CHAPTER 29

BOOLEAN SATISFIABILITY: CREATING SOLVERS OPTIMIZED FOR SPECIFIC PROBLEM INSTANCES

Peixin Zhong
Department of Electrical and Computer Engineering
Michigan State University
Margaret Martonosi, Sharad Malik
Department of Electrical Engineering
Princeton University

Boolean satisfiability (SAT) is a classic NP-complete problem with a broad range of applications. There have been many projects that use reconfigurable computing to solve it. This chapter presents a review of the subject with emphasis on a particular approach that employs a backtrack search algorithm and generates solver hardware according to the problem instance. This approach utilizes the reconfigurability and fine-grained parallelism provided by FPGAs.

The chapter is organized as follows: Section 29.1 is an introduction to the SAT formulation and applications. Section 29.2 describes the algorithms to solve the SAT problem. Sections 29.3 and 29.4 describe in detail two SAT solvers that use reconfigurable computing, and Section 29.5 provides a broader discussion.

29.1 BOOLEAN SATISFIABILITY BASICS

The Boolean satisfiability problem is well known in computer science [1]. Given a Boolean formula, the goal is to find an assignment to the variables so that the formula evaluates to true or 1 (it satisfies the formula), or to prove that such an assignment does not exist (the formula is not satisfiable). It has many applications, including theorem proving [5], automatic test pattern generation [2], and formal verification [3,4].

29.1.1 Problem Formulation

The Boolean formula in an SAT problem is typically represented in conjunctive normal form (CNF), also known as product-of-sums. Each sum of literals is called a clause. A literal is either a variable or the negation of a variable, denoted with a negation symbol or a bar (such as $\neg v_1$ or $\bar{v}_1$). Equations 29.1 and 29.2 are examples of simple CNFs.

$$(v_1+v_2+v_3)(\bar{v}_1+v_2+v_3)(v_1+v_2+\bar{v}_3)(\bar{v}_2+\bar{v}_3) \tag{29.1}$$

or

$$(v_1 \vee v_2 \vee v_3) \wedge (\neg v_1 \vee v_2 \vee v_3) \wedge (v_1 \vee v_2 \vee \neg v_3) \wedge (\neg v_2 \vee \neg v_3) \quad (29.2)$$

Each sum term, such as $(v_1 + v_2 + v_3)$, is a clause. In the clause, v_1 or $\neg v_1$ is called a literal. It can be easily tested that $v_1 = 1, v_2 = 1, v_3 = 0$ is a solution to the problem.

The SAT clauses represent implication relationships between variables. To satisfy the CNF, each clause should be satisfied (i.e., at least one literal in each clause should be 1). For a given partial assignment, if only one literal in a clause is not assigned but all others are assigned to 0, the unassigned literal is implied to be 1 to satisfy the clause. The first clause in equation 29.1 contains three possible implications. If $v_1 = 0$ and $v_2 = 0, v_3$ is implied to be 1, denoted as $\neg v_1 \neg v_2 \supset v_3$. Similarly, $v_1 = 0$ and $v_3 = 0$ imply $v_2 = 1$, and $v_2 = 0$ and $v_3 = 0$ imply $v_1 = 1$. Such implications can be used to construct powerful logic expressions. They are also the key to SAT-solving algorithms.

29.1.2 SAT Applications

The many applications of SAT include test pattern generation [2] and model checking [3, 4]. The logic relations of a digital circuit can also be represented in SAT CNF. Each logic gate is represented by a group of clauses, with each signal represented by a variable with two possible values, 1 or 0. A circuit is represented by a conjunction of clauses representing all gates in the circuit. What follows is the transformation from simple gates to clauses:

AND gate, $z <= ab$, maps to $(a + \neg z)(b + \neg z)(\neg a + \neg b + z)$
NAND gate, $z <= \neg(ab)$, maps to $(a + z)(b + z)(\neg a + \neg b + \neg z)$
OR gate, $z <= a + b$, maps to $(\neg a + z)(\neg b + z)(a + b + \neg z)$
NOR gate, $z <= \neg(a + b)$, maps to $(\neg a + \neg z)(\neg b + \neg z)(a + b + z)$
XOR gate, $z <= a \oplus b$, maps to $(\neg a + \neg b + \neg z)(\neg a + b + z)(a + \neg b + z)(a + b + \neg z)$
Buffer gate, $z <= a$, maps to $(\neg a + z)(a + \neg z)$
Inverter gate, $z <= \neg a$, maps to $(a + z)(\neg a + \neg z)$

SAT can be used in test pattern generation or to verify the equivalence of two combinational circuits. The circuit construction is shown in Figure 29.1. In equivalence checking, the two representations of the circuit are fed with the same primary inputs signals, and the corresponding primary outputs feed into an exclusive-or (XOR) gate. If an assignment of primary inputs can be found such that any of the XOR gates has 1 as an output, the circuits are different. If no such assignment can be found, the circuits are functionally identical.

For test pattern generation, instead of using two representations of one circuit, we use two copies of the same circuit. However, one copy has a fault introduced into the design, which we can detect by searching for some pattern of inputs. In this case any input pattern that can generate a 1 on an XOR output is a test for that fault. If no such assignment is possible, that fault is untestable.

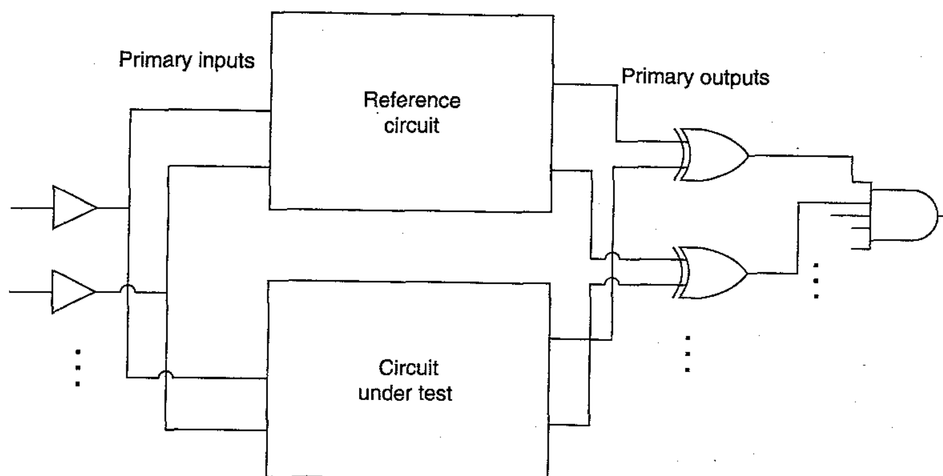

FIGURE 29.1 ■ Test pattern generation.

29.2 SAT-SOLVING ALGORITHMS

29.2.1 Basic Backtrack Algorithm

There are many algorithms to solve the SAT problem. They can be divided into two categories: complete and incomplete. A complete algorithm guarantees either to find a solution on termination or to prove that there is no solution. Complete algorithms typically employ a methodical search of the variable assignment space. For hard problems, the runtime may well exceed acceptable levels. An incomplete algorithm does not guarantee to find the solution and typically involves greedy or randomized search [22]. It can often find a solution of an easy problem very quickly, but if it fails to do so within a given time, it does not prove that no solution exists. Many applications require a complete algorithm to provide a definite answer, so this chapter concentrates on such algorithms for SAT.

An early SAT algorithm was proposed by Davis and Putnam [5]. Like theirs, most complete SAT algorithms are based on backtrack search [6–9], which is similar to depth-first search in traversing a tree. The pseudo-code of the basic algorithm, shown in Figure 29.2, starts with an empty variable partial assignment (i.e., every variable value is assumed to be unknown, or free). The search level is increased by branching—that is, assigning a value for a free variable. The algorithm checks if the incremented partial assignment can be part of a solution. If not, we say a conflict is detected. If there is no conflict, the algorithm will choose another free variable and branch on it; if a conflict is detected, it will backtrack to the most recently assigned variable and choose the opposite value. All decisions made after that backtrack point will be undone.

```
Solve_SAT()
{
  assign all variables to unknown;
  while (true) {
    if (implications force an unknown variable to a specific value)
      set that variable to that specific value;
    if (the current assignment has a conflict) {
      undo all implications and branches up to most recent untoggled branch;
      if (all branches undone)
        return No_solution;
      toggle value assigned to the variable of last untoggled branch;
    }
    if (no unassigned variables remain)
      return Solved;
    } else {
      start new branch by assigning a value to the next free variable;
    }
  }
}
```

FIGURE 29.2 ■ The basic backtrack algorithm to solve SAT.

