Solving Boolean Satisfiability with Dynamic Hardware Configurations

Peixin Zhong, Margaret Martonosi, Pranav Ashar, and Sharad Malik

Dept. of Electrical Engineering
Princeton University

NEC CCRL, Princeton NJ USA

Abstract. Boolean satisfiability (SAT) is a core computer science problem with many important commercial applications. An NP-complete problem, many different approaches for accelerating SAT either in hardware or software have been proposed. In particular, our prior work studied mechanisms for accelerating SAT using configurable hardware to implement formula-specific solver circuits. In spite of this progress, SAT solver runtimes still show room for further improvement.
In this paper, we discuss further improvements to configurable-hardware-based SAT solvers. We discuss how dynamic techniques can be used to add the new solver circuitry to the hardware during run-time. By examining the basic solver structure, we explore how it can be best designed to support such dynamic reconfiguration techniques. These approaches lead to several hundred times speedups for many problems. Overall, this work offers a concrete example of how aggressively employing on-the-fly reconfigurability can enable runtime learning processes in hardware. As such, this work opens new opportunities for high performance computing using dynamically reconfigurable hardware.

1 Introduction

Boolean satisfiability (SAT) is a core computer science problem with important applications in CAD, AI and other fields. Because it is an NP-complete problem, it can be very time-consuming to solve. The problem's importance and computational difficulty have attracted considerable research attention, and this prior research has included several recent attempts using configurable hardware to accelerate SAT solvers [9, 1, 10, 11].

In earlier work, we presented a novel approach to solving the SAT problem in which formula-specific hardware was compiled specially for each problem to be solved [11]. Implications are computed via Boolean logic customized to the formula at hand. Each variable in the formula was given a small state machine that dictated when new values should be tried for this variable. Control passed back and forth between different variables in this distributed, linearly-connected, set of state machines.

The advantages of our formula-specific approach are that the interconnect between clauses and variables are tightly specialized to the problem at hand. This leads to very low I/O and memory requirements. It also allows implication processing to be much faster compared to some previous approaches. As a result, we achieved order-of-magnitude speedups on many large SAT problems.

Our formula-specific approach, however, has two main disadvantages. First, the FPGA compile-time is on the critical path of the problem solution. This means that only long-running problems can garner speedups once compile-time is taken into account. A second problem is that the global interconnect and long compile-times make it inefficient to insert any on-the-fly changes to the SAT-solver circuit.

The second problem is particularly vexing because it precludes a key strategy employed by current software SAT solvers such as GRASP [7]: dynamic clause addition. In software solvers, as the problem is worked on, additional information about the formula is distilled into extra Boolean clauses which are added into the Boolean formula that defines the problem. These additional clauses allow one to more efficiently prune the search space, and they led to sizable performance improvements in the GRASP software SAT solver.

The work we describe here evaluates new designs for solving SAT in configurable hardware that use novel architectures and dynamic reconfiguration to allow us to circumvent the problems raised by our previous design. First, instead of an irregular global interconnect, a *regular* ring-based interconnect is used. This avoids the compile-time limitations of our prior approach. Second, by including generalized "spare" clause hardware in the design, we can add extra clauses as the solution progresses, which gives our approach the same advantages as GRASP's added clauses technique.

The remainder of this paper is structured as follows. In Section 2, we describe the SAT problem in more detail. Section 3 discusses design alternatives for solving it in configurable hardware, and then Section 4 proposes a particular hardware mapping that we evaluate further. Section 5 gives an overview of related work and Section 6 offers our conclusions.

2 The SAT algorithm

The Boolean satisfiability (SAT) problem is a well-known constraint satisfaction problem with many practical applications. Given a Boolean formula, the goal is either to find an assignment of 0-1 values to the variables so that the formula evaluates to 1, or to establish that no such assignment exists.

The Boolean formula is typically expressed in conjunctive normal form (CNF), also called product-of-sums form. Each sum term (clause) in the CNF is a sum of single literals, where a literal is a variable or its negation. In order for the entire formula to evaluate to 1, each clause must be satisfied, *i.e.*, at least one of its literals should be 1.

