31st Hawaii International Conference on System Sciences (HICSS-31), 1998

Solving Satisfiability Problems Using Logic Synthesis and Reconfigurable Hardware

Takayuki Suyama Makoto Yokoo Hiroshi Sawada
NTT Communication Science Laboratories
{*suyama, yokoo, sawada*}*@cslab.kecl.ntt.co.jp*

Abstract

This paper presents new results on an 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). This approach becomes feasible due to the recent advances in FPGAs and high-level logic synthesis. In this approach, each SAT problem is automatically analyzed and implemented on FPGAs.

We have developed an algorithm which is suitable for implementing on a logic circuit. This algorithm is equivalent to the Davis-Putnam procedure with a powerful dynamic variable ordering heuristic. The algorithm does not have a large memory structure like a stack; thus sequential accesses to the memory do not become a bottleneck in algorithm execution.

Simulation results show that this method can solve a hard random 3-SAT problem with 400 variables within 20 minutes at a clock rate of 1MHz.

1. Introduction

Recently, due to advances in Field Programmable Gate Array (FPGA) technologies [1], users can create original logic circuits and reconfigure them as well. Furthermore, by using current high level logic synthesis technologies [2], [3], users are able to describe their designs in a Hardware Description Language (HDL) and obtain the corresponding logic circuits. These recent hardware technologies enable users to rapidly create logic circuits specialized to solve each problem instance. We have selected satisfiability problems to test 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 [4]. In particular, a satisfiability problem (SAT) for propositional formulas in conjunctive normal form is an important subclass of CSP. Historically, this problem was the first computational task shown to be NP-hard [5].

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, except for [6]. This is because until quite recently, creating special-purpose hardware had been very expensive and time consuming. Therefore, making a logic circuit for each problem instance has not been realistic so far.

In [6], the initial report on solving the SAT problem by this approach is presented. This report describes a way of implementing an algorithm whose performance is equivalent to the basic Davis-Putnam procedure [7]. However, this algorithm is not efficient since the variable ordering is static. This algorithm cannot handle a large-scale problem such as a hard random 3-SAT problem with 400 variables within a reasonable amount of time, even if the algorithm is implemented on a logic circuit.

In this paper, we present a procedure of implementing a more efficient algorithm with a powerful dynamic variable ordering heuristic on a logic circuit. This algorithm is basically equivalent to the Davis-Putnam procedure with Mom's heuristic [8], [9]. The main feature of this algorithm is the elimination of a data structure like a stack to maintain the history of the tree search. Thus, sequential accesses to the memory do not become a bottleneck in algorithm execution.

Simulation results indicate that the number of clocks required to solve a hard random 3-SAT problem with 400 variables will be around 10^9. If we run the logic circuit at a clock rate of 1MHz, the problem can be solved within 20 minutes. As far as the authors' knowledge, it requires about a few hours to solve such problems using complete algorithms on a general purpose computer. We can obtain a further speedup if the logic circuit is capable of running at higher clock rates.

We show how the algorithm can be implemented on FPGAs by using recent hardware technologies.

In the remainder of this paper, we define the SAT problem (Section 2), and introduce the design flow of logic synthesis (Section 3). Then, we describe in detail the developed algorithm which is suitable for implementing on a logic circuit (Section 4). Furthermore, we show a procedure of implementing this algorithm on FPGAs (Section 5). Finally, we provide evaluation results obtained by software simulation and describe the status of the current implementation (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 equal to either *true* or *false* (represented as 1 or 0, respectively). The value assignment of one variable is called 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, i.e., to determine whether there exists 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 problems aim to find only one solution. Although this setting corresponds to the original problem definition, some application problems, such as visual interpretation tasks [10], and diagnosis tasks [11], require finding all or multiple solutions. Furthermore, since finding all or multiple solutions is usually much more difficult than finding only a single solution, solving the problem by special-purpose hardware is worthwhile. Therefore, in this paper we set our goal to finding 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 straightforward.

3. Design Flow

First, a brief description of FPGAs is provided. Then, we show how to synthesis logic circuits for FPGAs.

3.1. Field Programmable Gate Arrays

An example of FPGA architecture is shown in Figure 1. It consists of a two-dimensional array of programmable logic blocks, with routing channels between these blocks. The functions of these logic blocks and interconnections are user-programmable by rewriting static RAM cells.

