Solving Satisfiability Problems Using Reconfigurable Computing

Takayuki Suyama, Makoto Yokoo, Hiroshi Sawada, and Akira Nagoya, *Member, IEEE*

***Abstract*—This paper reports on an innovative approach for solving satisfiability problems for propositional formulas in conjunctive normal form (SAT) by 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. SAT is an important subclass of constraint satisfaction problems, which can formalize a wide range of application problems.**

We have developed a series of algorithms that are suitable for a logic circuit implementation, including an algorithm whose performance is equivalent to the Davis–Putnam procedure with powerful dynamic variable ordering. Simulation results show that this method can solve a hard random 3-SAT problem with 400 variables within 1.6 min at a clock rate of 10 MHz. Faster speeds can be obtained by increasing the clock rate. Furthermore, we have actually implemented a 128-variable 256-clause problem instance on FPGAs.

***Index Terms*—Algorithms, reconfigurable computing, satisfiability problems, search.**

I. Introduction

Recently, due to advances in field programmable gate array (FPGA) technologies [4], 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 existing high level logic synthesis technologies [5], [6]. These recent hardware technologies enable users to rapidly create logic circuits specialized to solve each problem instance. We call this problem solving approach the *Reconfigurable Computing* approach.

A constraint satisfaction problem (CSP) is a general framework that can formalize various application problems, and many theoretical and experimental studies have been performed on these problems [7]. 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 [8].

In this paper, we report on an innovative approach for solving SAT using the reconfigurable computing approach to create a logic circuit that is specialized to solve each problem instance on FPGAs. After the authors presented an initial report on this approach [1], various research efforts following this line have been carried out [9]–[15]. Consequently, solving SAT using FPGAs has become a vital research area.

In the remainder of the paper, we briefly describe the problem definition in Section II and the reconfigurable computing approach in Section III. Then, we present in detail the developed algorithm, which is suitable for implementation on a logic circuit. Such algorithms can be divided into two groups, i.e., algorithms using static variable ordering in Section IV and algorithms using dynamic variable ordering in Section V. Then, we show how these algorithms can be implemented on FPGAs in Section VI. Next, we show evaluation results obtained with the software simulation and actual implementation in Section VII. We discuss related works in Section VIII. Finally, we conclude in Section IX.

II. 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 such 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 [16] and diagnosis tasks [17], 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 a productive approach. Therefore, in this paper we set our goal to finding all or multiple solutions.

In the following sections, for simplicity we restrict our attention to 3-SAT problems, i.e., the number of literals in each clause is three. Any general SAT problem instance can be transformed into a 3-SAT problem instance by introducing auxiliary variables.

Manuscript received February 28, 2000; revised July 15, 2000. This paper is an extended version of the authors' previous conference papers [1]–[3].

T. Suyama is with Research and Development Center, NTT West, Tokyo, Japan (e-mail: t.suyama@rdc.west.ntt.co.jp).

M. Yokoo and H. Sawada are with NTT Communication Science Laboratories, Kyoto, Japan (e-mail: yokoo@cslab.kecl.ntt.co.jp; sawada@cslab.kecl.ntt.co.jp).

A. Nagoya is with NTT Network Innovation Laboratories, Kanagawa, Japan (e-mail: nagoya@exa.onlab.ntt.co.jp).

Publisher Item Identifier S 1063-8210(01)00713-2.

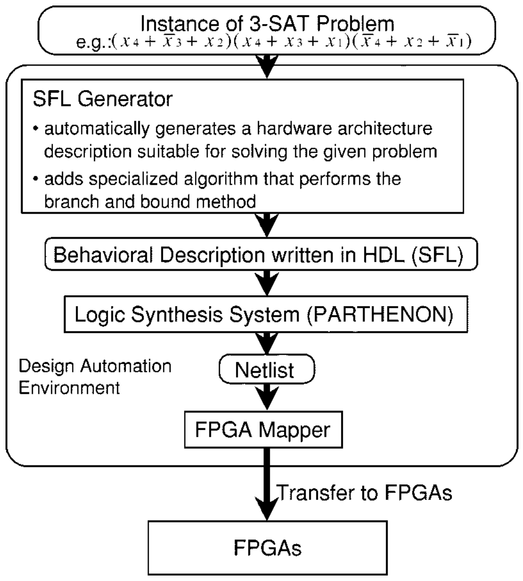

Instance of 3-SAT Problem
e.g.:$(x_4+\bar{x}_3+x_2)(x_4+x_3+x_1)(\bar{x}_4+x_2+\bar{x}_1)$
SFL Generator
• automatically generates a hardware architecture description suitable for solving the given problem
• adds specialized algorithm that performs the branch and bound method
Behavioral Description written in HDL (SFL)
Logic Synthesis System (PARTHENON)
Design Automation Environment
Netlist
FPGA Mapper
Transfer to FPGAs
FPGAs

Fig. 1. Flow of logic circuit synthesis and mapping to FPGAs.