An assignment of 0-1 values to a subset of variables (called a partial assignment) might satisfy some clauses and leave the others undetermined. If an undetermined clause has only one unassigned literal in it, that literal must evaluate to 1 in order to satisfy the clause. In such a case, the corresponding variable is said to be *implied* to that value. A variable is considered free if neither assigned nor implied. A conflict arises if the same variable is implied to be different values. This means that the corresponding partial assignment cannot be a part of any valid solution.

Most current SAT solvers are based on the Davis-Putnam algorithm [3]. This is a backtrack search algorithm. The basic algorithm begins from an empty

assignment. It proceeds by assigning a 0 or 1 value to one free variable at a time. After each assignment, the algorithm determines the direct and transitive implications of that assignment on other variables. If no conflict is detected after the implication procedure, the algorithm picks the next free variable, and repeats the procedure (forward search). Otherwise, the algorithm attempts a new partial assignment by complementing the most-recently assigned variable (backtrack). If this also leads to conflict, this variable is reset to the free value and the next most-recently assigned variable is complemented. The algorithm terminates when: (i) no free variables are available and no conflicts have been encountered (a solution has been found), or (ii) it wants to backtrack beyond the first variable, which means all possible assignments have been exhausted and there is no solution to the problem.

Determining implications is crucial to pruning the search space since it allows the algorithm to skip regions of the search space corresponding to invalid partial assignments.

Recent software implementations of the SAT algorithm have enhanced it in several ways while maintaining the same basic flow [6, 2, 8, 7]. The contribution of the GRASP work [7] is notable since it applies non-chronological backtracking and dynamic clause addition to prune the search space further. Significant improvements in run time are reported.

2.1 Conflict Analysis

Much of the performance improvement reported by GRASP comes from their implementation of conflict analysis. When the basic Davis-Putnam algorithm observes a conflict, it backtracks to change the partial assignment. It does not, however, analyze which variable is the true reason for the observed conflict. The backtrack process may complement variables irrelevant to the conflict and repeatedly explore related dead ends. More sophisticated conflict analysis works to identify the variable assignments that lead to the conflict. Acting as a reverse implication procedure, conflict analysis identifies the transitive predecessors of the implied literals leading to the conflict.

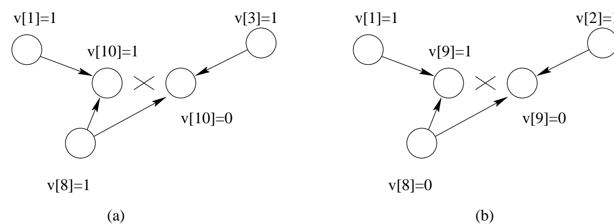

Fig. 1. Example of implication graphs. (a) Conflict after assignment v[8]=1. (b) Conflict after altered assignment v[8]=0.

Consider, for example, the partial formula $(\bar{v}_1 + v_8 + v_9)(\bar{v}_2 + v_8 + \bar{v}_9)(\bar{v}_1 + \bar{v}_8 + v_{10})(\bar{v}_3 + \bar{v}_8 + \bar{v}_{10})$. If v_1,v_2 ... and v_7 are previously assigned to 1 and then

v_8 is assigned 1, the resulting implication graph is shown in Fig. 1(a). A conflict is raised on v_{10}. The predecessors for the conflict are v_1, v_3 and v_8. Similarly, when v_8 is changed to 0 (Fig. 1b), it generates a conflict in v_9. The causes are v_1, v_2 and $\bar{v}_8$. At this point, we know either value of v_8 will lead to a conflict. The basic algorithm would change the value of v_7 to 0 and try again. It is, however, v_1, v_2 and v_3 that are actually responsible. Therefore, we can directly backtrack to the most recently assigned variable causing the dual conflict, i.e. v_3.

