of time.

Figure 2 shows that the new method (EUP) can solve a hard random 3-SAT problem with 400 variables within 1.6 minutes at a clock rate of 10MHz. In addition, we can see that the search tree growing at the rate of the EUP is $O(2^{n/20.0})$, where n is the number of variables, whereas the search tree of MOM's[16] grows at the rate of $O(2^{n/17.3})$. This result shows that the new method is more efficient with a larger number of variables.

Furthermore, we show the results for AIM benchmark problems[2] with 128 variables and 256 clauses on FPGAs. These problem instances are unsolvable, and are known to be very difficult. Table 1 shows the required number of states and the time to solve these problems when the clock rate is 10.0MHz. For comparison, we also show the cpu time of POSIT [7], a very sophisticated SAT solver that utilizes the MOM heuristic. We should have shown the result obtained by a software implementation that utilizes Experimental Unit Propagation, such as

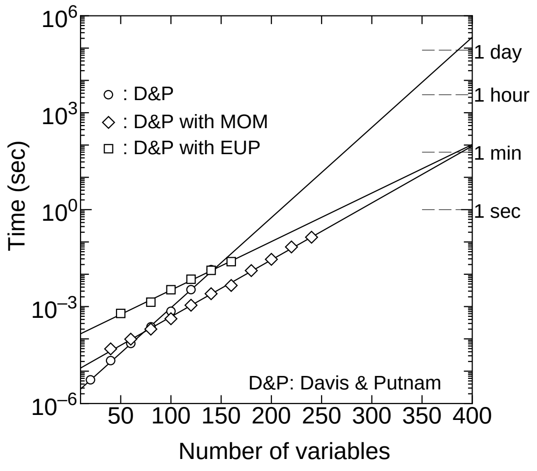

Fig. 2. Required time at 10MHz on hard random 3-SAT problems

Table 1. The required state and time for "aim-128-2_0-no-*.cnf"

problem	# of states	time (sec) @ 10MHz	cpu time of POSIT (sec)	speed-up ratio
aim-128-2_0-no-0	1,973,080,127	197.3	884.1	4.48
aim-128-2_0-no-1	1,662,111,245	166.2	931.6	5.61
aim-128-2_0-no-2	44,321,403	4.43	230.8	52.1
aim-128-2_0-no-3	3,651,710,074	365.2	2138.1	5.85
aim-128-2_0-no-4	2,726,316,935	272.6	2749.3	10.08
aim-128-2_0-no-5	839,166,228	83.9	471.6	5.62
average	1,816,117,668	181.6	1234.2	6.80

Satz [9]. Since we don't have a well-optimized code for this method at hand, we refrain from reporting the performance of this method. These programs run on a Sun Ultra 30 Model 300 (UltraSPARC-II 296MHz). We can see that the running time of EUP on FPGAs is faster than these programs.

Of course, this comparison is not very fair as we do not consider the time required to generate the logic circuit. Currently, generating a logic circuit from a problem description takes an hour. However, these routines can be highly optimized for SAT problems since many parts in a logic circuit are common in all problem instances. Therefore, the time required to generate a logic circuit would be negligible for larger-scale problems if we implement a logic circuit generator that is specialized for SAT problems.

6.2 Current Implementation Status

We use an ALTERA FLEX10K250 FPGA chip to implement the algorithm. FLEX10K250 has 12,160 Logic Cells (LCs) and its typical usable gates are from 149k to 310k. This system is connected to an IBM-PC via parallel port. The total system including mapping software costs around $16,000.

We have actually implemented a 3-SAT problem, "aim-128-2_0-no-*.cnf" as previously described. 11,042 LCs is used for implementing this problem. The utilized rate of LCs is 90%. We were able to run this circuit at a clock rate of 10.0MHz. In addition, we have successfully mapped "aim-200-1_6-yes1-1.cnf" with 200 variables and 320 clauses. The circuit is divided into 21 FLEX10K chips, and the total usability rate of LCs is low (13%). When a logic circuit cannot be fit into one chip, the circuit is divided into multiple chips. However, since the wiring resources among multiple chips (i.e., the number of interface pins of a chip) are very scarce compared with the wiring resources within one chip, we can implement only a small portion of the logic circuit on one chip.

Figure 3 shows the number of gates required for "aim-{50,100,200}-1_6-yes1-1.cnf". The figure shows the number of gates when the circuit is organized by primitive gates. Note that these circuits are the initial ones synthesized from HDL descriptions. If these circuits are optimized, the number of gates can be reduced further with keeping the same trend. EUP's required gates of are approximately 30% less than that of MOM. This is because almost all the circuits of Experimental Unit Propagation can be shared with the Main Procedure. On the other hand, the MOM algorithm requires additional circuits for branching.

7 Conclusions

This paper presented new results on solving SAT using FPGAs. In this approach, a logic circuit specific to each problem instance is created on FPGAs. We developed an algorithm that is suitable for implementation on a logic circuit. This algorithm is basically equivalent to the Davis-Putnam procedure, which introduces Experimental Unit Propagation.

