International Conference on Principles and Practice of Constraint Programming
1996, pp. 497–509

Solving Satisfiability Problems using Field Programmable Gate Arrays: First Results

Makoto Yokoo, Takayuki Suyama and Hiroshi Sawada

NTT Communication Science Laboratories
2-2 Hikaridai, Seika-cho, Soraku-gun, Kyoto 619-02, Japan
e-mail: yokoo/suyama/sawada@cslab.kecl.ntt.jp

Abstract. This paper presents an initial report on an innovative approach for solving satisfiability problems (SAT), i.e., creating a logic circuit that is specialized to solve each problem instance on Field Programmable Gate Arrays (FPGAs). Until quite recently, this approach was unrealistic since creating special-purpose hardware was very expensive and time consuming. However, recent advances in FPGA technologies and automatic logic synthesis technologies have enabled users to rapidly create special-purpose hardware by themselves.
This approach brings a new dimension to SAT algorithms, since all constraints (clauses) can be checked simultaneously using a logic circuit. We develop a new algorithm called *parallel-checking*, which assigns all variable values simultaneously, and checks all constraints concurrently. Simulation results show that the order of the search tree size in this algorithm is approximately the same as that in the Davis-Putnam procedure. Then, we show how the parallel-checking algorithm can be implemented on FPGAs. Currently, actual implementation is under way. We get promising initial results which indicate that we can implement a hard random 3-SAT problem with 300 variables, and run the logic circuit at clock rates of about 1MHz, i.e., it can check one million states per second.

1 Introduction

A constraint satisfaction problem (CSP) is a general framework that can formalize various problems in Artificial Intelligence, and many theoretical and experimental studies have been performed [9]. 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 [3].

Virtually all existing SAT algorithms are intended to be executed on general-purpose sequential/parallel computers. As far as the authors know, there has been no study on solving SAT problems by creating a logic circuit specialized to solve each problem instance. This is because until quite recently, creating special-purpose hardware was very expensive and time consuming. Therefore, making a logic circuit for each problem instance was not realistic at all. However, due to recent advances in Field Programmable Gate Array (FPGA) technologies [1],

users can now create logic circuits by themselves, and reconfigure them electronically, without any help from LSI vendors. Furthermore, by using current automatic logic synthesis technologies [2], [11], users are able to design logic circuits automatically using a high level hardware description language (HDL). These recent hardware technologies have enabled users to rapidly create logic circuits specialized to solve each problem instance.

In this paper, we present an initial report on an innovative approach for solving SAT, i.e., creating a logic circuit that is specialized to solve each problem instance using FPGAs. This approach brings a new dimension to SAT algorithms, since all constraints (clauses) can be checked simultaneously using a logic circuit.

We develop a new algorithm called the *parallel-checking* algorithm. This algorithm has the following characteristics.

- Instead of determining variable values sequentially, all variable values are determined simultaneously, and all constraints are checked concurrently. Multiple variable values can be changed simultaneously when some constraints are not satisfied.
- In order to prune the search space, this algorithm introduces a technique similar to *forward checking* [8].

Simulation results show that the order of the search tree size in this algorithm is approximately the same as that in the Davis-Putnam procedure [5], which is widely used as a complete search algorithm for solving SAT problems.

Then, we show how the parallel-checking algorithm can be implemented on FPGAs by using recent hardware technologies. Currently, actual implementation is under way. We get promising initial results which indicate that we can implement a hard random 3-SAT problem with 300 variables, and run the logic circuit at clock rates of about 1MHz, i.e., it can check one million states per second.

In the remainder of this paper, we briefly describe the problem definition (Section 2), and describe the parallel-checking algorithm in detail (Section 3). Then, we show simulation results for evaluating the search tree size of this algorithm (Section 4). Furthermore, we show the way for implementing this algorithm on FPGAs and describe the status of the current implementation (Section 5). Finally, we discuss the relation of this algorithm with recently developed algorithms [4], [7], [6] that improve the Davis-Putnam procedure (Section 6).