The algorithm has two possible terminating conditions. If all variables values are known and the formula is satisfied, a solution is found. If all branches fail to find a solution and the algorithm must backtrack beyond the first branch variable, there is no solution and the formula is unsatisfiable.

The key to the efficiency of the backtrack algorithm is effectively pruning the search space. Early detection of a conflict assignment avoids useless searches along this branch. The following are some basic rules and techniques used in the algorithm. At each stage of the search, a variable can have one of three possible values: 1, 0, and free (unassigned).

1. If at least one literal of a clause evaluates to 1, this clause is satisfied. There is no need to check other literals in the clause.
2. If all literals of a clause evaluate to 0, the partial assignment is a conflict and cannot be part of the solution.
3. If only one literal of a clause is free and all other literals evaluate to 0, the free literal is implied to be 1. This is called unit resolution or implication. Implication is a powerful mechanism because it can deduce implied values of variables not yet branched on. However, it can create another case of conflict if a variable is implied by two clauses to be of opposite values.
4. If all of the literals of a free variable in the as yet unsatisfied clauses are all of the same polarity (i.e., inverted or not inverted), a value can be chosen for this variable that safely satisfies these clauses.
5. Because the variable ordering of branches has a large impact on the efficiency of the algorithm, different dynamic or static ordering schemes have been investigated. A simple heuristic orders the variables based on the number of clauses they appear in. A variable with the most appearances

often has more influence than others. Therefore, branching on it early typically prunes the search space more quickly.

A basic algorithm can use a static variable ordering. It can also use a fixed branching scheme, such as always branching with value 1, in which, after each branch or backtrack, implication is checked exhaustively. This basic algorithm corresponds to the reconfigurable SAT solver described in Section 29.2.

29.2.2 Improving the Backtrack Algorithm

Among the advanced features explored to further improve the efficiency of the backtrack search algorithm [6, 7], an effective one is learning based on conflict analysis. With the search algorithm moving back and forth by branching and backtracking, similar spaces are explored many times. Consider a problem, as in equation 29.3, where some of the clauses are

$$(\neg v_i + v_j + v_k)(\neg v_i + v_j + \neg v_k)(\neg v_i + \neg v_j + v_k)(\neg v_i + \neg v_j + \neg v_k)\ldots\ldots \quad (29.3)$$

The variable v_i is branched to be 1, and many other variables may have been tested before v_j is branched on. When v_j is branched on and 1 is tested, a conflict on v_k is detected. Then v_j is switched to 0, which again causes a conflict. Thus, the algorithm will backtrack to the previous branch variable. However, switching variable assignments other than v_i will not help. The algorithm may reenter the same region many times before it backtracks to v_i. Conflict analysis would be helpful in this situation.

A new variable value is implied by the value choices of all other literals in this clause being 0. Each literal has obtained its value either from branch decisions or from earlier implications. Therefore, we can create a transitive implication graph where an implied variable is ultimately implied by a set of branch decisions. A conflict is detected when a variable is implied to be of opposite values. It can be identified by backtracking the implication graph to identify the complete set of branch assignments that led to it. This set of decisions is responsible for the conflict.

In the example just given, the first conflict is caused by $v_i = 1$ and $v_j = 1$. A new clause can be derived as $(\neg v_i + \neg v_j)$. This is a redundant clause that can be added to the formula without changing the solution. It can also be viewed as applying the following consensus theorem to clauses 3 and 4 in equation 29.4:

$$(x + y)(\neg x + z) = (x + y)(\neg x + z)(y + z) \quad (29.4)$$

With the conflict on $v_j = 1$ detected, it can be interpreted as v_j is implied to be 0. In this case, it is implied by $v_i = 1$. Another round of implication will render a conflict because of the first two clauses in the original formula. From the second conflict, a new clause can be derived as $(\neg v_i + v_j)$. Combined with the conflict analysis result of the previous conflict, the resulting clause is $(\neg v_i)$, which dictates $v_i = 0$.

The algorithm should instead directly backtrack to v_i, in what is called nonchronological backtracking by Marques-Silva and Sakallah [6]. The new clause can be added to the problem and thus help prune the future search space.

This example is extremely simple, but the principle is applicable to all conflicts and can reduce runtime by several orders of magnitude on many problems. For example, for the AIM200 group of problems, GRASP takes 10.8 seconds, whereas many other SAT solvers take more than 10,000 seconds. However, because of the heuristic nature of the algorithms, they show different performance characteristics with different problems.

Learning also has its trade-offs. Every conflict will generate one redundant clause, and storage will explode if every such clause is recorded permanently. Heuristics for discarding long or unused redundant clauses can keep the storage size manageable and still achieve significant speedup.

29.3 A RECONFIGURABLE SAT SOLVER GENERATED ACCORDING TO AN SAT INSTANCE

This section presents an example of generating an SAT solver according to the SAT instance [10–12]. That is, instead of creating a generic, hardware SAT solver, we generate a new configuration for the reconfigurable computing machine for each SAT equation being solved.

29.3.1 Problem Analysis

A hard SAT problem can take a very long time to solve, limiting the application of the formula and the solvers' powerful formalism. Therefore, we will look at the use of reconfigurable computing techniques to accelerate SAT solutions. For this it is necessary to compare the relative merit of FPGAs and CPUs and look at the characteristics of SAT algorithms to identify an efficient solution.

FPGAs allow the full customization of control and datapaths. In particular, they make it efficient to perform bit-level operations. Also, by allocating more computing resources for bottleneck operations, they can provide massive parallelism and deep pipelining for suitable applications. However, FPGA clock rates are lower than those for microprocessors of the same technology generation, so raw chip performance may suffer.

Two opportunities for parallel processing in the SAT algorithm stand out, one of which is the parallelism in the vast search space. For a problem with n variables, there are 2^n possible assignments (though with the backtrack algorithm pruning the search space, that number is actually much smaller). It is possible to split at the branch choices and allocate each subspace to its own processor. However, because the search space is typically unbalanced, such parallelization requires rebalancing the load and this would be very complex to implement in hardware. Another source of potential performance gain is implication and conflict checking. Whenever a new value is assigned to a variable, all clauses

containing the variable should be checked for implication and conflict. New implied values will trigger further checking and implication. Additionally, the variables are Boolean and suitable for low-level processing by logic circuits, and thus implication and conflict checking are good candidates for hardware acceleration. It has also been confirmed through software profiling that implication and conflict checking take up the majority of computing time.

The basic backtrack search includes branch, implication, and backtrack functions, which are relatively simple and can be implemented with finite-state machines. Many projects implement a full SAT solver on one or multiple FPGAs. The next section describes one of them.

29.3.2 Implementing a Basic Backtrack Algorithm with Reconfigurable Hardware

Since implication and conflict checking are time-consuming processes, they are good candidates for hardware acceleration. Checking all clauses in parallel is one approach enabled by reconfigurable computing techniques. The circuit used for such parallel checking is presented as follows.