III. Reconfigurable Computing

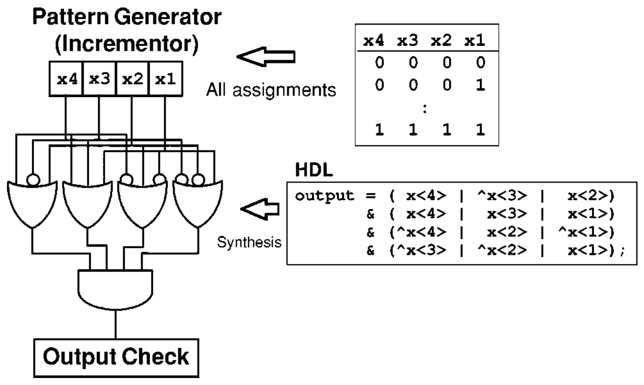
In this section, we describe our RC approach. RC systems are hardware systems with logical configurations that can be changed to solve a problem and/or quickly carry out an application. These systems are realized with FPGAs and logic synthesis systems. Conventional RC systems are reconfigured to a target application. On the other hand, our RC system features reconfiguration for each 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 that 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 significant speed increases by 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) in 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 bookkeeping becomes 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 by 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 procedure depicted in Fig. 1. First, a text file that describes a SAT problem is analyzed by a C program called “SFL generator.” This program generates a behavioral hardware description specific to the given problem with an HDL called SFL (Structured Function description Language) developed by NTT. Then, a logic synthesis system analyzes the description and synthesizes a netlist, which describes the logic circuit structure. We use a system called PARTHENON [5], [6], which was also developed by NTT. In general, scheduling and allocation are difficult at high level synthesis procedure. In our system, users design the finite state machine in order to help synthesis. PARTHENON integrates a description language, simulator, and logic synthesizer, which synthesizes 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.



```
output = ( x<4> | ^x<3> |  x<2>)
       & ( x<4> |  x<3> |  x<1>)
       & (^x<4> |  x<2> | ^x<1>)
       & (^x<3> | ^x<2> |  x<1>);
```

Fig. 2. Example of implementation of exhaustive algorithm.

IV. Algorithms with Static Variable Ordering

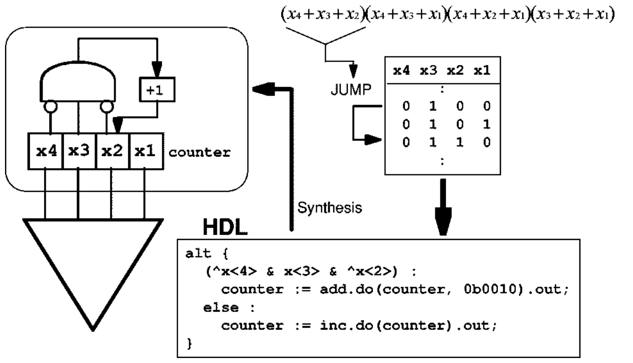
In this section, we present several algorithms with static variable ordering. In these algorithms, we check the complete assignment of variable values, i.e., we represent one combination of all variable values as an n-digit binary value. Assuming that variable x_is value is v_i, a combination of value assignments is represented by an n-digit binary value $\sum_{i=1}^{n} 2^{i-1}v_i$, in which the value of is digit (counted from the lowest digit) represents the value of x_i. We call the combination of all variable values a *state*.

A. Exhaustive Algorithm

The most straightforward algorithm with static variable ordering is an exhaustive algorithm as shown in Fig. 2. 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.

B. Backtracking Algorithm

When some clauses are not satisfied, instead of incrementing x_1s digit, we can increment the lowest digit that is included in these unsatisfied clauses; thus the number of searched states can be reduced (Fig. 3). More specifically, for each unsatisfied clause C_i, we choose the lowest digit x_i and increment max x_i. The algorithm obtained after this improvement is very similar to the backtracking algorithm [18] where the order of the variable selection is statically determined.



```
alt {
  (^x<4> & x<3> & ^x<2>) :
    counter := add.do(counter, 0b0010).out;
  else :
    counter := inc.do(counter).out;
}
```

Fig. 3. Example of implementation of backtracking algorithm.