2 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 the value true or false (represented as 1 or 0, respectively). In this paper, in order to simplify the algorithm description, we represent the fact that x_i is true as $(x_i, 1)$. We call the value assignment of one variable a *literal*. A *clause* is a disjunction of literals, e.g., $(x_1, 1) \vee (x_2, 0) \vee (x_4, 1)$, which represents a

logical formula $x_1 \vee \overline{x_2} \vee x_4$. 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

$$C_1 \wedge C_2 \wedge \ldots \wedge C_m$$

is satisfiable. That is, is there an assignment of values to the variables 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 aim to find only one solution. Although this setting corresponds to the original problem definition, some application problems, such as visual interpretation tasks [14], and diagnosis tasks [13], require finding all or multiple solutions. Furthermore, since finding all or multiple solutions is usually much more difficult than finding only one solution, solving the problem by special-purpose hardware will be worthwhile. Therefore, in this paper, we assume that the goal is to find all or multiple solutions.

In the following, 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.

3 Algorithm

3.1 Basic Ideas

We are going to describe the basic ideas of the parallel-checking algorithm. This algorithm is obtained by gradually improving a simple enumeration algorithm.

Simple Enumeration Algorithm: We represent one combination of value assignments of all variables as n-digit binary value. Assuming that variable x_i's value is v_i, a combination of value assignments can be represented by an n-digit binary value $\Sigma_{i=1}^{n} 2^{i-1} v_i$, in which the value of i's digit (counted from the lowest digit) represents the value of x_i. We call one combination of all variable values one *state*. In this algorithm, the state is incremented from 0 to $2^n - 1$. For each state, the algorithm checks whether clauses are satisfied. If all clauses are satisfied, the state is recorded as a solution. Obviously, this algorithm is very inefficient since it must check all 2^n states.

Introducing Backtracking: When some clauses are not satisfied, instead of incrementing x_1's digit, we can increment the lowest digit that is included in these unsatisfied clauses; thus the number of searched states can be reduced. The algorithm obtained after this improvement is very similar to the backtracking algorithm where the order of the variable/value selection is fixed.

Introducing Forward Checking (i): Furthermore, instead of checking the current value (0 or 1) only, if we check another value concurrently, we can reduce the number of searched states. For example, assume that there exist variables x_1, x_2, x_3, x_4, x_5 and three clauses:

C_1: $(x_1, 1) \vee (x_4, 1) \vee (x_5, 1)$,
C_2: $(x_1, 0) \vee (x_3, 1) \vee (x_4, 1)$,
C_3: $(x_1, 0) \vee (x_2, 1) \vee (x_5, 1)$.

The initial state $\{(x_1, 0), (x_2, 0), (x_3, 0), (x_4, 0), (x_5, 0)\}$ does not satisfy C_1. If we increment x_1's digit and change x_1's value to 1, then C_2 and C_3 are not satisfied. If we perform the check for the case that x_1's value is 1, we can confirm that incrementing x_1's digit is useless.

In this case, which digit should be incremented? If x_1 is 0, C_1 is not satisfied, and the second lowest digit in C_1 is x_4. If x_1 is 1, C_2 and C_3 are not satisfied, and the second lowest digit in C_2 is x_3, while the second lowest digit in C_3 is x_2. Therefore, we can conclude that at least x_3's digit must be changed to satisfy all clauses; changing digits lower than x_3 is useless.

This procedure is similar to the backtracking algorithm that introduces *forward checking* [8], where backtracking is performed immediately after some variable has no consistent value with the variables that have already assigned their values.

Introducing Forward Checking (ii): Another procedure that greatly contributes to the efficiency of forward checking is to assign the variable value immediately if the variable has only one value consistent with the variables that have already assigned their values. This procedure is called *unit resolution* in SAT.

In order to perform a similar procedure in this algorithm, for each variable x_i, we define a value called unit(x_i). If unit(x_i)=j, there exists only one possible value for x_i, which is consistent with the upper digit variables[1], and the second lowest digit in the clause that is constraining x_i is x_j's digit.