During the search, a variable can take one of three possible values: unknown, 1 (true), and 0 (false). A 2-bit encoding, denoted $(v, \bar{v})$, is used for the three variable values because it can conveniently represent them: (0, 0) is an unknown (free) variable; (1, 0) is value 1; and (0, 1) is value 0. The fourth combination, (1, 1), is used for conflict. The 2-bit encoding can be easily used for implication as well. For example, a clause with three literals $(v_i + \neg v_j + v_k)$ represents three possible implications that can be expressed with the 2-bit encoding as logical assignments, as shown in equation 29.5:

$$\begin{aligned} v_i &<= v_j \bar{v}_k \\ \bar{v}_j &<= \bar{v}_i \bar{v}_k \\ v_k &<= \bar{v}_i v_j \end{aligned} \tag{29.5}$$

When a literal appears in multiple clauses, its value is 1 if any one of the clauses implies it to be 1. The general form can be written as

$$\begin{aligned} v_{inew} &<= \sum_{\substack{\text{each clause } v_i \\ \text{appears in}}} \Big(\prod_{\substack{\text{each uninverted} \\ \text{literal } v_k}} \bar{v}_k \prod_{\substack{\text{each inverted} \\ \text{literal } \neg v_l}} v_l \Big) \\ \bar{v}_{inew} &<= \sum_{\substack{\text{each clause } \bar{v}_i \\ \text{appears in}}} \Big(\prod_{\substack{\text{each uninverted} \\ \text{literal } v_k}} \bar{v}_k \prod_{\substack{\text{each inverted} \\ \text{literal } \neg v_l}} v_l \Big) \end{aligned}$$

The summation $\sum$ is a logic OR over the set of clauses in which the implied literal appears. The production $\prod$ is a logic AND over all other literals in the clause. Note that the literal in the formula is inverted from the one in the clause, meaning that the implication is effective if and only if all other literals are known to be 0. With this formula, a complete CNF can be converted to circuits that evaluate all possible implications in parallel.

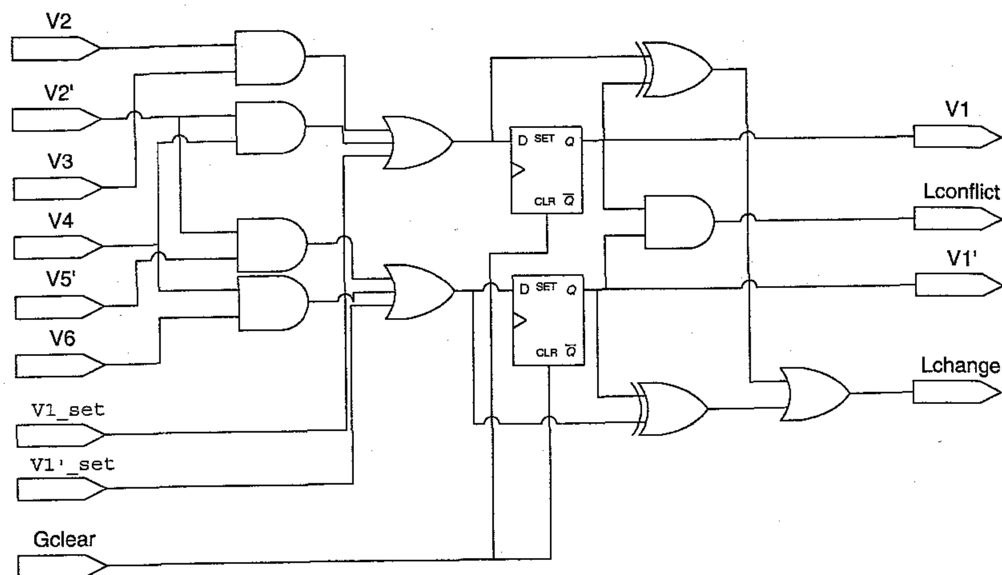

FIGURE 29.3 ▪ The implication circuit for one variable, V1.

The implication circuit for V1, shown in Figure 29.3, corresponds to the partial CNF of $(v_1+\neg v_2+\neg v_3)(v_1+v_2+\neg v_4)(\neg v_1+v_2+v_5)(\neg v_1+\neg v_4+\neg v_6)$, and is directly derived from the implication equation. A variable may assume a value because of either a branch decision or implication. An OR gate adds the assigned value. Since a newly implied variable may take part in generating new implications, registering the newly implied values allows implication to propagate one level in each clock cycle and avoids combinational cycles. To determine when implications have settled, an XOR gate checks the difference between the current and the next value. An AND gate checks if both literals of a variable are assigned to 1. If such a situation exists, the conflict (also called contradiction) signal is raised.

The other part of the algorithm is the control for the backtrack search. A distributed control architecture is used, with each finite-state machine (FSM) controlling one variable. Using a predetermined variable ordering, the architecture can be implemented by a linear array of communicating FSMs, as shown in Figure 29.4. Other than a few global signals, each FSM communicates only with the two neighboring FSMs. During the SAT-solving process, only one variable is active in terms of branching and backtracking. Its active status is represented by an active token. Two wires connect each pair of FSMs to pass the active token back and forth. Only one variable is the owner of the token at any given time.

In addition to the basic clock and reset signals, there are three global control signals. `Gconflict` is asserted when a conflict is detected. It is the wide OR function of all local conflicts, `Lconflict`. A local conflict is asserted when both

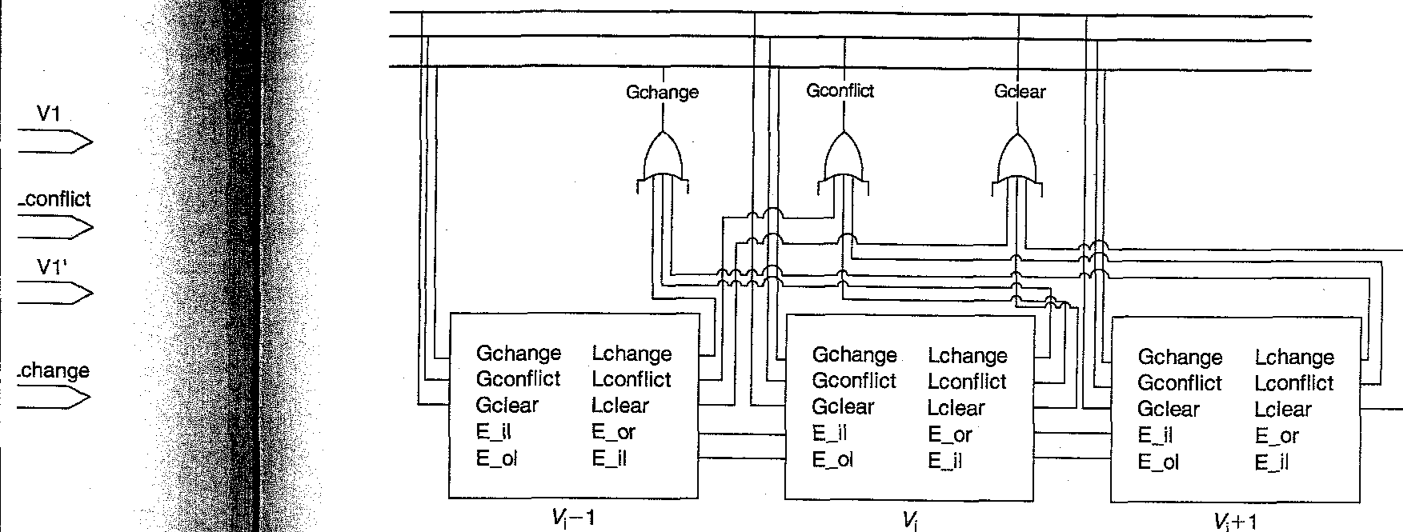

FIGURE 29.4 ■ The global topology for a basic SAT solver circuit.

v_i and $\bar{v}_i$ are assigned or implied to be 1. `Gchange` is asserted when any variable has changed value. It is the wide OR function of all local changes, `Lchange`. A local change is asserted when v_{inew} is different from v_i or when $\bar{v}_{inew}$ is different from $\bar{v}_i$. `Gclear` tells each state machine to clear the implied values. It is issued when the algorithm needs to backtrack and erase earlier implications.