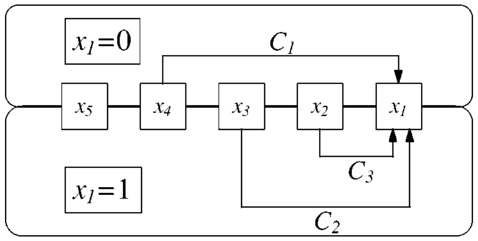


Fig. 4. Example of forward checking algorithm $(C_1 = (x_5+x_4+x_1), C_2 = (x_4 + x_3 + \overline{x_1}), C_3 = (x_5 + x_2 + \overline{x_1}))$.

C. Forward-Checking Algorithm

If we check not only the current value (0 or 1) but also 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 clauses $C_1 = (x_5 + x_4 + x_1)$, $C_2 = (x_4 + x_3 + \overline{x_1})$, $C_3 = (x_5 + x_2 + \overline{x_1})$. The initial state $(x_5, x_4, x_3, x_2, x_1) = (0, 0, 0, 0, 0)$ does not satisfy C_1. If we increment x_1s digit and change x_1s value to 1, then C_2 and C_3 are not satisfied. If we perform the check for the case that x_1s value is 1, we can confirm that incrementing x_1s digit is useless. In this case, 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_3s digit must be changed to satisfy all clauses and that changing digits lower than x_3 is useless (Fig. 4).

D. Unit Resolution

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 been assigned values. This procedure is called *unit resolution* [19].

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, and the second lowest digit in the clause that is constraining x_i is x_js digit. When there exist multiple possible values, unit$(x_i) = i$. We use this information to calculate the digit to increment.

For example, in the initial state $(x_5, x_4, x_3, x_2, x_1) = (0, 0, 0, 0, 0)$ of the problem described above, x_1 has only one consistent value 1 by C_1. Therefore, we set unit(x_1) to 4 (since

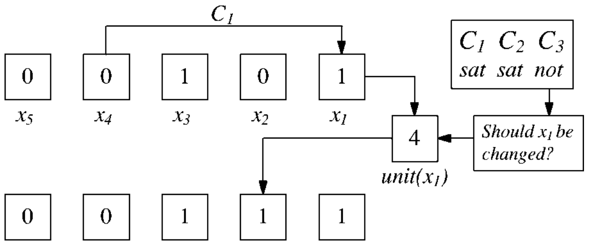


Fig. 5. Example of unit resolution.

x_4 is the second lowest digit in C_1) and change x_1s value to 1. The value of unit(x_1) represents the fact that unless at least x_4s digit is changed, x_1 has only one possible value. The next state will be $(0, 0, 1, 0, 1)$. This state does not satisfy C_3. Since the lowest digit in C_3 is x_1, x_1s value is changed in the original procedure. However, unit$(x_1) = 4$ and the second lowest digit in C_3 is x_2, where x_2 is lower than x_4. Therefore, x_2s value is changed. The next state is $(0, 0, 1, 1, 1)$. This state satisfies all of the three clauses (Fig. 5).

V. Algorithms with Dynamic Variable Ordering Heuristics

By introducing forward-checking and unit resolution, the performance of the obtained algorithm becomes equivalent to the basic Davis–Putnam procedure [19]. However, this algorithm turns out to be inefficient since the variable ordering is static (except for unit-resolution). To solve a large-scale problem within a reasonable amount of time, we need to introduce dynamic variable ordering heuristics.

As the number of variables increases, the time required to solve the problem grows exponentially. However, since the rate of growing decreases by using dynamic variable ordering, it is better for large-scale problems. This is well-known by studies of constraint satisfaction problems [20].

In this section, we first describe the outline of an algorithm with dynamic variable ordering heuristics, then explain how to implement such an algorithm on FPGAs, and finally describe two dynamic variable ordering heuristics (EUP heuristic and MOMs heuristic).

A. Outline of the Algorithm

The outline of the algorithm with a dynamic variable ordering heuristic 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, 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 that satisfies the clause, simplify the clauses of S and go to 2).

5) **Branching:** Select the variable according to some heuristics. 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).