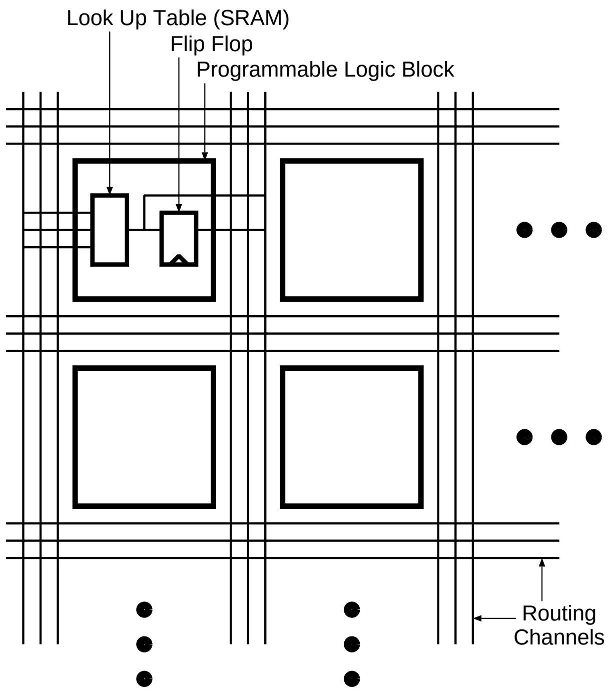


Figure 1. General architecture of FPGA

3.2. Logic Circuit Synthesis

Typically, software is sequentially executed on general-purpose computers. In addition, access to all data in memories is controlled by the pointers in this case. Hardware can basically operate in parallel. In order to transfer data, their senders and receivers are connected by physical connections. Because of these differences, we employed following strategies;

1. a logic circuit is synthesized for solving each problem instance.

2. an algorithm suitable for hardware is synthesized as logic circuits.

The first strategy is derived from the flexibility of hardware. Namely, hardware has less flexibility than software. Therefore, synthesizing the logic circuit for each instance is better than that for a class such as SAT. The second one is caused by the characteristic differences between hardware and software.

A logic circuit that solves a specific SAT problem is synthesized by the procedure depicted in Figure 2. First, a text file that describes a SAT problem is analyzed by C program called "SFL generator". This program generates a behavioral hardware description specific to the given problem with a Hardware Description Language (HDL) called SFL (Structured Function description Language). Then, a logic synthesis system analyzes the description and synthesizes a netlist, which describes the logic circuit structure. We use a system called PARTHENON [2], [3], which was developed by NTT. PARTHENON is a system that integrates a description language, simulator, and logic synthesizer, which synthesize logic circuits from a behavioral hardware description written in HDL. Finally, the FPGA Mapper of the FPGA system generates FPGA mapping data from the netlist.

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 heuristic. 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 satisfied yet.

1. If L is empty, then I is unsatisfiable, 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:** select the variable x having the maximum number of occurrences in binary clauses (clauses with two variables) of S. Add to the top of L the simplified instance S' obtained by setting x to true. Set x to false and simplify S. Go to 2.

The variable ordering heuristic used for the branching is called *Maximum Occurrences in clauses of Minimum Size (Mom's) heuristic*. Although this heuristic is very simple, it has been reported to be very effective [8], [9] in improving the efficiency of the Davis-Putnam procedure.