For example, in the initial state $\{(x_1, 0), (x_2, 0), (x_3, 0), (x_4, 0), (x_5, 0)\}$ of the problem described in 3.1, x_1 has only one consistent value 1 by C_1. Therefore, we set unit(x_1) to 4 (since x_4 is the second lowest digit in C_1), and change x_1's value to 1. The value of unit(x_1) represents the fact that unless at least x_4's digit is changed, x_1 has only one possible value. The next state will be $\{(x_1, 1), (x_2, 0), (x_3, 1), (x_4, 0), (x_5, 0)\}$. This state does not satisfy C_3. Since the lowest digit in C_3 is x_1, x_1's value is changed in the original procedure. However, the value of unit(x_1) is 4 and the second lowest digit in C_3 is x_2, where x_2 is lower than x_4. Therefore, x_2's value is changed. The next state will be $\{(x_1, 1), (x_2, 1), (x_3, 1), (x_4, 0), (x_5, 0)\}$. This state satisfies all of the three clauses.

[1] When there exist multiple possible values, unit(x_i)=i.

3.2 Details of the Algorithm

Term Definitions: In the following, we define concepts and terms used in the algorithm.

- Each clause $(x_i, v_i) \vee (x_j, v_j) \vee (x_k, v_k)$ is converted to the following rule (where flip(0)=1, flip(1)=0):

  $$\begin{aligned} &\text{if } (x_j, \text{flip}(v_j)) \wedge (\text{unit}(x_j) > i) \wedge (x_k, \text{flip}(v_k)) \wedge (\text{unit}(x_k) > i) \\ &\text{then} \qquad\qquad (x_i, \text{flip}(v_i)) \text{ is prohibited.} \end{aligned}$$

  It must be noted that one clause is converted to three different rules, since there are three possibilities for choosing the variable in the consequence part.
- Each rule is associated with the value flip(v_i) of variable x_i in the consequence part.
- We call a rule is *active* if the condition is satisfied in the current state.
- For value v_i of variable x_i, if some rule associated with v_i is active, we call value v_i is *prohibited.*
- For each active rule, if the variables in the condition part are x_j, x_k, we call min(unit(x_j), unit(x_k)) the *backtrack-position* of the rule.
- If value v_i of variable x_i is prohibited, the maximum of the backtrack-positions of active rules associated with v_i is called v_i's backtrack-position.
- When a state is given, we classify the condition of variable x_i in the following four cases:

  satisfied/free: both values are not prohibited.

  satisfied/constrained: the current value v_i is not prohibited, but flip(v_i) is prohibited.

  not-satisfied/possible: the current value v_i is prohibited, but flip(v_i) is not prohibited.

  not-satisfied/no-way: both values are prohibited.
- For variable x_i which is not-satisfied, we define the backtrack-position of x_i as follows:

  when x_i is not-satisfied/no-way: the minimum (the lower digit) of the backtrack-positions of values 0 and 1.

  when x_i is not-satisfied/possible: i (its own digit).

Parallel-Checking Algorithm: In the initial state, all variable values are 0, and the value of unit(x_i) is i.

1. For each value of each variable, concurrently check whether the associated rules are active. For a prohibited value, calculate the backtrack-position of the value. Calculate the condition and backtrack-position of each variable.
2. For each variable x_i, if its condition is satisfied/constrained, set the value of unit(x_i) to the backtrack-position of flip(v_i), where v_i is x_i's current value. Otherwise, set unit(x_i) to i.

3. Calculate m, which is the maximum of the backtrack-positions of not-satisfied variables. If all variables are satisfied, record the current state as a solution, and set m to 1.
4. Calculate max, which is the lowest digit that satisfies the following conditions: $max \geq m$, $v_{max} = 0$, and x_{max} is satisfied/free, where v_{max} is x_{max}'s current value. If there exists no value that satisfies these conditions, terminate the algorithm.
5. Change the value of x_{max} from 0 to 1. For each variable x_i which is lower than x_{max}, execute the following procedure:
   when x_i is satisfied/constrained, and unit(x_i) is larger than max: do not change x_i nor unit(x_i).
   when x_i is not-satisfied and for one of x_i's values v_i, v_i's backtrack-position is larger[2] than max: set x_i's value to flip(v_i), and set unit(x_i) to v_i's backtrack-position.
   otherwise: set x_i's value to 0, and set unit(x_i) to i.
6. Return to 1.