Simplifying clauses means that if we set $x_1 = 1$, we discard all clauses that include x_1 [e.g., $(x_5+x_4+x_1)$], since these clauses are already satisfied, and for each clause that includes $\overline{x_1}$, we remove $\overline{x_1}$ from the clause [e.g., $(x_4 + x_3 + \overline{x_1})$ is simplified to $(x_4 + x_3)$].

B. Implementation of the Algorithm with Dynamic Variable Ordering

In most cases, such a backtracking tree search algorithm is implemented by 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 assigning separate registers for each variable. More specifically, there exists a register for each variable that 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.

We are going to describe the details of the algorithm. We first 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_is value is determined. If determined$(x_i) = 0$, x_is 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_is value is determined by unit resolutions. If branch$(x_i) = 1$, this means that x_is 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: determined$(x_i) = 1$ and x_is value is v_i, determined$(x_j) = 1$ and x_js value is v_j, or determined$(x_k) = 1$ and x_ks 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 one remaining 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. 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 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 best_pos using some variable ordering heuristic.
   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 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 depth(x_i) to current_depth-1.
   5.3 set current_depth to current_depth-1, go to 1).

C. Algorithm with EUP Heuristic

The branching heuristic called experimental unit propagation (EUP) was shown to be very effective [21]. In short, when selecting a variable to branch using this heuristic, we *experimentally* set each unassigned variable's value to 0 and 1 (experimental unit propagation procedure). We select the variable x that causes the maximum number of unit propagation.

One advantage of using this heuristic is that the logic circuit for implementing it is very similar to the logic circuit for the main tree search procedures; thus, we can share the hardware resources between these routines.

D. Algorithm with MOMs Heuristic

The other variable ordering heuristic used for the branching is called *Maximum Occurrences in clauses of Minimum Size (MOMs) heuristic*. In this heuristic, we simply count the occurrences of each variable in binary clauses and choose the variable that appears most in binary clauses. Although this heuristic is very simple, it has been reported to be very effective [22], [18] in improving the efficiency of the Davis–Putnam procedure. Compared with the EUP heuristic, performing this heuristic in software is easy, but implementing this heuristic on FPGAs requires more hardware resources than those needed for the EUP. We can consider that this heuristic performs a kind of approximation of the EUP heuristic, since if a variable occurs often in binary clauses, we can expect that assigning value to this variable will cause many unit-resolutions.

VI. Hardware Implementation

In this section, we explain how to implement the algorithm described in Section V-C on FPGAs, since this algorithm turns out to be the most efficient. The algorithm can be straightforwardly represented as a finite state machine.

Since there are many similarities between the main procedure and the EUP procedure, hardware can be shared between them. Hardware works exclusively, that is, at a given time hardware works in either the main procedure mode or in the EUP 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 mode, and if *eup* is 1, the algorithm is in the EUP procedure mode. The *evaluation, unit,* and *backtrack* states are used both in the main procedure mode and in the EUP procedure mode. However, the behavior of these states is slightly different according to the value of *eup*.

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*:

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

In the *backtracking* state, for each variable x_i, depth(x_i) and current_depth are compared, and the 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.

The high-level hardware description language SFL can handle the finite state machine representation, and the LSI CAD system PARTHENON can automatically generate an RT-level hardware description. The state transitions of the finite state machine are shown in Fig. 6.

Fig. 7 shows the hardware block diagram. In the `Variables` part, the value and parameters of each variable are stored. In the *evaluation* state, they are sent to the `condition check` part, and then the condition of each clause is checked. The connections between the `Variables` part and the `condition check` part represent clauses in which variables are included. The condition of each clause is checked concurrently. Fig. 8 is a part of the description of the `condition check` part written in HDL

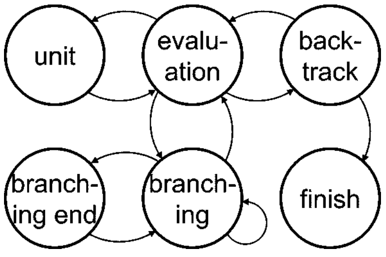


Fig. 6. Finite state machine transitions of experimental unit propagation.