4.2. Modification to the Basic Algorithm

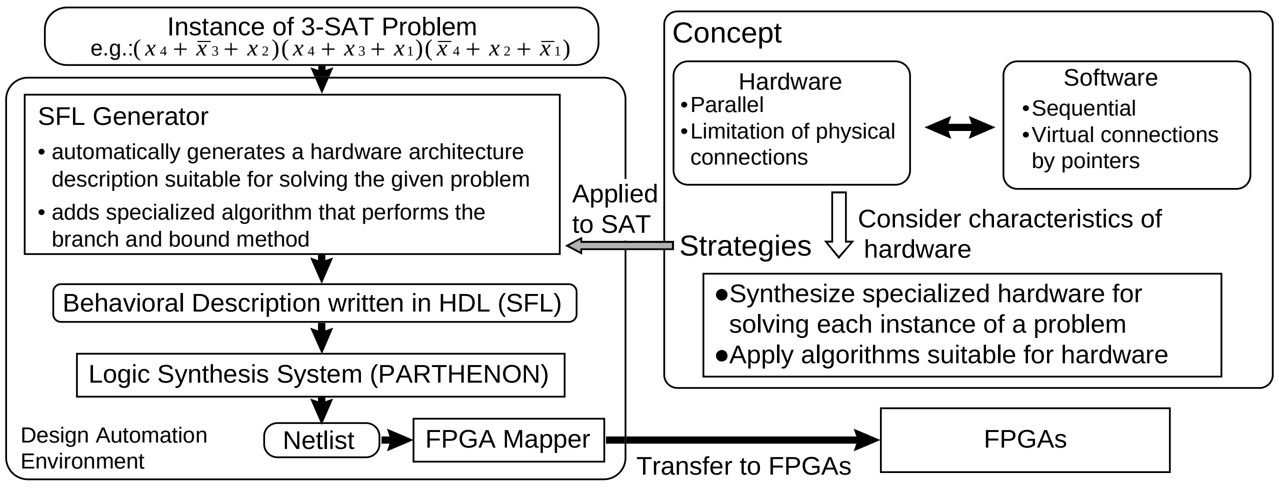
In most cases, such a backtracking tree search algorithm is implemented using a stack. However, implementing a stack is not appropriate since having a large memory is difficult in a logic circuit. Even if we can manage to implement a large memory, sequential accesses to the memory can become a bottleneck in algorithm execution. We avoid the overhead of sequential accesses to a large memory 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.

On the other hand, one merit of using a logic circuit is that all constraints (clauses) can be checked simultaneously. In order to make use of this advantage, we change the algorithm so that if there exist multiple unit clauses, multiple variable values are determined at the same time. The details of the algorithm are described in the following.

4.3. Details of the Algorithm

In the following, we define concepts and terms used in the algorithm. In order to simplify the algorithm description, 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.



Instance of 3-SAT Problem
e.g.:$(x_4 + \overline{x}_3 + x_2)(x_4 + x_3 + x_1)(\overline{x}_4 + x_2 + \overline{x}_1)$
SFL Generator
• automatically generates a hardware architecture description suitable for solving the given problem
• adds specialized algorithm that performs the branch and bound method
Behavioral Description written in HDL (SFL)
Logic Synthesis System (PARTHENON)
Netlist
FPGA Mapper
Design Automation Environment
Applied to SAT
Concept
Hardware
•Parallel
•Limitation of physical connections
Software
•Sequential
•Virtual connections by pointers
Strategies
Consider characteristics of hardware
●Synthesize specialized hardware for solving each instance of a problem
●Apply algorithms suitable for hardware
Transfer to FPGAs
FPGAs

Figure 2. Flow of logic circuit synthesis

- 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 determined yet. 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. The initial value is 0.

- 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: $determined(x_i) = 1$ and x_i's value is v_i, $determined(x_j) = 1$ and x_j's value is v_j, or $determined(x_k) = 1$ 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.

  binary: For any one of x_i, x_j, x_k, its variable value is determined, and for the remaining two variables, their values are not determined. The value is not equal to v_i, v_j, v_k, respectively, for the variable whose value is determined.

  trinary: For each variable x_i, x_j, x_k, its variable value is not determined.

- If there exist multiple unit clauses, and these unit clauses assign different values to the same variable, we call these unit clauses *inconsistent*.

In the following, we show the details of the algorithm. In the initial state, $current\_depth$ is 0, and for each variable x_i, $determined(x_i)$ and $branch(x_i)$ is 0.

1. **Not Satisfied:** If there exists a *not-satisfied* clause, or there exist inconsistent *unit* clauses, go to 5.

2. **Satisfied:** If all clauses are *satisfied*, print the current value assignments as a solution. Go to 5.

3. **Unit:** If there exist *unit* clauses, and these clauses are not inconsistent, for all unit clauses, for each variable x_i where $determined(x_i)$ 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, select the variable x_i, where $determined(x_i) = 0$ and x_i has the maximum

number of occurrences in *binary* clauses, set x_i's value to 0, $determined(x_i)$ to 1, $branch(x_i)$ to 1, and $depth(x_i)$ to $current\_depth + 1$. Increment $current\_depth$ by 1. Go to 1.