With the external interface defined, each FSM should hold the assigned value, the implied value, and its state of backtrack search. The state machine is designed as registers for the implied value and an FSM combining the assigned value and state in the backtrack search. The state diagram of the latter FSM, shown in Figure 29.5, contains five states:

- *Idle:* This is the initial state, in which the internal variable value is (0, 0). The FSM will stay in the idle state unless it has received the active token from its neighbor through branching or backtracking. When the token is received, if this variable already has an implied value, there is no need to branch, and the FSM will simply pass the token to the next variable at the next clock. If this variable has no implied value and the token has been passed from the left, it will branch and choose the branch value as 1 (the active 1 state).
- *Active 1:* This state is the result of branching from the idle state, in which the variable value is chosen to be 1. The new value will be available for implication and conflict checking. The FSM will keep the token until there is no more change or until a conflict is detected. In the case of no conflict, it will pass the token to the right and will transition to the

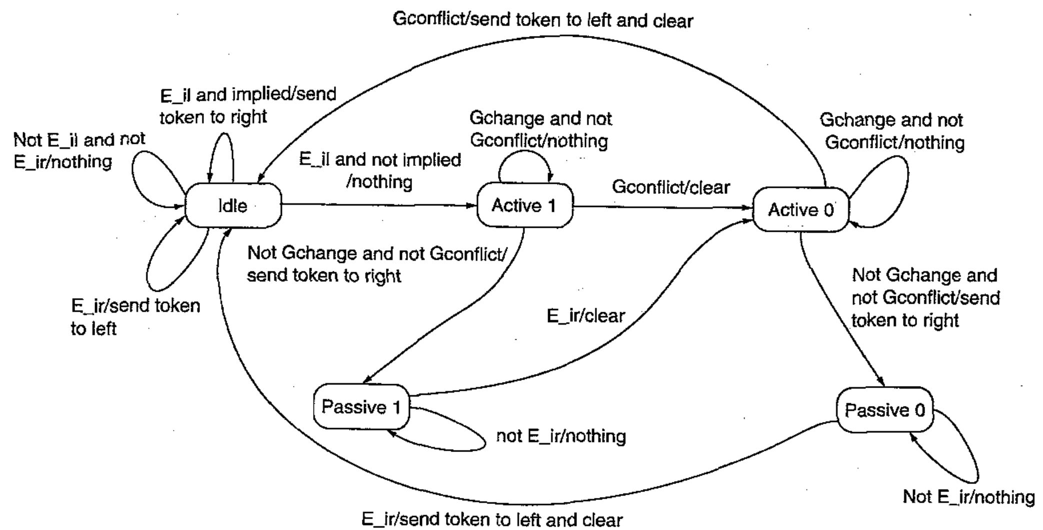

FIGURE 29.5 ■ The FSM associated with one variable.

passive 1 state. If a conflict is detected, it will transition to the active 0 state and restart the implication and conflict checking.

- *Active 0:* This state is the result of a conflict in the active 1 state or of the token being passed to passive 1 by backtracking. The variable value is set to 0. Implication and conflict are checked. If there is no conflict, the FSM passes the token to the right and transitions to passive 0. If there is a conflict, it will transition to the idle state and pass the token to the left.
- *Passive 1:* This state is the result of branching further from active 1. If the FSM receives a token from the right because of backtracking, it will transition to active 0.
- *Passive 0:* This state is the result of branching further from active 0. If the FSM receives a token from the right because of backtracking, it will transition to idle and pass the token to the left.

With these FSMs logically forming a linear chain, the branching of the algorithm corresponds to passing the token to the right and performing implications during the process. When a conflict is detected, backtracking is needed. Backtrack switches a value from 1 to 0. If it is already 0, the token is passed to the left. Whenever a conflict is detected, all of the implied values are cleared by the global clear signal and reset to free. The termination condition is easy to test: If the token is passed to the left of the first variable, the problem is unsatisfiable; if it is passed to the right of the last variable, a solution has been found. In addition to the regular problem-solving mode, the linear chain of variables can also be configured as a shift register. When a solution is found, it can be shifted out as a bitstream.

At the time of the design of this SAT solver (1997–1998), a single FPGA chip provided a very limited number of logic gates, and so for typical problems a multi-FPGA solution was needed. The algorithm was implemented on an IKOS (now part of Mentor Graphics) VirtualLogic SLI Emulator, which contained one to six FPGA boards, each containing 64 Xilinx XC4013E FPGA chips to form an 8×8 mesh. Thus, it provided the logic capacity to handle a midsize to large SAT problem. While the FPGA itself could support a clock rate of about 20 MHz, the Ikos system used a time-multiplexing I/O scheme called VirtualWire to overcome the pin limitation (see Section 6.4). Thus, the system clock rate was reduced to the 1-MHz range. An HP logic analyzer/function generator was connected to provide the initial input signal and collect the result.

To provide perspective, in 1992 the mainstream FPGA XC4013E had 1368 logic cells. In 2006, the large XC4VLX200 FPGA had 200,448 logic cells (i.e., about 146 times the logic capacity), which was more than what two big Ikos boards could provide.

To solve an SAT problem on this platform, the following steps are needed:

1. *Generate VHDL.* A software tool written in C++ reads in the problem CNF file and generates the VHDL code that models the SAT solver circuit. The FSM is manually coded in VHDL and reused for each SAT problem.
2. *Compile the FPGA.* The VHDL is compiled to bitstream files for programming the FPGAs. For a single FPGA implementation, this can be done by the FPGA tools. For the Ikos emulator, in contrast, this process takes three steps: (1) the design is synthesized into a netlist and partitioned to multiple FPGAs by the IKOS tool; (2) the partitioned netlist is generated; and (3) the netlist is compiled by Xilinx tools into bitstream files. The main function of the Xilinx tools is placement and routing.
3. *Configure the FPGA.* The bitstream is downloaded to the FPGA board, and the FPGA is configured with these files.
4. *Run the problem solver in the FPGA and load the result.* The logic analyzer/function generator creates the initial signals to start the computation. When the problem is solved, the solution is shifted out, where it can be captured by a logic analyzer.

The runtime performance of the FPGA SAT accelerator is shown in Figure 29.6 as a histogram of speedup ratios. This test was carried out in 1998 using the problem set from the DIMACS SAT challenge benchmark. The software runtime basis was obtained by running GRASP with parameter settings close to those of the basic backtrack algorithm. GRASP was run on a Sun 5 workstation with a 110-MHz processor and 64 MB of RAM. The hardware performance was normalized to a 1.33-MHz system clock rate, which is representative of implementations on the IKOS emulator. In the figure, the x-axis is the ratio of software solver runtime to reconfigurable hardware runtime. It does not include the compilation time and the time to configure the FPGAs.

As we see from Figure 29.6, the result indicates that even though the reconfigurable solution has a clock rate 82 times slower than that of the microprocessor-based system, it can still achieve 20 times or greater speedup

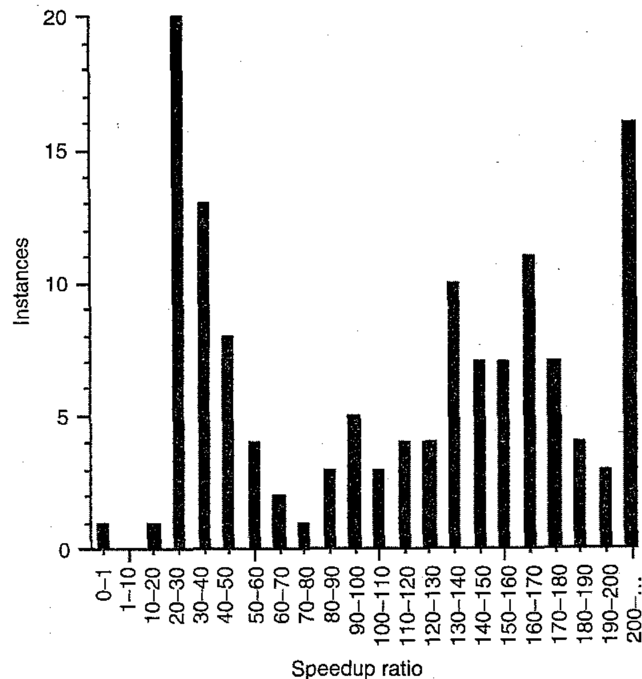

FIGURE 29.6 ■ A performance comparision of the FPGA SAT accelerator and the software version implementing the same algorithm as hardware.

for many problems. It should be noted that the comparison is based on runtime alone. The reconfigurable approach suffers from compilation overhead, which in 1998 required hours to perform logic synthesis and placement and route for the FPGAs. Current FPGA tools can perform such compilation within a minute. Ways to ameliorate compilation issues will be discussed in later sections.