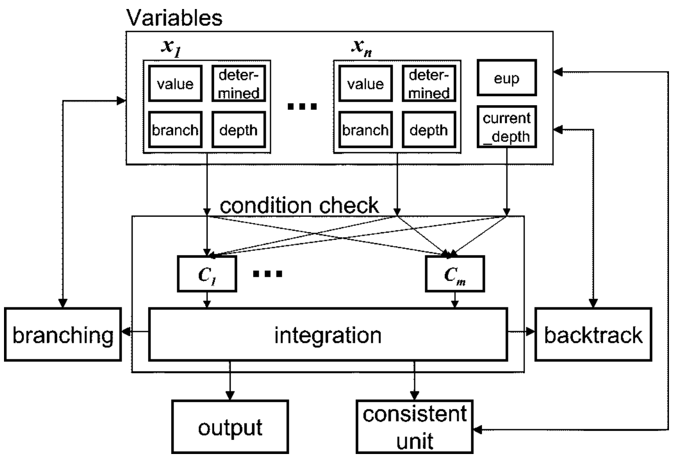


Fig. 7. Hardware block diagram for experimental unit propagation.

```
module clause {
  input   value<3>;
  input   condition<3>;
  output  sat, no_sat, unit, bin;
  instrin do;

  instruct do par {
    sat = (determined<0> & value<0>)
        | (determined<1> & value<1>)
        | (determined<2> & value<2>);

    no_sat = (determined<0> & ~value<0>)
           & (determined<1> & ~value<1>)
           & (determined<2> & ~value<2>);

    unit = ...
  }
}
```

Fig. 8. HDL description of the part for clause evaluation.

(SFL), where & is logical AND and | is logical OR. In this case, `sat` and `no_sat` are checked concurrently, and then the results are outputted.

The condition of each clause is sent to the `integration` part, and then the next step is determined by the integration of all conditions according to the algorithm. For example, if *not-satisfied* clause exists, the next step is *backtracking*.

In the `branching` part, the behavior shown in 4) of main procedure is done. In 4), clauses are evaluated by assigning the

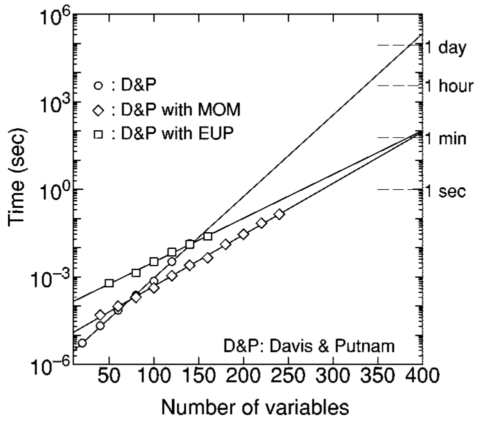


Fig. 9. Required time at 10 MHz on hard random 3-SAT problems.

values of variables experimentally. For this behavior, *eup* is set to 1, and variables values are assigned one-by-one. Then, the state is changed to the *evaluation* state, and the evaluation is done. In the *evaluation* state, the behavior is changed by the value of *eup*. After all variables are evaluated, *eup* is set to 0, the values are assigned as shown in the **branching end**, and then return to Main mode.

In the `backtracking` part, the *backtracking* is done. The `branching` part is shared by Main mode and EUP mode. After the mode is distinguished by the value of *eup*, its behavior is changed. In the `consistent unit` part, the behavior is done as shown in 3) of the main procedure. In the `output` part, the values are outputted as the solution in the case of 2) of the main procedure.

VII. Evaluation

A. Simulation

We first evaluated the efficiency of the developed algorithms by software simulation. We use hard random 3-SAT problems as examples. 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 problems [23].

In Fig. 9, we show the log-scale plot of the average required time of over 100 problems, assuming the clock rate is 10 MHz. If several chips are used to implement the problem, clock speed would be reduced by the connection between chips. However, the capacity of an FPGA chip is increasing very rapidly, so we assume that we can implement a problem instance of this size in the very near future. 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.

In Fig. 9, we show the results of the algorithm that introduces forward-checking and unit-resolution with static variable ordering (D&P), the algorithm that uses the MOMs heuristic (D&P with MOM), and the algorithm that uses the EUP heuristic for variable ordering (D&P with EUP). It is obvious that dynamic variable ordering is indispensable to solving large-scale problems in a reasonable amount of time. We can see that the D&P with EUP can solve a hard random 3-SAT problem with 400 variables within 1.6 min at a clock rate of 10 MHz. In addition, we can see that the search tree of EUP grows at the rate of $O(2^{n/20.0})$, where n is the number of variables, whereas the search tree of MOMs grows at the rate of $O(2^{n/17.3})$. This result shows that the D&P with EUP is more efficient with a larger number of variables.