2.2 Dynamic Clause Addition

In the previous example, we showed how conflict analysis can deduce after the fact that if v_1, v_2 and v_3 are all 1, the formula can not be satisfied. This piece of information is not obvious, however, when these variables are assigned. To add such learned information into the solver's knowledge base, we can add a new clause $(\bar{v}_1 + \bar{v}_2 + \bar{v}_3)$ to the Boolean formula being solved. Adding this clause to the formula allows the solver to detect this conflict earlier and avoid exploring the same space in the future.

Conflict analysis and clause addition are relative easy to implement in software. When a new value is implied, it is added to a data structure summarizing the implication graph. When a conflict occurs, traversing the graph backwards identifies predecessors of the conflict. In the next section, we will show this can also be implemented in configurable hardware.

3 Design Alternatives

In designing configurable hardware to solve SAT, a number of design issues arise. Overall, the key is determining what functions are implemented on hardware and how they are implemented. Design decisions should be made on these questions:

- How do we choose which variable to assign next?
- How are logical implications computed?
- How are new implications sent to other units?
- How do we detect conflicts and backtrack?
- What further mechanisms do we harness for pruning the search space?

Our previous design compiles the problem formula and the solver into one custom circuit [11]. Covering the above design decisions, we note that this approach uses a statically-determined variable ordering based on how frequently each variable appears in the formula. The clauses are translated into logic gates to generate implications. All clauses are evaluated in parallel. Implications are communicated using hardwired connections. There is no analysis when a conflict is raised, and backtracking simply reverses the order of variable assignment.

Because so much of the formula is hardwired into the design, its run-time flexibility is limited. Here we discuss different alternatives for making dynamic modifications in SAT solver circuits. We identify the ability to dynamically add new clauses as the major objective in our new design. We have evaluated the following alternatives.

3.1 Supporting Dynamic Changes through Configuration Overlays

One way to achieve dynamic hardware modification is by implementing multiple configurations of each FPGA as overlays. An initial circuit is generated according to some static strategy. As this solver runs, new information is obtained and a better circuit can be designed and compiled. When the new circuit is available, the problem is then switched to run using it. This approach is very general for applications involving run-time hardware learning, and can clearly be applied to SAT clause addition. By adding conflict analysis to the hardware, it can direct new clauses to be added, and initiate new compiles of the improved designs. These compiles occur in parallel with further solutions using the current design.

This approach is particularly attractive with a system with multiple configuration capabilities and the ability to switch quickly between them. Extra memory can be used as configuration cache. While the solver is running, the cache is updated with a new design, and then we can quickly switch to the updated configurations. There are several drawbacks however. First, few commercial products currently support multiple configuration contexts, and those that do will inevitably pay a price in decreased logic density. This approach also requires very fast FPGA compilation; if the compilation is too slow, the newer circuit may be of little value by the time it is actually compiled.

3.2 Supporting Dynamic Changes through Partial Reconfiguration

Because of the difficulties inherent in approaches relying on configuration overlays, we chose to explore alternative techniques in which we redesign our base circuit to make it more amenable to partial configuration. In essence, we want to be able to modify only a small portion of the circuit and achieve the same performance goal as with full overlays.

Focusing on dynamic clause addition, we note that it would be natural to design each clause as a module. In this way, when a new clause is generated, a new module is simply added to the circuit. In our current design, this is difficult because we have implemented a variable-oriented, rather than clause-oriented, design. Section 4 discusses a mechanism for using partial reconfiguration by leaving "spare" clause templates in the solver and then customizing them into a specific added clause during runtime. A key aspect of using partial reconfiguration is implementing communication using a *regular* communication network; this ensures that no random global routing will be needed when a new clause is added.

4 Configurable Hardware Mapping

This section describes a hardware organization based on the partial reconfiguration approach from the previous section and evaluates a SAT solver algorithm based on it. We envision this hardware being implemented on an array of FPGA chips, because one FPGA does not provide the capacity necessary for interesting (i.e., large) SAT problems.

