

# Acceleration of Satisfiability Algorithms by Reconfigurable Hardware

Marco Platzner and Giovanni De Micheli

Computer Systems Laboratory, Stanford University  
Stanford, CA 94305, U.S.A.  
marco.platzner@computer.org

**Abstract.** We present different architectures to solve Boolean satisfiability problems in instance-specific hardware. A simulation of these architectures shows that for examples from the DIMACS benchmark suite, high raw speed-ups over software can be achieved. We present a design tool flow and prototype implementation of an instance-specific satisfiability solver and discuss experimental results. We measure the overall speed-up of the instance-specific architecture that takes the hardware compilation time into account. The results prove that many of the DIMACS examples can be accelerated with current FPGA technology.

## 1 Introduction

The *Boolean satisfiability problem* (SAT) is a fundamental problem in mathematical logic and computing theory with many practical applications in areas such as computer-aided design of digital systems, automated reasoning, and machine vision. In computer-aided design, tools for synthesis, optimization, verification, timing analysis, and test pattern generation use variants of SAT solvers as core algorithms. The SAT problem is commonly defined as follows [1]: Given

- a set of  $n$  Boolean variables  $x_1, x_2, \dots, x_n$ ,
- a set of literals, where a literal is a variable  $x_i$  or the complement of a variable  $\bar{x}_i$ , and
- a set of  $m$  distinctive clauses  $C_1, C_2, \dots, C_m$ , where each clause consists of literals combined by the logical *or* connective  $\vee$ ,

determine, whether there exists an assignment of truth values to the variables that makes the Conjunctive Normal Form (CNF)

$$C_1 \wedge C_2 \wedge \dots \wedge C_m \tag{1}$$

true, where  $\wedge$  denotes the logical *and* connective.

Since the general SAT problem is NP-complete, exact methods to solve SAT problems show an exponential worst-case runtime complexity. This limits the applicability of exact SAT solvers in many areas. Heuristics can be used to find solutions faster, but they may fail to prove satisfiability.

The SAT problem is a *discrete, constrained decision problem* [1]. A straightforward but inefficient procedure to solve it exactly is to enumerate all possible truth value assignments and check if one satisfies the CNF. Many of the improved techniques that have been proposed to solve SAT problems eliminate one variable from the CNF at a time. There are two basic methods: *splitting* and *resolution*. Resolution was implemented in the original Davis-Putnam (DP) algorithm [2]. Splitting was used first in Loveland’s modification to DP, the DPL algorithm [3]. In splitting, a variable is selected from the CNF and two sub-CNFs are generated by setting the variable to 0 and 1, respectively. The iterative application of splitting generates a *search tree*; a leaf of the tree denotes a full assignment of values to variables. Most practical SAT solvers use the splitting technique and combine it with *backtracking*. Backtracking searches the search tree in a depth-first order and thus avoids excessive memory requirements.

A general template for backtracking SAT solvers is described in [4] and includes 3 steps: decision, deduction, and diagnosis. In the decision step, a variable is selected for the next assignment. In the deduction step, information is inferred from the current partial assignment. This information is then used to guide the search process, e.g., to prune the search tree. If the current partial assignment leads to a contradiction, a diagnosis step can be used to analyze this situation and to avoid running into the same contradiction in future.

Existing software SAT solvers use a wide variety of backtracking methods and strategies for decision, deduction, and diagnosis. GRASP [4] is a sophisticated SAT solver that implements all steps of the described template. We use GRASP as software reference system in our work. The powerful strategies that are implemented by sophisticated SAT solvers reduce the number of variable assignments required to find a solution or to prove that there is no solution. However, these strategies can be computationally very expensive.

The goal of our work is to speed up exact SAT solvers by exploiting the fine-grain parallelism in the SAT problem instances. For each new problem instance (CNF), a new hardware is generated that reflects the particular structure of the CNF. This class of hardware architectures is called *instance-specific* and relies on fine-grained reconfigurable computing structures, e.g., FPGAs. Instance-specific SAT solvers use less powerful strategies than software solvers for decision and deduction; diagnosis methods in hardware have not been reported at all. The advantage of SAT in hardware is that the deduction step can be implemented very fast. This is because many deduction strategies operate on values in 2-, 3-, or 4-valued logic and show large amounts of fine-grained parallelism. This makes fine-grained parallel computing structures, such as FPGAs, an optimal target.

## 2 Related Work

In this section, we mention related projects that apply reconfigurable hardware to solve the SAT problem. Zhong et al. [5] [6] described an instance-specific architecture to solve SAT problems that uses Boolean constraint propagation as deduction strategy and models the variables in 4-valued logic. They simulated

their architecture and reported speed-ups in the order of several magnitudes for the DIMACS benchmarks [7]. Their prototype translates a SAT problem into a logic description in VHDL. This description is partitioned and mapped onto an array of Xilinx XC4K FPGAs by an IKOS logic emulation system. Suyama et al. [8] proposed an architecture for SAT that combines a forward checking technique with non-chronological backtracking. They model the variables in 2-valued logic. Their tool flow also targets the Xilinx XC4K line.