4 Simulation Results

In this section, we evaluate the efficiency of the parallel-checking algorithm by software simulation. We measured the number of searched states in the algorithm. For comparison, we used the Davis-Putnam procedure [5], which is widely used as a complete algorithm for solving SAT problems. The Davis-Putnam procedure is essentially a resolution procedure. It performs backtracking search by assigning the variable values and simplifying clauses. We call a clause that is simplified, such that it contains only one literal, a *unit clause*. When a unit clause is generated, the value of the variable that is contained in the unit clause is assigned immediately so that the unit clause is satisfied. This procedure is called *unit resolution*.

We use hard random 3-SAT problems as example problems. 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 the *clause density*, and the value 4.3 has been identified as the critical value that produces particularly difficult problems [10].

In Fig. 1, we show the log-scale plot of the average number of visited states over 100 example problems, by varying the number of variables n, where the clause density is fixed to 4.3. The number of visited states in the Davis-Putnam procedure is the number of binary choices made during the search; it does not include the number of unit resolutions. Since a randomly generated 3-SAT problem tends to have a very large number of solutions when it is solvable, in order

[2] Since m is the maximum of the backtrack-positions of all not-satisfied variables, the backtrack-positions of both values can not be larger than max, which is larger than m.

to finish the simulation within a reasonable amount of time, we terminate each execution after the first 100 solutions are found.

We perform a simple variable rearrangement before executing these algorithms, i.e., the variables are rearranged so that strongly constrained variables (variables included in many clauses) are placed in higher digits, and the variables that are related by constraints (the variables included in the same clause) are placed as close as possible. The Davis-Putnam procedure utilizes this rearrangement by selecting a variable in the order from x_n to x_1 (except for unit resolutions).

From Fig. 1, we can see the following facts.

- The number of visited states in the parallel-checking algorithm is three to eight times larger than that in the Davis-Putnam procedure. This result is reasonable since the number of states in the Davis-Putnam procedure does not include the number of unit resolutions, while the number of states in the parallel-checking includes state transitions that are caused by unit resolutions.
- The order of visited states (the order of the search tree size) in the parallel-checking algorithm is approximately the same as that in the Davis-Putnam procedure, i.e., for each algorithm, the number of visited states grows at the same rate as the number of variables increases.

The computation executed for each state in the Davis-Putnam procedure is in the order of $O(n)$ (which includes repeated applications of unit-resolutions). On the other hand, the computation executed for each state in the parallel-checking algorithm can be finished in one clock[3] when the algorithm is implemented on FPGAs.

These results indicate that the parallel-checking algorithm implemented on FPGAs will be much more efficient than the Davis-Putnam procedure implemented on a general-purpose computer.

5 Implementation

In this section, we give a brief description of FPGAs. Then, we show how the parallel-checking algorithm can be implemented on FPGAs, and report the current status of our implementation.

5.1 Field Programmable Gate Arrays

An example of FPGA architecture is shown in Fig. 2. It consists of a two-dimensional array of programmable logic blocks, with routing channels between

[3] The required time for one clock is not constant, since the possible clock rate of a logic circuit is determined by the delay of the logic circuit, and the delay is certainly affected by the problem size n. However, the order would be much smaller than $O(n)$, i.e., at most $O(log(n))$.