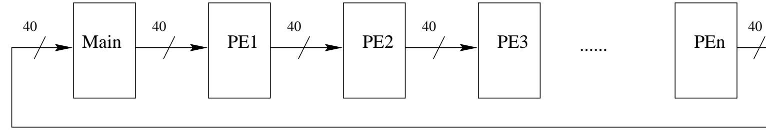

Fig. 2. Global topology of the SAT solver circuit

4.1 Hardware organization

Global topology: The circuit topology is based on a regular ring structure as shown in Fig. 2. The ring is a pipelined communication network and the processing elements (PEs) are distributed along the ring. Each PE contains multiple modules with each module representing a clause. This regular, modular design allows for the easy addition of clauses during runtime. There is also a main control unit to maintain multiple control functions.

The communication network is used to send the updated variables to other units. Previously, we used direct wire connection between different variables. This direct approach has better performance for implication passing, but lacks the modularity needed to allow dynamic circuit changes.

The bus is 40 bits wide, so the FPGA chip will use 40 I/O pins for the incoming ring signals and another 40 pins for the output to the next FPGA. The bus consists of 32 data wires and 8 control bits. Signals are pipelined through a series of D-flipflops. All the variable data bits pass through the bus in a fixed order, with a synchronizing signal at the beginning.

In our design, the variable values are all rotating on the bus. Since each variable requires two bits ('00' denotes unassigned, '10' denotes a 1, and '01' denotes a zero) each stage can contain 16 variables.

Main control: The main control maintains the global state and monitors the ring for value changes and conflicts. When a variable changes from unassigned to either 1 or 0, the ordering of such changes must be recorded so that we know in what order to backtrack. These orderings are encoded in memory at the main control module. The main control also checks for conflicts by simply monitoring the ring for the '11' variable value that indicates a contradiction.

Implication Processing: Fig. 3 shown the connection between the bus and functional modules. Between each pipeline flipflop, there are only two levels of logic gates; this allows us to achieve a high clock rate.

Processing implications is generally compute-intensive, but as with our previous approach we employ large amounts of fine-grained parallelism. For each clause in the formula, we implement a module called a *clause cell.* The clause cell has a local counter to track which variables are currently on the bus. Each clause cell monitors the values of variables relevant to it, and uses them to determine when implications are needed. For an n-literal clause, if n-1 literals are set to 0, the other literal should be implied to be true, and this implied value is then propagated around the ring.

Conflict analysis: Since conflict analysis is the inverse of implication, it is natural to merge this function into each clause cell. When a new clause is generated, the cell stores its implication. In conflict analysis mode (initiated by

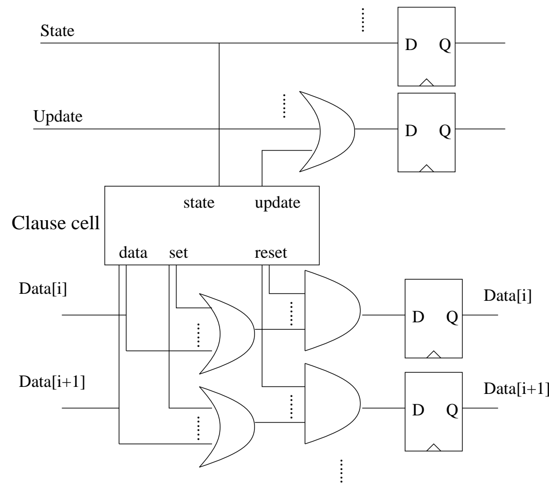

Fig. 3. One stage of the processing element

the main control), when the implied literal appears on the bus, it resets this variable and puts the predecessors on the bus.

4.2 Hardware SAT Algorithm

This hardware implements a SAT algorithm similar to software approaches. The major difference is that it tries to execute operations in parallel whenever possible. Since each clause is implemented as a separate cell in hardware, many clauses can be evaluated in parallel.

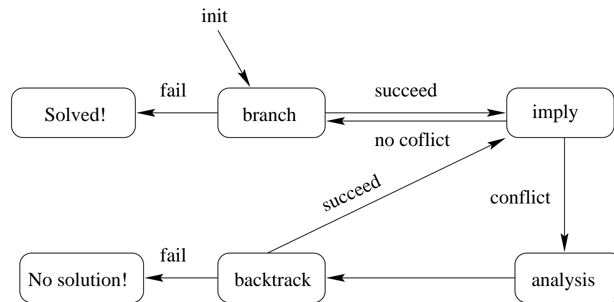

Fig. 4. Basic state diagram of the SAT solver

The basic flow diagram of the SAT solver is shown in Fig. 4. It includes four major states: Branch, Imply, Analysis and Backtrack.