Furthermore, we show the results for AIM benchmark problems [24] with 128 variables and 256 clauses on FPGAs. These problem instances are unsolvable and known to be very difficult. They can be actually implemented on an FPGA chip. Table I shows the required numbers of states and the times needed to solve these problems when the clock rate is 10.0 MHz. For comparison, we also show the cpu time of POSIT [22], a very sophisticated SAT solver that utilizes the MOMs heuristic. These programs run on a Sun Ultra 30 Model 300 (UltraSPARC-II 296 MHz). 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 1 h. The bottleneck routine is the FPGA Mapper of the FPGA system, which generates FPGA mapping data from the netlist. In addition, many factors affect change compile times, e.g., the depth of synthesis optimization, synthesis method for FPGA, target FPGA structure, FPGA Mapping tools, and machine power for synthesis and mapping. Accordingly, the compile time is easily changed. In this paper, we compiled the entire circuit for each problem. However, this routine can be highly optimized for SAT problems since many parts in a logic circuit are common in all problem instances. Therefore, compile time can be dramatically reduced, and the time required to generate a logic circuit can be negligible for larger-scale problems.

B. Current Implementation Status

We use an ALTERA FLEX10K250 FPGA chip to implement the algorithm. The FLEX10K250 has 12 160 Logic Cells (LCs) and its typical usable gates are from 149 k to 310 k. We have actually implemented a 3-SAT problem, “aim-128-2_0-no-*.cnf” as previously described with 11 042 LCs. The utilization rate of the LCs was 90%. We were able to run this circuit at a clock rate of 10.0 MHz. In addition, we have successfully mapped “aim-200-1_6-yes1-1.cnf” with 200 variables and 320 clauses, but the circuit is divided into 21 FLEX10 K chips. Many chips are used because the circuit requires a lot of wiring resources. In this case, since the number of pins is restricted, wiring resources are insufficient. As a result, the total utilization rate of the LCs was 13%. This situation will be improved by increasing the gates of a chip and by a dynamic reconfiguration technique.

Fig. 10 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 further reduced while keeping the same trend. EUPs required number of gates is approximately 30% less than that of MOM. This is

TABLE I
REQUIRED STATE AND TIME FOR BENCHMARK PROBLEMS

problem	#v	#C	# of states	time (s)	POSIT (s)	ratio
aim-128-2_0-no-0	128	256	1,973,080,127	197.3	884.1	4.48
aim-128-2_0-no-1	128	256	1,662,111,245	166.2	931.6	5.61
aim-128-2_0-no-2	128	256	44,321,403	4.43	230.8	52.1
aim-128-2_0-no-3	128	256	3,651,710,074	365.2	2138.1	5.85
aim-128-2_0-no-4	128	256	2,726,316,935	272.6	2749.3	10.08
aim-128-2_0-no-5	128	256	839,166,228	83.9	471.6	5.62
dubois20	60	160	206,568,894	20.7	25.3	1.22
dubois21	63	168	425,721,254	42.6	81.7	1.92

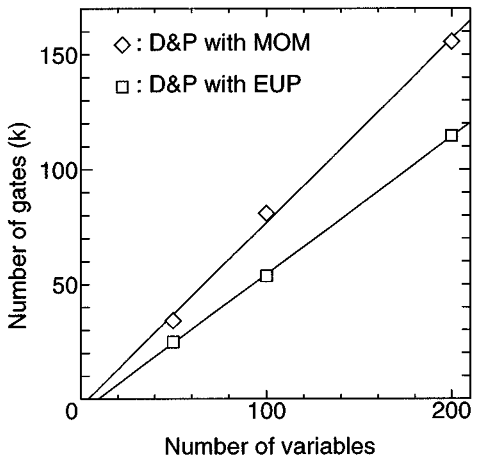


Fig. 10. Required gates for "aim-⟨#variables⟩-1_6-yes1-1.cnf."

because almost all of the circuits of the EUP can be shared with the main procedure. On the other hand, the MOM algorithm requires additional circuits for branching.

VIII. Related Work

The authors presented the first result of this approach in [1]. After this report, various research projects following this line have been carried out [9]–[15].

One of the major differences between the other approaches and ours in this paper is the method used for variable ordering. While the other approaches solve a problem instance using static variable ordering, our approach uses dynamic variable ordering. Dynamic variable ordering is indispensable to solve large-scale problems, as described in Section V.