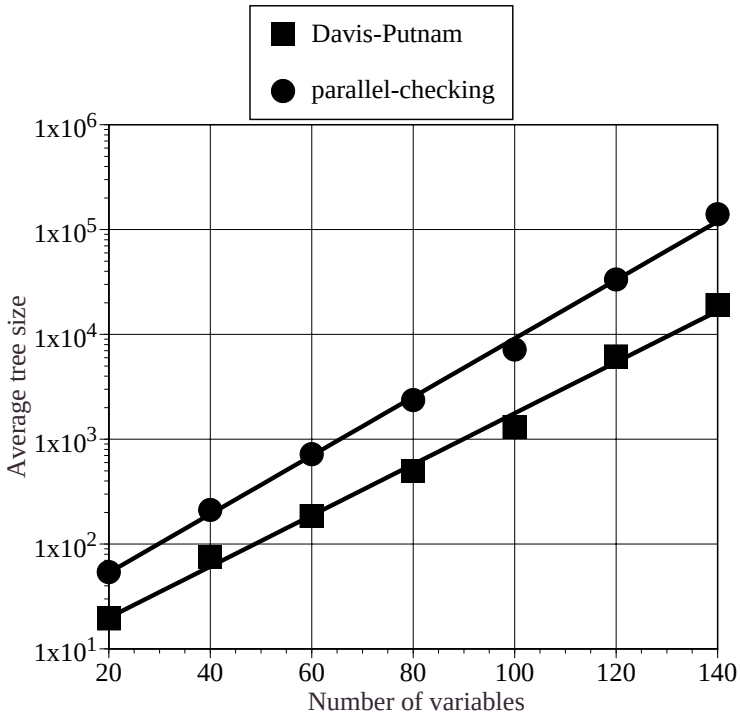

Fig. 1. Results for parallel-checking and Davis-Putnam procedure on hard random 3-SAT problems

these blocks. These logic blocks and interconnections are user-programmable by rewriting static RAM cells. We use an FPGA hardware system called ZyCAD RP2000 [16]. This system has 32 FPGA chips (each chip is a Xilinx XC4010), and can implement a large-scale logic circuit by dividing it into multiple FPGA chips. The equivalent gate count of a Xilinx XC4010 is about 8.0k to 10.0k.

5.2 Logic Circuit Configuration

We show the configuration of the logic circuit that implements the parallel-checking algorithm in Fig.3. The logic circuit consists of the following three functional units.

1. Rule Checker
2. Next State Generator
3. Next Unit Generator

In the Rule Checker, the condition and the backtrack-position of each digit are calculated from the current state and the unit values in parallel. The Next State

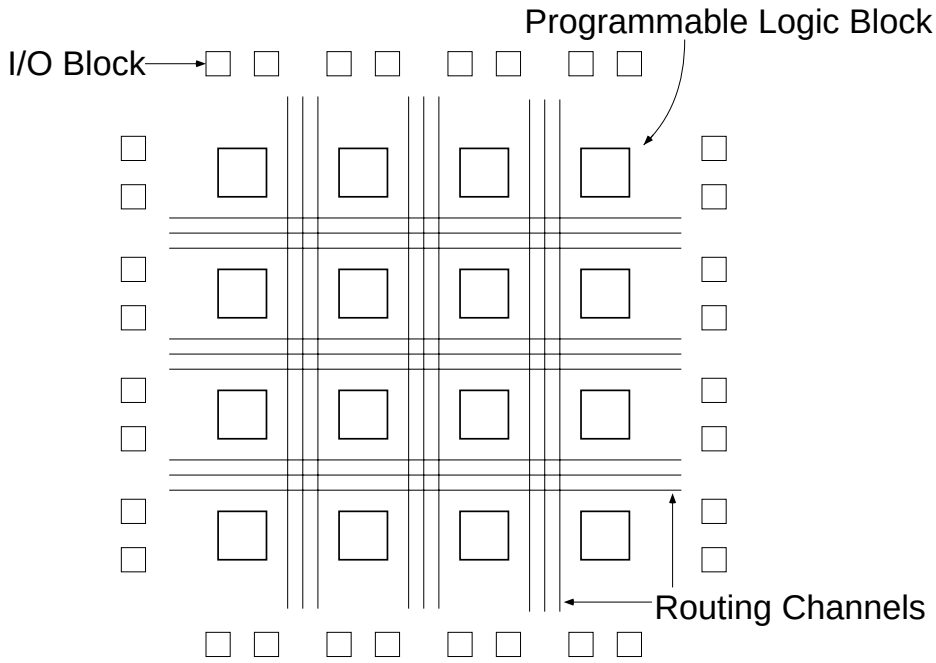

Fig. 2. General architecture of FPGA

Generator first calculates max, i.e., the digit that must be incremented, using the outputs of the Rule Checker. Then, the Next State Generator calculates the next state by incrementing the digit of max. The value of each lower digit is determined by its condition and backtrack-position. The Next Unit Generator calculates the unit values in the next state, using the outputs of the Rule Checker and max. These calculated values (the next state and next unit values) are used as feedback and stored in registers.

5.3 Logic Circuit Synthesis

A logic circuit that solves a specific SAT problem is synthesized by the following procedure (Fig. 4). 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. We use a system called PARTHENON [2], [11], which was developed at NTT. PARTHENON is a highly practical system that integrates a description language, simulator, and logic synthesizer. Furthermore, the FPGA Mapper of the Zycad system generates FPGA mapping data for RP2000 from the netlist.