For an understanding of the speedup results, Table 29.1 shows the speedup ratios for different problems. The average number of clause evaluations per cycle serves as a rough measure of the utilization of parallelism. It is defined as the number of clauses that contain at least one literal from the variables newly assigned in the previous clock cycle. There is a correlation between parallelism in clause evaluation and speedup ratio. Another factor in the speedup is that custom hardware effectively reduces a complex operation into single-cycle implication.

29.3.3 Implementing an Improved Backtrack Algorithm with Reconfigurable Hardware

The example in the previous section shows the performance benefit of reconfigurable computing. However, the hardware solution was implemented with the

TABLE 29.1 ■ Speedup ratios for different problems

Problem	Number of clauses	Average clause evaluations/cycle	Clock rate (MHz)	Speedup ratio
aim-50-2_0-yes1-2	100	7.1	1.78	44.5
aim-100-2_0-yes1-4	200	8.4	0.95	20.9
aim-200-6_0-yes1-1	1200	62.3	0.92	101
dubois20	160	8.0	1.78	13.9
hole7	204	18.3	1.78	44.5
hole8	297	21.9	1.78	45.6
hole9	415	25.9	1.57	40.2
hole10	561	30.1	1.48	41.4
ii8a2	800	15.8	1.07	923
par-8-1-c	254	29.4	1.57	174
par-16-1-c	1264	60.4	0.99	153
pret60_40	160	8.5	2.05	39
ssa0432-003	1027	11.0	0.95	24.7

basic backtrack algorithm, and improvements to the algorithm have brought thousands of times speedup in the software solution. The following example shows a more sophisticated backtrack algorithm with reconfigurable computing. As demonstrated by GRASP, conflict analysis helps identify the true reasons for conflict. Nonchronological backtracking and learning based on the analysis can greatly improve search efficiency.

Knowing that the hardware can perform fast implication checking, an alternative to conflict analysis-based backtracking was developed through trial assignments. When a conflict is detected, there are two possible scenarios regarding the most recently assigned variable. In the first, the variable has just been assigned by branching—it will be assigned the alternative value and tested. In the second, the variable has been assigned to an alternative value because of previous conflicts, so backtracking is needed. GRASP shows that conflict analysis can identify the reasons for conflict and may backtrack multiple levels, saving search time.

In the reconfigurable hardware approach, trial backtrack is performed. The algorithm moves back one decision level at a time and flips the assigned variable. Unlike a real backtrack, the most recent assignment is not turned to unknown. Instead, two implication/conflict tests are run for both value 0 and value 1. If both lead to conflict, we can trial-backtrack another level. If either case leads to no conflict, we have seen the real backtrack destination and the search reverts to regular search mode. This leads to much improved performance, with the only drawback being an increase in finite-state machine complexity.

Figure 29.7 is a diagram of the state machine for this enhanced algorithm. It is an extension of the basic backtrack algorithm, but with nine states instead of five.

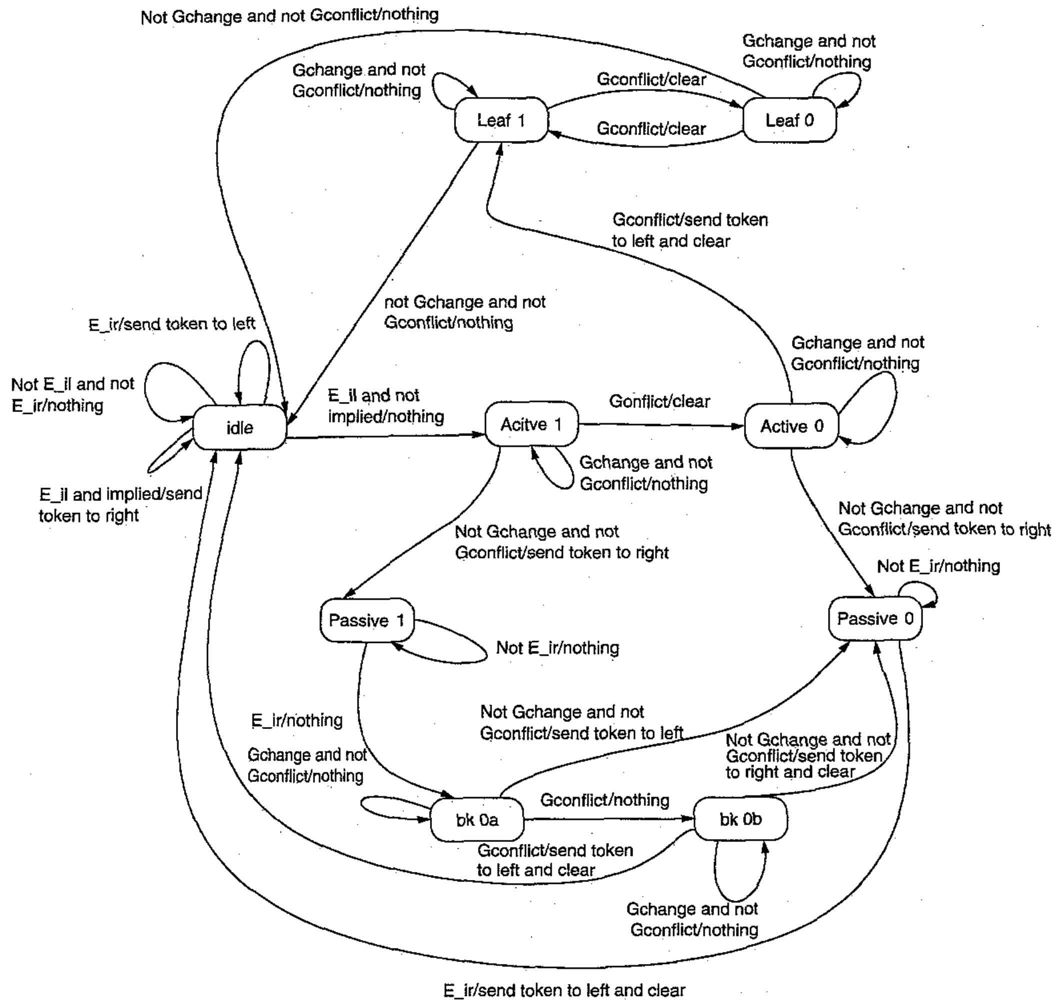

FIGURE 29.7 ■ A state diagram of the improved algorithm.

- *Idle:* This is the state before branch; it is also the state if the value is already determined by implication.
- *Active 1:* This is the state after branch on value 1.
- *Active 0:* This is the state after backtrack on the branched value 1. When a conflict is detected, instead of a simple backtrack, a new phase of testing is added. It passes the token to the left and transitions to leaf 1.
- *Passive 1:* The variable value is 1 because of branching, and active control has been passed to the right in branching.
- *Passive 0:* The variable value is 0 because of backtracking, and active control has been passed to the right in branching.

- *Leaf 1:* Leaves 1 and 0 are testing states after conflict is detected with value 0. If the testing settles with no conflict, we have found the most recent branch assignment that contributes to the conflict. The FSM will backtrack directly to that variable. If a conflict is detected, it will try a 0 value in the leaf 0 state.
- *Leaf 0:* This is also a testing state. If the testing settles with no conflict, we have found the most recent branch assignment that contributes to the conflict. The FSM will backtrack directly to that variable. If a conflict is detected, it will switch to 1 and continue the testing.
- *bk0a:* This state works in coordination with the leaf 0 state. It is reached through testing backtrack to the passive 1 state. If the test results in no conflict, this variable is the backtrack target.
- *bk0b:* This state works in coordination with the leaf 1 state. If the test results in no conflict, this variable is the backtrack target. If the conflict persists, FSM passes the token to the left and returns to idle.