5. **Backtracking:**

   5.1 If $current\_depth$ is 0, 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's value 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.

For example, let's assume there are four variables x_1, x_2, x_3, and x_4, and four clauses $C_1 = (x_1 + x_2)$, $C_2 = (x_1 + \overline{x_3})$, $C_3 = (\overline{x_3} + x_4)$, and $C_4 = (x_3 + \overline{x_4})$.

First, we set x_1's value to 0, $depth(x_1) = 1$, $determined(x_1) = 1$, and $branch(x_1) = 1$. C_1 and C_2 become unit clauses, so we set x_2's value to 1, and x_3's value to 0 (where $depth(x_2) = depth(x_3) = 1$, $branch(x_2) = branch(x_3) = 0$, and $determined(x_2) = determined(x_3) = 1$). Then, C_3 and C_4 become unit clauses, but they are inconsistent (these clauses try to assign different values to x_4). Since $current\_depth = depth(x_1) = depth(x_2) = depth(x_3)$, x_1's value is changed to 1 (where $branch(x_1) = 0$, $depth(x_1) = 0$), and $determined(x_2)$ and $determined(x_3)$ becomes 0. Then, C_1 and C_2 are satisfied, and x_4's value is set to 0 (where $depth(x_4) = 1$, $branch(x_4) = 1$). Then, C_3 becomes a unit clause, so we set x_2's value to 0. At this point, all clauses are satisfied. The algorithm continues for finding other solutions.

5. Implementation

5.1. Hardware Architecture

The algorithm described in Section 4 can be straightforwardly represented as a finite state machine. 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 described in Figure 3.

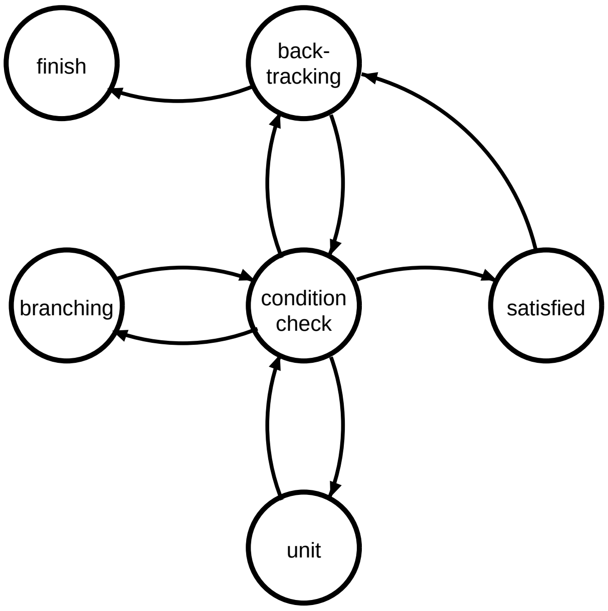


Figure 3. Transitions of the finite state machine

- condition check
- backtracking
- satisfied
- unit
- branching
- finish

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

$$(determined(x_i) \cdot determined(x_j) \cdot determined(x_k)) \cdot ((x_i \oplus v_i) \cdot (x_j \oplus v_j) \cdot (x_k \oplus v_k))$$
where $\oplus$ is Exclusive-OR

After the conditions of the clauses are determined and integrated, the next state will be either *backtracking*, *satisfied*, *unit*, or *branching*.

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 $current\_depth = 0$, 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 *condition check* state.

In the *branching* state, for each variable, the number of occurrences in the *binary* clauses must be counted. We implement counters for each variable. These counted numbers are compared and the variable with the maximum value is selected. In a naive implementation, the counting procedures require as many clock cycles as the maximum number of occurrences of variables in the clauses. We can speed up these procedures by counting several clauses for each variable in one clock cycle. The comparison procedures require about one to five clock cycles, depending on the number of variables and the way the logic circuit is implemented. For example, in the current naive implementation of a 3-SAT problem with 30 variables, 80 clauses, the counting procedures require eight clock cycles, and the comparison procedures require one clock cycles. These numbers can be improved using more sophisticated implementation techniques.