In [10], a hill-climbing algorithm is implemented on FPGAs. Although hill-climbing algorithms are very efficient and can solve very large-scale problems, the completeness of the algorithm, i.e., always finding a solution if one exists or terminating if no solution exists, cannot be guaranteed.

In [9], a PODEM-based algorithm [25] is implemented on FPGAs. This algorithm is more suitable for testing multi-level logic circuits than for solving generic SAT problem instances. In [11], a similar method to [9] is used, but this method tries to optimize the total time, including the time for generating the logic circuit.

In [14], a massively parallel fine-grain satisfier architecture is introduced. This architecture accelerate to implement a SAT solver on reconfigurable hardware.

In [12] and [13], a tree search algorithm based on the Davis–Putnam procedure is implemented on FPGAs. A specialized logic circuit is used to perform a more powerful resolution method than unit-resolution. This resolution method seems very effective for solving problem instances that are particularly difficult in software implementation. However, in this method, the variable ordering for branching is statically determined.

IX. Conclusion

This paper presented results on solving SAT problems using FPGAs. In this approach, a logic circuit specific to each problem instance is created on FPGAs. We developed a series of algorithms that are suitable for implementation on a logic circuit.

Simulation results showed that the best method, which utilizes the EUP, can solve a hard random 3-SAT problem with 400 variables within 1.6 min at a clock rate of 10 MHz. Furthermore, we have actually implemented a benchmark problem with 128 variables that can run at 10 MHz.

We are currently refining the implementation of the algorithm on FPGAs and performing various evaluations on implemented logic circuits.

Acknowledgment

The authors would like to thank Dr. Y. Asahiro and Prof. K. Iwama for providing instances of SAT benchmark problem.

References

[1] M. Yokoo, T. Suyama, and H. Sawada, "Solving satisfiability problems using field programmable gate arrays: First results," in *Proc. 2nd Int. Conf. Principles Practice Constraint Programming (CP'96)*, 1996, pp. 497–509.
[2] T. Suyama, M. Yokoo, and H. Sawada, "Solving satisfiability problems using logic synthesis and reconfigurable hardware," in *Proc. 31st Annual Hawaii Int. Conf. System Sciences*, vol. VII, 1998, pp. 179–186.
[3] T. Suyama, M. Yokoo, and A. Nagoya, "Solving satisfiability problems on FPGAs using experimental unit propagation," in *Proc. 5th Int. Conf. Principles Practice Constraint Programming (CP'99)*: Springer-Verlag, 1999, Lecture Notes in Computer Science 1713, pp. 434–445.
[4] S. D. Brown, R. J. Francis, J. Rose, and Z. G. Vranesic, *Field-Programmable Gate Arrays*: Kluwer, 1992.
[5] R. Camposano and W. Wolf, *High-Level VLSI Synthesis*: Kluwer, 1991.
[6] Y. Nakamura, K. Oguri, A. Nagoya, M. Yukishita, and R. Nomura, "High-level synthesis design at NTT systems labs," *IEICE Trans. Inf. Syst.*, vol. E76-D, no. 9, pp. 1047–1054, 1993.
[7] A. K. Mackworth, *Encyclopedia of Artificial Intelligence*: Wiley-Interscience, 1992.
[8] S. Cook, "The complexity of theorem proving procedures," in *Proc. 3rd Annual ACM Symp. Theory Computation*, 1971, pp. 151–158.
[9] M. Abramovici and D. Saab, "Satisfiability on reconfigurable hardware," in *Int. Workshop Field Programmable Logic Applications*, 1997, pp. 448–456.

[10] Y. Hamadi and D. Merceron, "Reconfigurable architectures: A new vision for optimizing problem," in *Proc. 3rd Int. Conf. Principles Practice Constraint Programming (CP'97)*, 1997, pp. 209–221.

[11] A. Rashid, J. Leonard, and W. H. Mangione-Smith, "Dynamic circuit generation for solving specific problem instances of Boolean satisfiability," in *Proc. IEEE Symp Field-Programmable Custom Computing Machines*, 1998, pp. 196–204.

[12] P. Zhong, M. Martonosi, P. Ashar, and S. Malik, "Accelerating Boolean satisfiability with configurable hardware," in *Proc. Symp. Field-Programmable Custom Computing Machines*, 1998, pp. 186–195.

[13] ——, "Solving Boolean satisfiability with dynamic hardware configurations," in *Int. Workshop Field Programmable Logic Applications*, 1998, pp. 326–335.