29.4 A DIFFERENT APPROACH TO REDUCE COMPILATION TIME AND IMPROVE ALGORITHM EFFICIENCY

A practical issue in creating an FPGA-based SAT solver circuit optimized to a specific problem instance is the time needed to generate the circuit. While the VHDL for the solver circuit can be generated in less than a second, the process of FPGA compilation is quite long. It can take at least 10 to 20 minutes to compile the mapping for a single FPGA. FPGA hardware and software have improved to the point that a compilation may take a few minutes; however, compilation time still cannot be ignored. In the next section we describe an SAT solver with reduced compilation time and a further improved algorithm.

29.4.1 System Architecture

The solution described in the previous section directly maps the SAT formula into an SAT solver circuit. It does, however, have limitations:

- The circuit design does not take into account any physical design issues. The implication circuit includes connections between state machines that may be placed far away from each other. There are also wide OR gates that generate global control signals. The solver requires massive routing resources, and the system clock rate is low.
- The circuit is a complex netlist with little locality, and it takes a long time to compile into FPGA configurations.
- The solver implements the basic backtrack search algorithm. Although an improved nonchronological backtracking was implemented, the architecture does not support learning.

To deal with these issues, we developed a follow-on SAT solver with lessons learned from the previous design [13, 14]. The following characteristics of the new design address the previous design's shortcomings:

- Structural regularity is a high priority. A regular structure allows easier physical design. Specially designed processing elements allow regular placement and distributed processing. Overall, modular approaches can improve clock speed and allow fast circuit generation.
- Shared-wire global signaling is used to distribute data across the system. For example, a pipelined ring-style bus replaces the random interconnects. The bus allows a faster clock rate, a low pin count between chips, and a regular structure.
- The algorithm control is separated from the parallel data processing in the architecture. This allows the development of sophisticated control algorithms.
- Algorithm improvements have been implemented. In addition to implication, the circuit is capable of conflict analysis. Therefore, nonchronological backtracking and learning can be implemented.

The core of the new design is an optimized pipelined bus system, in which the bus width can be customized according to the hardware resources. The bus includes both control and data bits. The control bits notify the processing elements of actions to take; the data bits utilize a fixed sequence to encode the variable values. The system uses the same 2-bit encoding for variable values. Thus, a width of 32 data bits supports 16 variables. Also, the variables are encoded with a fixed order. For example, if at clock t the variables are v_1 through v_{16}, then, at $t+1$, the variables are v_{17} through v_{32}. In n clock cycles, $w*n$ variables pass through a stage, where $2w$ is the bit width of the data bus. The bus only propagates the variable value. There is no need to propagate the variable identification because it is inferred from the sequence. At each stage, the data bit may be OR'ed with a local signal, allowing it to be set to 1.

Figure 29.8 shows the global topology. The bus width is 40 bits, with 32 bits for data and 8 bits for control. Figure 29.9 shows one stage of the bus. The value is accessible to the PE as `Vi_in`. The propagated value can be set or reset through the signals `Vi_set` and `Vi_reset_n`. The main control block is the core of the algorithm control. It maintains an internal copy of the variable states and controls the backtrack algorithm.

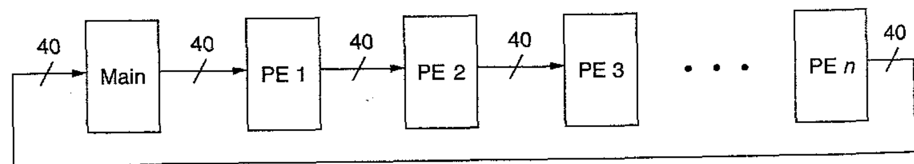

FIGURE 29.8 ■ The global topology for processing and communication in the new SAT architecture, with improved conflict analysis and nonchronological backtracking.

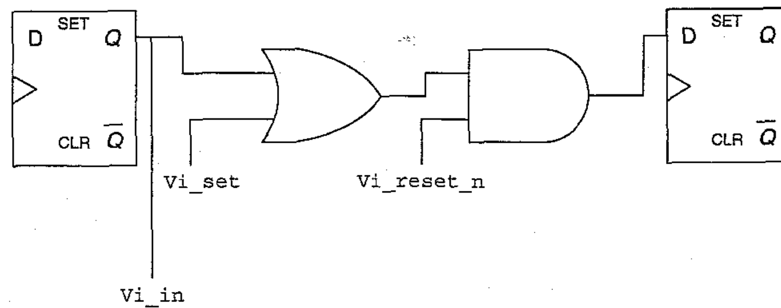

FIGURE 29.9 ■ One stage of the pipelined bus.

Multiclause modules can be placed in one processing element (PE). The total number of PEs depends on the total number of clauses in the CNF and the number of clauses per PE. Each PE contains a resettable counter to count the sequence of variables. The clause modules use the counter to identify variables on the bus.

A clause module holds the data corresponding to one clause. To simplify the hardware design, a 3-SAT formula is assumed (i.e., each clause has at most three literals). This assumption does not lose generality, because any SAT formula can be transformed into a 3-SAT formula in polynomial time by introducing new variables and breaking up long clauses. Each clause module has the following functions:

- *Implication.* Each clause should check for implication and put implied values onto the bus.
- *Conflict analysis.* This is the reversal of the implication process. Given an implied variable, the module finds the variables that lead to the implication.
- *Storage and interface.* The module interfaces with the bus, taking commands and variable values from it. It also sends new values and flags for value updates to the bus. It needs to store the values of variables related to the clause as well as the implication information.

Clause modules have three basic states: reset, implication, and analysis. The reset state will reset variables to (0, 0) if the corresponding value on the bus is (0, 0) and the state bus dictates reset. It is used during backtrack to undo the decisions and implications made after the backtrack point. Implication uses the same algorithm defined in the previous section. However, because the variable value is propagated on the bus, the clause module should also hold variable values locally. The data latching takes place when the PE counter matches the count stored in the module. The implied value is also stored locally until the correct bit passes through. The module will update the bus value at that moment. An internal flag denotes the implied value. It will be used in the analysis phase.

The analysis phase is the reverse of implication. The goal is to find the list of branch decisions that are transitive predecessors. This can be easily obtained

if the history is stored. When the clause module is in analysis mode, it will be idle if it has not generated an implication. If it has generated an implication, it will check if the implied literal is asserted on the bus during analysis. If so, the module will reset this literal on the bus and set the complement of other literals in the clause. In this way it signals to the units that generated the values of these other literals. For example, in the clause $(v_i + v_j + \neg v_k)$, if v_i is implied, the implying predecessors are $v_j = 0$ and $v_k = 1$. These variables may in turn be implied by other variables.

The main control unit handles flow control and decision making. It has the following major states and functions:

- *Branch.* Branch chooses the next free variable and assigns a value to it. Using a fixed variable order and always choosing 1 simplifies the function. A priority encoder can quickly select the first row with a free variable and assign it to 1. The branch state is associated with the first round of broadcasting the variable values. The next state is implication.
- *Implication.* The controller checks for conflicts, in which case it performs conflict analysis. Alternatively, if in two cycles of data movement no new values have been found, all iterative implications have settled. It then performs the next round of branching.
- *Conflict analysis.* This step identifies the variable assignments leading to the conflict. The control bus shows the analysis state. The conflict variable is set to (1, 1), while all other variables are set to (0, 0). When a clause that implied a variable currently asserted on the bus is found, that implied literal is reset to 0 and the implying literals are all set to 1.