6. Evaluation

6.1. Simulation

In this section, we first evaluate the efficiency of the developed algorithm by software simulation. We measure the required number of clocks when the algorithm is implemented on a logic circuit.

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 [12].

As described in Section 5.1, when the algorithm in Section 4 is implemented on a logic circuit, one clock cycle is required to determine whether the next state will be *backtracking*, *satisfied*, *unit*, or *branching*. The procedure for *backtracking* requires another one clock cycle. If the current state is *unit*, the next state can be generated within the same clock cycle. On the other hand, the *branching* procedure is rather complicated, and requires about 10 clock cycles, depending on the number of variables and the way the logic circuit is implemented.

In Figure 4, we show the log-scale plot of the average number of required clocks over 100 example problems, by varying the number of variables n, where the clause density is fixed to 4.3. The required number of clocks for the *branching* procedure is set to 10. For additional information, we show the number of branches, i.e., the size of the search tree in Figure 4. Since a randomly generated 3-SAT problem tends to have a very large number of solutions when it is solvable, in order to finish the simulation within a reasonable amount of time, we terminate each execution after the first 100 solutions are found.

We can see the following facts from Figure 4.

- The required number of clocks for a 400-variable problem is estimated to be 10^9. Therefore, if we run the logic circuit at the clock rate of 1MHz, the problem can be solved in 1000 seconds, i.e., less than 20 minutes. If we can increase the clock rate, a further speedup can be obtained. As far as the authors' knowledge, finding a solution for a 400-variable hard random 3-SAT problem requires a few hours, even by a very sophisticated algorithm optimized to find only one solution [8], [13], [14].

- The search tree size grows at the rate of $O(2^{n/17.3})$, where n is the number of variables. This rate is comparable to that of sophisticated algorithms (e.g., $O(2^{n/18.7})$ in POSIT [8]). We can see that although this algorithm is very simple, it is fairly efficient.

6.2. Current Implementation Status

We use an FPGA hardware system called ZyCAD RP2000 [15]. This system has 16 FPGA chips (each chip is a Xilinx XC4025), and can implement a large-scale logic circuit by dividing it into multiple FPGA chips. The typical usable gate count of a Xilinx XC4025 is about 15.0k. We have actually implemented a particularly difficult 3-SAT problem instance with 30 variables, 80 clauses, which was created by a problem generator contributed to

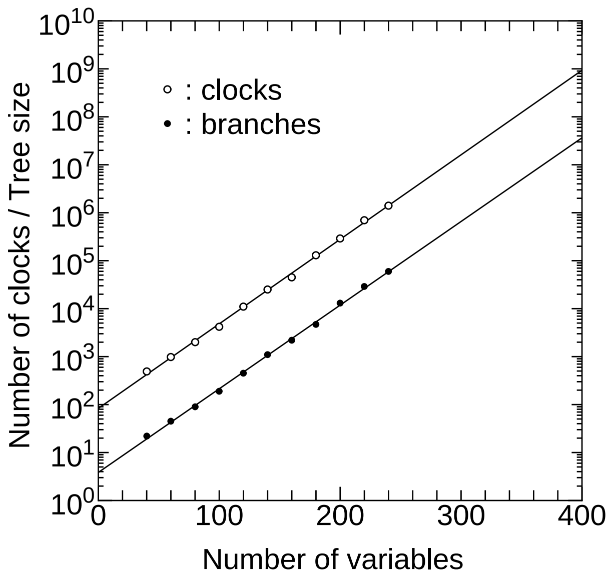


Figure 4. Required clocks on hard random 3-SAT problems

the DIMACS Challenge by Oliver Dubois [9]. This logic circuit is capable of running at a clock rate of at least 1MHz, which is the maximal setting of the clock generator we are currently using. A logic circuit simulator shows that the delay of this circuit is around 108.2 nsec, which means that this circuit is capable of running at a clock rate of 9.24 MHz.

Although the logic circuit for this problem fits in the current hardware resources, and there are plenty of logic blocks left (less than 10% of logic blocks are used), we are having difficulty for implementing larger-scale problems. This is because a logic circuit for solving a SAT problem requires exceptionally many routing resources. For example, the logic circuit for the *branching* procedure needs to compare n numbers in parallel, where n is the number of variables. Therefore, routing resources of the FPGA system we are using can be exhausted fairly quickly.