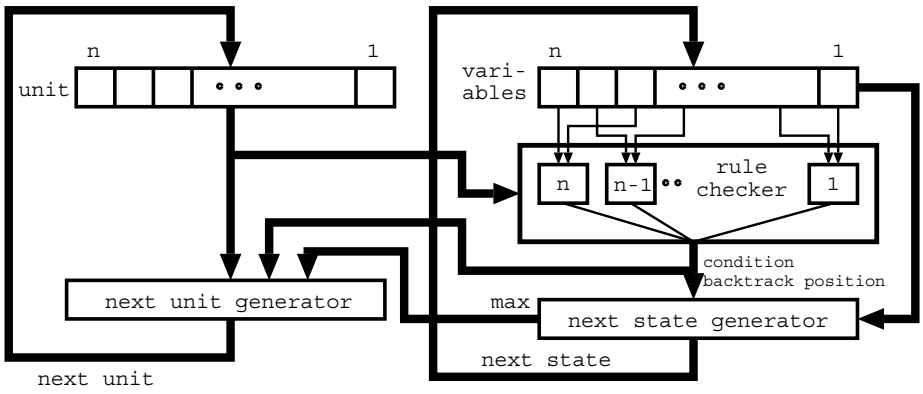

Fig. 3. Logic circuit configuration

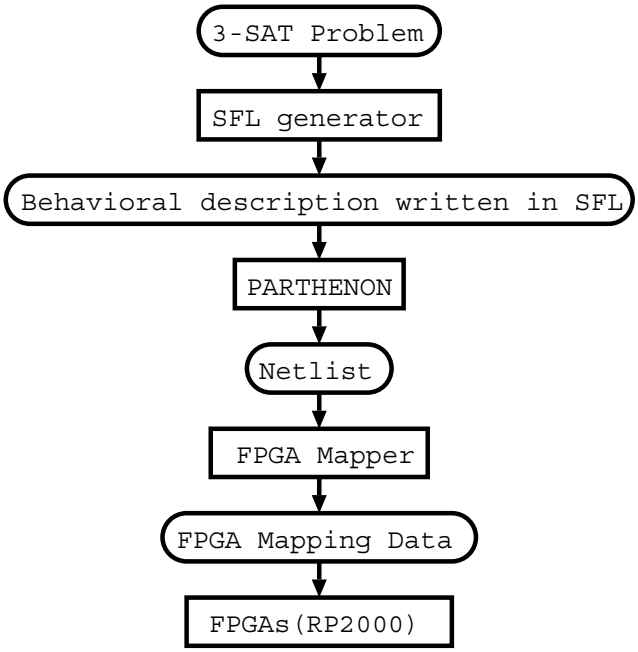

Fig. 4. Flow of logic circuit synthesis

5.4 Current Implementation Status

We have developed an SFL generator which generates a description of the logic circuit that implements the parallel-checking algorithm. By using this program, we have successfully implemented a hard random 3-SAT problem with 128 variables and 550 clauses using our current hardware resources. This logic circuit is capable of running at the clock rates of at least 1MHz, which is the maximal setting of the clock generator we are currently using. Therefore, we can assume that it is capable of running at higher clock rates. The logic circuit for a 128-variable problem fits in the current hardware resources. Since the number of FPGA chips in this system can be increased up to 64 (twice as many as the current configuration) and the mapping quality can stand further improvement, we could possibly implement much larger problems, e.g., a problem with 300 variables, without any trouble[4]. We are increasing our hardware resources and trying to implement much larger problems.

Currently, generating a logic circuit from a problem description takes a few hours. This is because we are using general-purpose synthesis routines. These routines can be highly optimized for SAT problems since many parts in a logic circuit are common in all problem instances. The required time for generating a logic circuit could be reduced to at most several ten minutes.