When a conflict arises from a branch, a list of variable assignments contributing to it can be collected through conflict analysis. The current branching variable is considered to be implied by this set of literals. The implication is stored in the main control unit and can be expressed as a redundant clause. For example, if assignments $v_i = 1$, $v_j = 1$, $v_k = 1$, $v_l = 1$ lead to conflict, the new clause is $(\neg v_i + \neg v_j + \neg v_k + \neg v_l)$. If v_l is the current branch variable, it is implied to be 0 by this new clause. Conceptually, the new value is not a branch decision. Rather, it is forced to be the opposite value because of the recent conflict. It is a redundant implication not explicitly visible from the original formula. Adding the new clause to the database is a learning process that has been used in modern SAT solvers to prune future search space. Such learning can be carried out in hardware by reserving some FPGAs for this purpose and generating new compilations during runtime.

29.4.2 Performance

The performance of the new design is shown in Table 29.2. It should be noted that the table lists the cycle counts, but the clock rates of the two designs are different. The new design has a regular structure, and communication is pipelined. It is therefore easy to achieve a much higher clock rate. Based on the same Xilinx XC4000 FPGAs, the earlier design, implemented on the IKOS

TABLE 29.2 ■ Performance comparison

Problem	Acceleration of new design without added clauses	Acceleration of new design with added clauses
aim-50_2_0-yes1-2	33.00	65.87
aim-200-6_0-yes1-1	1.32	3.66
aim-50-1_6-no-1	8.10	487.19
aim-50-2_0-no-1	4.95	2449.26
aim-50-2_0-no-4	13.89	1121.68
aim-100-1_6-yes1-1	20.57	4354.04
aim-100-3_4-yes1-4	2.81	10.58
hole7	4.63	4.63
hole8	3.95	3.95
hole9	3.46	3.46
par8-1-c	5.03	5.03
par16-1-c	1.29	1.29
pret60_40	4.05	2154.23
ssa0432-003	0.65	2.04

Note: The comparison is based on normalized speedup against the old design, assuming 20 × clock speed improvement in the new design.

Logic Emulator, achieved a 1- to 2-MHz clock rate. The new design could easily achieve a 20-MHz clock rate in 1998. In 2006, the achievable clock rate was in the range of 200 MHz. This shows that the new design will likely achieve better performance even without added clauses. Still, added clauses can bring dramatic improvement in many problems.

29.4.3 Implementation Issues

One of the objectives of the new design is to reduce compilation time by exploiting its regular structure. However, typical FPGA tools use simulated annealing or similar algorithms to place the components. They are not capable of utilizing the regular structure automatically, and so a regular structure will not yield faster compilation times. It is necessary to bypass the automated tool and directly generate the system layout.

JBits is a tool set that allows direct programming of Xilinx FPGAs. It is an application programming interface (API) to the Xilinx configuration bitstream file that permits Java applications to dynamically modify Xilinx XC4000EX/XL bitstream configurations quickly.

A two-step approach can take advantage of the JBits tool and effectively reduce compilation time. The first step is to create a generic SAT solver template mapped to the FPGAs. The second step is customization to modify the configuration according to a specific problem instance. For each instance, only the second step is needed to compile the SAT solver. It can be performed quickly if the number of changes is small.

The architecture described in the previous section is used with additional constraints to minimize the customization. At each pipelined stage of the bus, multiple clause modules are connected to the bus. By limiting the problem formulation to 3-SAT, all clause modules are the same. The only difference is the variable identification of these three variables and the bus connection. The variable identification is expressed as a constant that can be programmed as a ROM that feeds a comparator. The connection to the bus also depends on the variable identity and polarity.

The points where a clause module wire interconnects with the bus wire should be programmed in the second step. Another simple constraint, that each bus wire connect to no more than one clause module, can be met with a simple greedy assignment algorithm.

The complete methodology to create an SAT solver is as follows:

1. *Design of a single clause module.* An SAT clause module is designed in VHDL. The synthesized netlist is further optimized manually. The design is expressed by schematic capture, which provides a more direct correspondence between design and implementation.
2. *Placement and routing of the module in a bounding box.* Placement constraints/floorplanning sets the bounding box of the clause module. The Xilinx tool automatically places and routes within the bounding box.
3. *Manual improvement.* The Xilinx EPIC tool provides a graphical user interface to manually edit the placement and routing on the FPGA.
4. *Solver generation.* With the bounding box constraints, a sample SAT solver is generated. Additional manual editing creates a regular layout.
5. *Template extraction.* The JBits tool reads the configuration bitstream and identifies the modification points.
6. *Java generator.* The SAT solver generator is created in Java with the JBits library and templates.
7. *Instance-specific bitstream.* The SAT solver generator is run with the problem instance, and the bitstream files are created.
8. *Load/run.* The programming is loaded to the FPGAs and the solver is run.

Only steps 7 and 8 are needed for each problem instance. For this reason, the compilation time is reduced from hours to merely seconds compared to the logic emulator implementation.

The target implementation is the Xilinx XC4036EX FPGA. Each FPGA contains 36×36 CLBs, and each clause module takes 4×16 CLBs. Sixteen clauses are placed in each FPGA. Each FPGA forms a stage of the pipeline, and multiple FPGAs can form a ring. The Sun Java 1.1.7 tool is used to compile and run the Java program. The host computer is an Intel Pentium Pro running Microsoft NT 4.0. The CPU clock rate is 200 MHz, and the main memory is 128 MB.

Table 29.3 shows the performance comparison, with times given in seconds. The Old Hardware and New Hardware columns include the time to create the FPGA mapping (CAD) and the time to find the solution on the hardware engine (HW). Numbers in parentheses are speedups as compared to the GRASP software.

Table 29.3 ■ Performance comparison between the standard GRASP software and two versions of the hardware SAT solver

	GRASP	Old hardware			New hardware		
Problem	**SW**	**CAD**	**HW**	**Total**	**CAD**	**HW**	**Total**
a50-2_0-y1-2	0.05	10783	0.0011 (45x)	10783	1.9	0.0004 (125x)	19 (<1x)
a100-2_0-y1-4	894	89530	42 (21x)	89572	2.4	9.7 (92x)	12.1 (74x)
a200-6_0-y1-1	128	>100K	1.35 (94x)	>100K	7.9	0.89 (144x)	8.8 (14x)
dubois20	986	11377	70.8 (14x)	11447	2.3	8.44 (117x)	10.7 (92x)
par8-1-c	0.02	12834	0.000011 (1818x)	12834	2.7	0.000035 (571x)	2.7 (<1x)
par16-1-c	202	83191	1.3 (155x)	83192	9.4	2.2 (92x)	11.6 (17x)
pret60_40	705	12396	18 (39x)	12414	2.3	9 (78x)	11.3 (62x)
Geometric Mean			75.6x	<1x		(134x)	(4.14x) (27.6x speedup problems only)

29.5 DISCUSSION

Many groups have demonstrated that reconfigurable computing, compared to software, can achieve speedups of about 100 times in solving SAT problems. The main reasons are massive parallelism and fine-grained operation due to customized hardware. Software/hardware solutions have been explored to reduce hardware complexity and allow larger problems to be solved. A recent survey of these systems is presented by Skliarova and Ferrari [15].

In each of the software/hardware systems, the massive computation to find unit resolutions/implications and conflicts is the target of hardware acceleration. However, there are several differences among these SAT solvers:

- *Algorithms.* The base algorithms are different. Several of them are based on backtracking similar to that of GRASP. Some use a full variable assignment and employ flipping during the search. Some use matrix representations.
- *Logic engine implementation.* Different styles are used to implement the massively parallel engine. Some use circuit translation, where the SAT formula is translated into logical circuits. This means that the FPGA configuration must be compiled for each problem instance, which is slow. Alternatively, the formula is translated into memory, often distributed into small blocks, which can avoid the compilation time.
- *HW/SW organization.* Some implementations are all hardware, where the entire solver is mapped onto one or multiple FPGAs. Some implementations are SW/HW, in which part of the problem is handled by software.