This problem can be avoided by using different types of FPGA systems that have more routing resources (and less logic blocks). Alternatively, we can create a custom LSI for functions that are common to all problem instances (such as the comparison unit), and create only the instance-specific parts on FPGAs. Now, We are trying a new implementation which requires less wire resources. We are also currently renewing our hardware resources and trying to implement much larger problems. Since FPGA technologies are advancing very rapidly, we assume that this resource limitation can be removed soon.

Currently, generating a logic circuit from a problem description takes a few hours. This is because we use general-purpose synthesis routines. Most of this generation time is spent on optimizing the allocation of the logic circuit. These routines can be highly optimized for SAT problems since many parts in a logic circuit are common to all problem instances, and the allocation problem does not have to be solved from scratch for each problem instance. The required time for generating a logic circuit can be reduced to at most several ten minutes.

7. Conclusions

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

Simulation results showed that by this approach, a hard random 3-SAT problem with 400 variables can be solved within 20 minutes at a clock rate of 1MHz, while complete algorithms implemented on a general purpose computer require a few hours for solving such a problem. A further speedup can be obtained if we can increase the clock rate.

We implemented a hard random 3-SAT problem with 30 variables, and ran the logic circuit at a clock rate of 1MHz. Currently, the routing resource limitation of our FPGA system prevents us from implementing larger-scale problems. We are renewing our hardware resources and trying to implement larger-scale problems.

References

[1] S. D. Brown, R. J. Francis, J. Rose, and Z. G. Vranesic, *Field-Programmable Gate Arrays*, Kluwer Academic Publishers, 1992.

[2] R. Camposano, and W. Wolf, *High-level VLSI synthesis*, Kluwer Academic Publishers, 1991.

[3] Y. Nakamura, K. Oguri, A. Nagoya, M. Yukishita, and R. Nomura, "High-level synthesis design at NTT

systems labs.", *IEICE Trans. Inf & Syst. E76-D(9)*, 1993, pp. 1047-1054.

[4] A. K. Mackworth, "Constraint satisfaction", *Encyclopedia of Artificial Intelligence*, S. C. Shapiro, Wiley-Interscience Publication, New York, 1992, pp. 285-293.

[5] S. Cook, "The complexity of theorem proving procedures", *Proceedings of the 3rd Annual ACM Symposium on the Theory of Computation* 1971, pp. 151-158.

[6] T. Suyama, M. Yokoo, and H. Sawada, "Solving satisfiability problems on FPGAs", *Proceedings of the 6th International Workshop on Field Programmable Logic and Applications*, Springer, 1996, pp. 136-145.

[7] M. Davis and H. Putnam, "A computing procedure for quantification theory", *Journal of the ACM 7*, 1960, pp. 201-215.

[8] J. W. Freeman, *Improvements to propositional satisfiability search algorithms*, Ph.D. Dissertation, the University of Pennsylvania, 1995.

[9] O. Dubois, P. Andre, Y. Boufkhad, and J. Carlier, "Can a very simple algorithm be efficient for solving the SAT problem?" *Proceedings of the DIMACS Challenge II Workshop*, 1993a.

[10] R. Reiter, and A. Mackworth, "A logical framework for depiction and image interpretation", *Artificial Intelligence 41(2)*, 1989, pp. 125-155.

[11] R. Reiter, "A theory of diagnosis from first principles", *Artificial Intelligence 32(1)*, 1987, pp. 57-95.

[12] D. Mitchell, B. Selman, and H. Levesque, "Hard and easy distributions of SAT problem", *Proceedings of the Tenth National Conference on Artificial Intelligence*, 1992, pp. 459-465.

[13] O. Dubois, P. Andre, Y. Boufkhad, and J. Carlier, "SAT versus UNSAT", *Proceedings of the DIMACS Challenge II Workshop*, 1993b.

[14] J. M. Crawford, and L. D Auton, "Experimental results on the crossover point in satisfiability problems", *Proceedings of the Eleventh National Conference on Artificial Intelligence*, 1993, pp. 21-27.

[15] Zycad Corp., *Paradigm RP Concept Silicon User's Guide, Hardware Reference Manual, Software Reference Manual*, 1994.