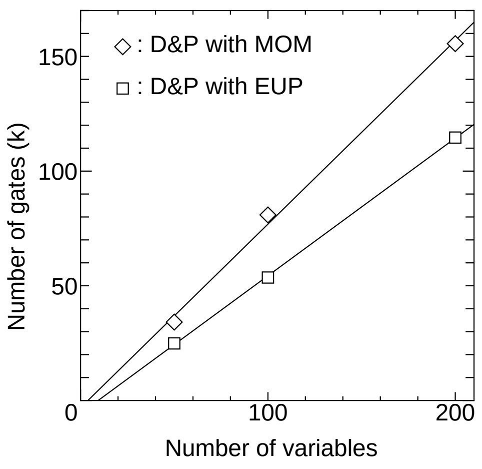

Fig. 3. Required gates for "aim-<#variables>-1_6-yes1-1.cnf"

The number of gates required to implement the proposed method is 30% less than that of MOM. In addition, the order of the proposed method is also better than that of MOM. We have actually implemented a benchmark problem with 128 variables that can run at 10MHz.

We are now refining the implementation of the algorithm on FPGAs, and planning to perform various evaluations on implemented logic circuits.

Acknowledgments

The authors would like to thank Yuichi Asahiro and Kazuo Iwama for providing AIM benchmark problems.

References

[1] Abramovici, M. and Saab, D.: Satisfiablilty on Reconfigurable Hardware, *International Workshop on Field Programmable Logic and Applications* (1997) 448–456
[2] Asahiro, Y., Iwama, K., and Miyano, E.: Random generation of test instances with controlled attributes, *Proceedings of the DIMACS Challenge II Workshop* (1993)
[3] Brown, S. D., Francis, R. J., Rose, J., and Vranesic, Z. G.: *Field-Programmable Gate Arrays*, Kluwer Academic Publishers (1992)
[4] Camposano, R. and Wolf, W.: *High-level VLSI synthesis*, Kluwer Academic (1991)
[5] Davis, M. and Putnam, H.: A computing procedure for quantification theory, *Journal of the ACM*, Vol. 7, (1960) 201–215
[6] Dubois, O., Andre, P., Boufkhad, Y., and Carlier, J.: Can a very simple algorithm be efficient for solving the SAT problem?, *Proc. of the DIMACS Challenge II Workshop* (1993)

[7] Freeman, J. W.: *Improvements to propositional satisfiability search algorithms*, PhD thesis, the University of Pennsylvania (1995)

[8] Hamadi, Y. and Merceron, D.: Reconfigurable Architectures: A New Vision for Optimizing Problem, *Proc. of Third International Conference on Principles and Practice on Constraint Programming (CP'97)* (1997) 209–221

[9] Li, C. M. and Anbulagan, : Heuristics Based on Unit propagation for Satisfiability Problems, *Proc. of 15th International Joint Conference on Artificial Intelligence* (1997) 366–371

[10] Mackworth, A. K.: *Encyclopedia of Artificial Intelligence*, Wiley-Interscience Publication (1992)

[11] Mitchell, D., Selman, B., and Levesque, H.: Hard and easy distributions of SAT problem, *Proc. of the Tenth National Conference on Artificial Intelligence* (1992) 459–465

[12] Nakamura, Y., Oguri, K., Nagoya, A., Yukishita, M., and Nomura, R.: High-level synthesis design at NTT systems labs., *IEICE Trans. Inf & Syst.*, Vol. E76-D, No. 9, (1993) 1047–1054

[13] Rashid, A., Leonard, J., and Mangione-Smith, W. H.: Dynamic Circuit Generation for Solving Specific Problem Instances of Boolean Satisfiability, *Proc. of IEEE Symposium on Field-Programmable Custom Computing Machines* (1998) 196–204

[14] Reiter, R.: A theory of diagnosis from first principles, *Artificial Intelligence 32(1):* (1987) 57–95

[15] Reiter, R. and Mackworth, A.: A logical framework for depiction and image interpretation, *Artificial Intelligence 41(2):* (1989) 125–155

[16] Suyama, T., Yokoo, M., and Sawada, H.: Solving Satisfiability Problems Using Logic Synthesis and Reconfigurable Hardware, *Proc. of the 31st Annual Hawaii International Conference on System Sciences Vol. VII* (1998) 179–186

[17] Yokoo, M., Suyama, T., and Sawada, H.: Solving satisfiability problems using field programmable gate arrays: First results, *Proc. of the Second International Conference on Principles and Practice of Constraint Programming*, Springer-Verlag (1996) 497–509

[18] Zhong, P., Martonosi, M., Ashar, P., and Malik, S.: Accelerating Boolean Satisfiability with Configurable Hardware, *Proc. of Symposium on Field-Programmable Custom Computing Machines* (1998) 186–195