Branch: After initialization (and whenever a partial assignment has successfully avoided a conflict), the circuit is in the branch state. The decision of

which variable to assign (i.e., branch on) next, is determined by the main control. In the simple approach we simulate here, the variable ordering is statically determined, but more elaborate dynamic techniques are the subject of future research. It then proceeds to the Imply state. If no more free variables exist, a solution has been found.

Imply: This state is used to compute the logical implications of the assignment. The clause cells check the values and generate implications when necessary. The main control monitors the value changes. When implications have settled, it will proceed to branch on a new variable. It also monitors conflicts in variables, and initiates conflict analysis when needed.

Analysis: To indicate conflict analysis mode, both bits of the conflicted variable are set to 1, and all other data bits are set to 0. When the clause cells detect this mode, they generate predecessors for variables with an implied value. When this process settles, only the assigned variables responsible for the conflict will have a value on the bus. This list may be used by the main control to determine when to generate a new conflict clause cell. In our work, a new clause is generated only when both assignments for a variable end with a conflict. The backtrack destination is chosen to be the most recently-assigned variable of the ones responsible for the conflict. If there is no variable to backtrack to, there is no solution to the formula because the observed conflict cannot be resolved.

Backtrack: After conflict analysis, during backtrack, the bus is reset back to the variable values to allow further implication. Variables assigned before the backtrack point take on those values. Variables after the backtrack point are reset to free. When this is done, implications are tried again.

4.3 Performance results

Problem name	FCCM98 design	New Design: No added clauses	New Design: Added clauses
aim-50-1_6-no-1	37806	93396	1552
aim-50-2_0-no-1	2418518	9763745	19749
aim-50-2_0-no-4	204314	294182	3643
aim-100-1_6-yes1-1	3069595	2985176	14100
aim-100-3_4-yes1-4	108914	775641	205845
hole6	32419	129804	129804
jnh16	84909	778697	513495
par8-1-c	176	700	700
ssa0432-003	194344	5955665	1905633

Table 1. Configurable SAT solver run-time in cycles

In order to evaluate the performance of this design, we built a C++ simulator and used DIMACS SAT problems as input [4]. Table 1 shows the number of hardware cycles needed to solve the problems. The first column of data shows

the cycle counts for the FCCM98 formula-specific approach. The next column shows performance data for the newer design discussed here. This column includes conflict analysis and non-chronological backtracking, but does not include dynamic clause addition. The following column of data is the new design with the dynamic clause addition.

Although the run-time is expressed in number of cycles in each case, the actual cycle time is very different between the FCCM98 design and the current one. In the old design, long wires between implication units made routing difficult, and typical user clock rates were several hundred KHz to 2 MHz. In the newer design, the communication is pipelined and the routing is shorter and more regular. Initial estimates are that the clock rate should be at least 20 MHz. Therefore, speedups occur whenever the new design requires 10X or fewer cycles compared to the old design.

From the results, we can see the new design without clause addition has a speedup of about 1x to 10x. Speedups occur in this case due to (1) the improved clock rate and (2) the improved conflict analysis which leads to more direct backtracking.

Implementing dynamic clause addition offers benefits that vary with the characteristics of the problems. For some problems, there is marginal or no benefit. For the aim problems, the speed-up ration ranges from less than 4 times to about 500 times. The performance gain is especially significant in the unsatisfiable problems. In these cases, the dynamically-added clauses significantly prune the search space allowing the circuit to rule the problem unsatisfiable much earlier.