[14] M. Abramovici, J. T. de Sousa, and D. Saab, "A massively-parallel easily-scalable satisfiability solver using reconfigurable hardware," in *Proc. Design Automation Conf.*, 1999.

[15] M. Boyd and T. Larrabee, "A scalable, loadable custom programmable logic device for solving Boolean satisfiability problems," in *Proc. IEEE Symp. Field-Programmable Custom Computing Machines*, 2000.

[16] R. Reiter and A. Mackworth, "A logical framework for depiction and image interpretation," *Artificial Intelligence*, vol. 41, no. 2, pp. 125–155, 1989.

[17] R. Reiter, "A theory of diagnosis from first principles," *Artificial Intelligence*, vol. 32, no. 1, pp. 57–95, 1987.

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

[19] M. Davis and H. Putnam, "A computing procedure for quantification theory," *J. ACM*, vol. 7, pp. 201–215, 1960.

[20] R. M. Haralick and G. L. Elliot, "Increasing tree search efficiency for constraint satisfaction problems," *Artificial Intelligence*, vol. 14, pp. 263–313, 1980.

[21] C. M. Li and X. Anbulagan, "Heuristics based on unit propagation for satisfiability problems," in *Proc. 15th Int. Joint Conf. Artificial Intelligence*, 1997, pp. 366–371.

[22] J. W. Freeman, "Improvements to Propositional Satisfiability Search Algorithms," Ph.D. dissertation, Univ. Pennsylvania, 1995.

[23] D. Mitchell, B. Selman, and H. Levesque, "Hard and easy distributions of SAT problem," in *Proc. 10th National Conf. Artificial Intelligence*, 1992, pp. 459–465.

[24] Y. Asahiro, K. Iwama, and E. Miyano, "Random generation of test instances with controlled attributes," in *Proc. DIMACS Challenge II Workshop*, 1993.

[25] P. Goel, "An implicit enumeration algorithm to generate tests generation algorithms," *IEEE Trans. Computers*, vol. C-30, pp. 215–222, 1981.

Takayuki Suyama received the B.E. and M.E. degrees in mechanical engineering from Osaka University, in 1990 and 1992, respectively.

He joined the NTT Communication Science Laboratories in 1992. He is currently with the Research and Development Center, NTT West, Kyoto, Japan. His research interests include computer architecture design, FPGA-related systems, and applications of reconfigurable computing.

Mr. Suyama is a member of IPSJ.

Makoto Yokoo received the B.E., M.E., and Ph.D. degrees from the University of Tokyo, Japan, in 1984, 1986, and 1995, respectively.

He is currently a distinguished technical member in the NTT Communication Science Laboratories, Kyoto, Japan. He was a Visiting Research Scientist at the Department of Electrical Engineering and Computer Science, the University of Michigan, Ann Arbor, from 1990 to 1991. His research interests include multi-agent systems, constraint satisfaction, and mechanism design among self-interested agents. His research on constraint satisfaction among multiple agents is presented in his book *Distributed Constraint Satisfaction: Foundation of Cooperation in Multi-Agent Systems* (Springer, 2000).

Dr. Yokoo is an editorial board member of international journals, *Constraints*, *Autonomous Agents and Multi-Agent Systems*, and *Journal of Artificial Intelligence Research*.

Hiroshi Sawada was born in Osaka, Japan, on October 31, 1968. He received the B.E. and M.E. degrees in information science from Kyoto University, Kyoto, Japan, in 1991 and 1993, respectively.

In 1993, he joined NTT Communication Science Laboratories, where he has been engaged in research of computer-aided design of digital systems and computer architecture. His areas of research interest include logic synthesis, BDD techniques, optimization problems, and VLSI design.

Mr. Sawada is a member of IEICE and IPSJ.

Akira Nagoya (M'99) received the B.E. and M.E. degrees in electronic engineering from Kyoto University in 1978 and 1980, respectively.

He joined the NTT Electrical Communication Laboratories in 1980 where he is now a Research Group Leader at the Network Innovation Laboratories. From 1990 to 1991, he was with the Department of Computer Science, University of Illinois at Urbana-Champaign, as a Visiting Scholar. His areas of research interest include reconfigurable computing, computer architecture, VLSI design, and electronic design automation.

Mr. Nagoya is a member of IEICE and IPSJ. He received the Okochi Memorial Technology Prize in 1992 and the Achievement Award from IEICE in 2000.