While there has been significant progress in reconfigurable SAT solvers, we do not see them replacing software solvers in real applications for several reasons:

- *The need for flexibility.* The SAT problem is NP-complete—that is, the worst case is assumed to be exponential to the problem size. However, sophisticated heuristics make many large problems solvable in practice. Modern software SAT solvers typically contain many heuristics and allow the user to choose different heuristic combinations to tackle especially hard problems. Reconfigurable solvers generally have only a few heuristics, and there is little flexibility on which ones to use.
- *Algorithm efficiency.* Most reconfigurable SAT solvers have algorithm efficiencies similar to that of the basic backtrack algorithm with some simple heuristics. In the meantime, software algorithms have made significant efficiency gains. More elaborate analysis, such as conflict analysis, leads to more efficient backtracking and learning. Learning can improve SAT solver speed by several orders of magnitude. Reconfigurable SAT solvers generally lag in algorithm sophistication.
- *The scalability of hardware.* The implementations of reconfigurable SAT solvers are generally limited to moderate-size problems. However, large problems are more likely to benefit from hardware acceleration.

Many projects have designed Boolean satisfiability solvers with reconfigurable computing. These projects demonstrate the performance potential of these solvers through fine-grained custom hardware and massively parallel processing. Significant progress has been made in software algorithms as well, and recently, reconfigurable computing solutions have not kept up in incorporating these innovations. This is partly because the tools for reconfigurable computing are not yet mature.

Future research may result in a breakthrough by studying these issues:

- *Hardware/software solution.* The complex algorithms are difficult to implement and verify in hardware. It is more efficient to partition the problem and allocate only the massively parallel portion to the reconfigurable hardware. With microprocessors embedded in FPGAs, such as Xilinx Virtex-II Pro and Virtex-4, communication between the processor and the FPGA is greatly improved. The proliferation of multicore processors and high-bandwidth interconnects enables the exploitation of parallelism at different levels with heterogeneous processing technologies.
- *System-level design and synthesis methodologies.* Models of computation that preserve concurrency can be mapped to heterogeneous multicore architectures. The designer can decide the trade-off between parallelism and hardware usage. FPGA-based fabrics provide the massive parallelism and low-level customization, while other components, such as embedded processor or controller, can be chosen for their desirable characteristics.
- *Distribution of data and customization of hardware.* Mapping SAT formulas to FPGA circuits generates random routing and requires long compilation times. Mapping problem instances into distributed memory

blocks can solve the time issue but it forces some degree of sequential access. Learning from the design of content addressable memory may lead to hardware architectures better able to solve SAT and other Boolean problems.

- Simultaneous exploration of multiple states. Creating an algorithm that can efficiently explore multiple states in the assignment space simultaneously will allow the utilization of large amounts of computing resources. A simplified approach is to simultaneously run the search on multiple machines with different heuristics. However, efficient utilization of learning across different searches remains an open problem.

References

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest. *Introduction to Algorithms*, MIT Press, 1990.

[2] T. Larrabee. Test pattern generation using Boolean satisfiability. *IEEE Transactions on Computer-Aided Design* 11, January 1992.

[3] A. Biere, A. Cimatti, E. M. Clarke, Y. Zhu. Symbolic model checking without BDDs. *Proceedings of the Workshop on Tools and Algorithms for Analysis and Construction of Systems (TACAS)* 1579, LNCS, 1999.

[4] A. Gupta, M. Ganai, C. Wang, Z. Yang, P. Ashar. Learning from BDDs in SAT-based bounded model checking. *Proceedings of the Design Automation Conference*, 2003.

[5] M. Davis, H. Putnam. A computing procedure for quantification theory. *Journal of the ACM* 7, 1960.

[6] J. P. Marques-Silva, K. A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers* 48(5), May 1999.

[7] R. J. Bayardo Jr., R. C. Schrag. Using CSP look-back techniques to solve real-world SAT instances. *Proceedings of the 14th International Conference on Artificial Intelligence*, 1997.

[8] E. Goldberg, Y. Novikov. BerkMin: A fast and robust SAT-solver. *Design, Automation and Test in Europe*, 2002.

[9] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, S. Malik. Chaff: Engineering an efficient SAT solver. *Proceedings of the 38th Design Automation Conference*, 2001.

[10] P. Zhong, M. Martonosi, P. Ashar, S. Malik. Using configurable computing to accelerate Boolean satisfiability. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 18(6), June 1999.

[11] P. Zhong, P. Ashar, S. Malik, M. Martonosi. Using reconfigurable computing techniques to accelerate problems in the CAD domain: A case study with Boolean satisfiability. *Proceedings of the 35th Design and Automation Conference*, June 1998.

[12] P. Zhong, M. Martonosi, P. Ashar, S. Malik. Accelerating Boolean satisfiability with configurable hardware. *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, April 1998.

[13] P. Zhong, M. Martonosi, P. Ashar, S. Malik. Solving Boolean satisfiability with dynamic hardware configurations. *Proceedings of the Eighth International Workshop on Field-Programmable Logic and Applications: From FPGAs to Computing Paradigms*, August–September 1998.

[14] P. Zhong, M. Martonosi, P. Ashar. FPGA-based SAT solver architecture with near-zero synthesis and layout overhead. *IEE Proceedings on Computer and Digital Techniques* 147(3), May 2000.

[15] I. Skliarova, A. B. Ferrari. Reconfigurable hardware SAT solvers: A survey of systems. *IEEE Transactions on Computers* 53(11), November 2004.

[16] M. Yokoo, T. Suyama, H. Sawada. Solving satisfiability problems using field-programmable gate arrays: First results. *Proceedings of the Second International Conference on Principles and Practice of Constraint Programming*, 1996.

[17] T. Suyama, M. Yokoo, H. Sawada, A. Nagoya. Solving satisfiability problems using reconfigurable computing. *IEEE Transactions on VLSI Systems* 9(1), 2001.

[18] T. Suyama, M. Yokoo, A. Nagoya. Solving satisfiability problems on FPGAs using experimental unit propagation. *Proceedings of the Fifth International Conference on Principles and Practice of Constraint Programming*, 1999.

[19] T. Suyama, M. Yokoo, H. Sawada. Solving satisfiability problems using logic synthesis and reconfigurable hardware. *Proceedings of the 31st Hawaii International Conference on System Sciences* 7, 1998.

[20] J. de Sousa, J. P. Marques-Silva, M. Abramovici. A configware/software approach to SAT solving. *Proceedings of the Ninth IEEE International Symposium on Field-Programmable Custom Computing Machines*, 2001.

[21] I. Skliarova, A. B. Ferrari. A software/reconfigurable hardware SAT solver. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 12(4), April 2004.

[22] J. Gu. Local search for satisfiability (SAT) problem. *IEEE Transactions on Systems, Man, and Cybernetics* 23(4), July 1993.

[23] H. Zhang, M. Stickel. An efficient algorithm for unit-propagation. *Proceedings of the Fourth International Symposium on Artificial Intelligence and Mathematics*, 1996.

[24] H. Zhang. SATO: An efficient propositional prover. *Proceedings of the International Conference on Automated Deduction*, 1997.

[25] L. Zhang, S. Malik. The quest for efficient Boolean satisfiability solvers. *Proceedings of the Eighth International Conference on Computer-Aided Deduction*; *Proceedings of 14th Conference on Computer-Aided Verification*, July 2002.

[26] L. Zhang, S. Malik. Validating SAT solvers using an independent resolution-based checker: Practical implementations and other applications. *DATE2003*, March 2003.

[27] F. A. Aloul, A. Ramani, I. L. Markov, K. A. Sakallah. Solving difficult instances of Boolean satisfiability in the presence of symmetry. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 22(9), September 2003.

[28] P. T. Darga, M. H. Liffiton, K. A. Sakallah, I. L. Markov. Exploiting structure in symmetry detection for CNF. *Proceedings of the 41st IEEE/ACM Design Automation Conference*, 2004.

[29] Y. Oh, M. N. Mneimneh, Z. S. Andraus, K. A. Sakallah, I. L. Markov. AMUSE: A minimally-unsatisfiable subformula extractor. *Proceedings of the 41st IEEE/ACM Design Automation Conference, 2004.*