6 Discussions

Recently, several improved versions of the Davis-Putnam procedure have been developed [4], [6], [7]. These algorithms use various sophisticated variable/value ordering heuristics. How good is the parallel-checking algorithm compared with these algorithms? Unfortunately, these algorithms aim to find only one solution, and various procedures for simplifying formulas, such as removing variable values that do not affect the satisfiability of the problem, are introduced. Therefore, the evaluation results of these algorithms can not be compared directly with the results of the parallel-checking algorithm. It is not very straightforward to modify these algorithms so that they can find all solutions. In our future works, we are going to examine these algorithms carefully, modify them so that they can find all solutions, and compare the modified algorithms with the parallel-checking algorithm. Furthermore, we are going to examine the possibility of introducing the heuristics used in these algorithms into the parallel-checking algorithm.

7 Conclusions and Future Works

This paper presented an initial report on solving SAT using FPGAs. In this approach, a logic circuit specific to each problem instance is created on FPGAs. This approach brings a new dimension to SAT algorithms since all constraints

[4] Of course, the efficiency of the parallel-checking algorithm must be improved in order to solve such a large-scale problem within a reasonable amount of time.

can be checked in parallel using a logic circuit. We developed a new algorithm called *parallel-checking*, which assigns all variable values simultaneously, and checks all constraints concurrently. Simulation results showed that the order of the search tree size in the parallel-checking algorithm is approximately the same as that in the Davis-Putnam procedure, which is widely used as a complete algorithm for solving SAT problems. We have implemented a hard random 3-SAT problem with 128 variables, and run the logic circuit at clock rates of about 1MHz, i.e., it can check one million states per second. Currently, we are increasing our hardware resources so that much larger problems can be implemented. We are going to perform various evaluations on implemented logic circuits.

Our future works include comparing this approach to recently developed algorithms [4], [6], [7] that improve the Davis-Putnam procedure, and introducing the heuristics used in these algorithms into the parallel-checking algorithm. Furthermore, we are going to implement iterative improvement algorithms for solving SAT [12], [15] on FPGAs.

Acknowledgments

The authors wish to thank K. Matsuda for supporting this research. We also appreciate helpful discussions with N. Osato and A. Nagoya.

References

1. Brown, S. D., Francis, R. J., Rose, J., and Vranesic, Z. G.: *Field-Programmable Gate Arrays*, Kluwer Academic Publishers (1992)
2. Camposano, R. and Wolf, W.: *High-level VLSI synthesis*, Kluwer Academic Publishers (1991).
3. Cook, S.: The complexity of theorem proving procedures, *Proceedings of the 3rd Annual ACM Symposium on the Theory of Computation* (1971) 151–158
4. Crawford, J. M. and Auton, L. D.: Experimental results on the crossover point in satisfiability problems, *Proceedings of the Eleventh National Conference on Artificial Intelligence* (1993) 21–27
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.: SAT versus UNSAT, in Johnson, D. S. and Trick, M. A. eds., *Proceedings of the Second DIMACS Implementation Challenge*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science (1993)
7. Freeman, J. W.: *Improvements to propositional satisfiability search algorithms*, PhD thesis, the University of Pennsylvania (1995)
8. Haralick, R. and Elliot, G. L.: Increasing tree search efficiency for constraint satisfaction problems, *Artificial Intelligence*, Vol. 14, (1980) 263–313
9. Mackworth, A. K.: Constraint satisfaction, in Shapiro, S. C. ed., *Encyclopedia of Artificial Intelligence*, Wiley-Interscience Publication, New York (1992) 285–293, second edition
10. Mitchell, D., Selman, B., and Levesque, H.: Hard and easy distributions of SAT problem, *Proceedings of the Tenth National Conference on Artificial Intelligence* (1992) 459–465

11. 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, pp. 1047–1054 (1993).
12. Morris, P.: The breakout method for escaping from local minima, *Proceedings of the Eleventh National Conference on Artificial Intelligence* (1993) 40–45
13. Reiter, R.: A theory of diagnosis from first principles, *Artificial Intelligence*, Vol. 32, No. 1, (1987) 57–95
14. Reiter, R. and Mackworth, A.: A logical framework for depiction and image interpretation, *Artificial Intelligence*, Vol. 41, No. 2, (1989) 125–155
15. Selman, B., Levesque, H., and Mitchell, D.: A new method for solving hard satisfiability problems, *Proceedings of the Tenth National Conference on Artificial Intelligence* (1992) 440–446
16. Zycad Corp.: *Paradigm RP Concept Silicon User's Guide, Hardware Reference Manual, Software Reference Manual* (1994).