5 Related Work

Prior work includes several proposals for solving SAT using reconfigurable hardware [9, 1]. Suyama *et al.* [9] have proposed their own SAT algorithm distinct from the Davis-Putnam approach. Their algorithm is characterized by the fact that at any point, a full (not partial) variable assignment is evaluated. While the authors propose heuristics to prune the search space, they admit that the number of states visited in their approach can be 8x larger than the basic Davis-Putnam approach.

The work by Abramovici and Saab also proposed a configurable hardware SAT solver [1]. Their approach basically amounts to an implementation of a PODEM-based [5] algorithm in reconfigurable hardware. PODEM is typically used to solve test generation problems. Unlike PODEM, which relies on controlling and observing primary inputs and outputs, Davis-Putnam's efficient data structures also capture relationships between *internal* variables in a circuit; this reduces the state space visited and the run time significantly [6, 2, 7].

In prior work, we designed a SAT solver based on the basic Davis-Putnam algorithm, and implemented it on an IKOS Virtualogic Emulator This work was the first to publish results based on an actual implementation in programmable logic. We also designed an improved algorithm that uses a modified version of non-chronological backtracking to prune the search space [10]. This method indirectly identifies predecessors for a conflict, but is not as efficient as the direct

conflict analysis we evaluate here. Finally, none of the prior configurable SAT solvers have employed dynamic clause addition.

Most importantly, all the prior projects involve generating formula-specific solver circuits. In these approaches, the compilation overhead can not be amortized among many runs. While they may have benefits on some long-running problems, the approach we describe here is much amenable to direct module generation and avoids much of the compiler overhead of prior work. We hope the work on fast module generation and the modular design methodology may lead to wider application of these input-specific hardware approaches.

6 Conclusions

This paper has described a new approach that takes advantage of dynamic reconfiguration for accelerating Boolean satisfiability solvers in configurable hardware. The design we evaluate is highly modular and very amenable to direct module generation. One of the key improvements of this design is its ability to dynamically add clauses. Overall, the approach has potential speedups up to 500X versus our previous configurable approach without dynamic reconfiguration. More broadly, this hardware design demonstrates the application of machine learning techniques using dynamically reconfigurable hardware.

References

1. M. Abramovici and D. Saab. Satisfiability on Reconfigurable Hardware. In *Seventh International Workshop on Field Programmable Logic and Applications*, Sept. 1997.
2. S. Chakradhar, V. Agrawal, and S. Rothweiler. A transitive closure algorithm for test generation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 12(7):1015–1028, July 1993.
3. M. Davis and H. Putnam. A Computing Procedure for Quantification Theory. *Journal of the ACM*, 7:201–215, 1960.
4. DIMACS. Dimacs challenge benchmarks and ucsc benchmarks. Available at ftp://Dimacs.Rut-gers.EDU/pub/challenge/sat/benchmarks/cnf.
5. P. Goel. An Implicit Enumeration Algorithm to Generate Tests for Combinational Logic Circuits. *IEEE Transactions on Computers*, C30(3):215–222, March 1981.
6. T. Larrabee. Test Pattern Generation Using Boolean Satisfiability. In *IEEE Transactions on Computer-Aided Design*, volume 11, pages 4–15, January 1992.
7. J. Silva and K. Sakallah. GRASP-A New Search Algorithm for Satisfiability. In *IEEE ACM International Conference on CAD-96*, pages 220–227, Nov. 1996.
8. P. Stephan, R. Brayton, and A. Sangiovanni-Vincentelli. *Combinational Test Generation Using Satisfiability.* Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, 1992. UCB/ERL Memo M92/112.
9. T. Suyama, M. Yokoo, and H. Sawada. Solving Satisfiability Problems on FPGAs. In *6th Int'l Workshop on Field-Programmable Logic and Applications*, Sept. 1996.
10. P. Zhong, P. Ashar, S. Malik, and M. Martonosi. Using reconfigurable computing techniques to accelerate problems in the cad domain: A case study with boolean satisfiability. In *35th Design Automation Conference*, 1998.
11. P. Zhong, M. Martonosi, P. Ashar, and S. Malik. Accelerating boolean satisfiability with configurable hardware. In *FCCM'98*, 1998.

This article was processed using the LaTeX macro package with LLNCS style