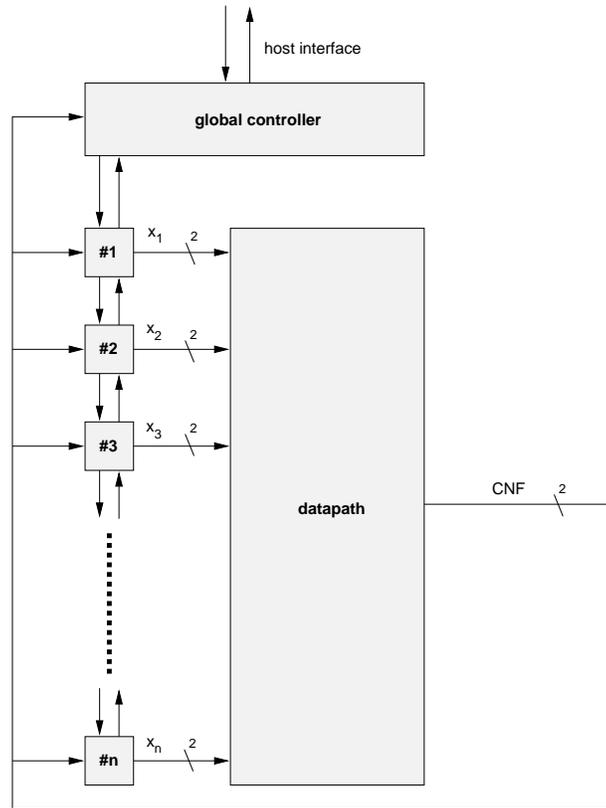
In [9], an instance-specific architecture for SAT problems in arbitrary Boolean expressions was presented. This architecture uses a strategy similar to the PO-DEM algorithm for automatic test pattern generation. The same architecture is used in [10], with an emphasis on a fast hardware compilation. To address this issue, the use of Xilinx XC62xx FPGAs is proposed which allows to develop SAT-specific tools for synthesis, partition, placement, and routing.

### 3 Hardware Architectures

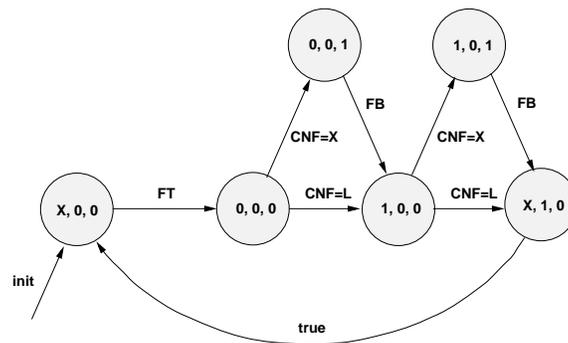
The basic architecture for backtracking search is shown in Figure 1 and consists of three blocks: i) an array of finite state machines (FSMs), ii) a datapath, and iii) a global controller. Each variable of the CNF corresponds to one FSM. The FSMs are connected in a one-dimensional array; each FSM can activate its two neighboring FSMs at the top and at the bottom. The datapath is a combinational circuit that takes the variables as input and computes outputs that are fed back to the FSMs. The global controller starts the computation and handles I/O communication. All the architectures presented in this section consist of these three blocks. However, they differ in the modeling of the variables and the used deduction strategy, which is reflected in the actual implementation of the datapath and the FSM.

#### 3.1 Architecture CE

CE (CNF evaluation) models the variables in 3-valued logic. A variable can take on the values  $\{0, 1, X\}$ , where  $X$  denotes an unassigned variable. The datapath computes the 3-valued result of the CNF expression. Initially, all variables are unassigned which also leads to CNF value  $X$ , and the global controller activates the top-most FSM. The state diagram for an FSM is shown in Figure 2. An activated FSM assigns 0 to its variable and checks the resulting CNF value. If the CNF value is 1, the partial assignment already satisfied the CNF and the computation stops. If the CNF value is 0, the partial assignment made the CNF unsatisfiable. In this case, the FSM assigns the complementary value to its variable. If the CNF value is  $X$ , the partial assignment did neither satisfy the CNF nor did it make the CNF unsatisfiable. In this case, the FSM activates the next FSM at the bottom. If both value assignments have been tried, the FSM relaxes its variable by assigning  $X$  to it, and activates the previous FSM at the top. When the first FSM relaxes its variable and activates the global controller, the SAT problem is proven to be unsatisfiable. By this procedure, the array of interconnected FSMs implements chronological backtracking.



**Fig. 1.** Block diagram for the basic architecture (CE), consisting of an array of FSMs (#1 ... #n), a datapath, and a global controller. The variables  $x_i$  and the CNF are modeled in 3-valued logic.



**Fig. 2.** State diagram for an FSM of the architecture CE. The inputs are FT (from top) and FB (from bottom) that activate the FSM, and the 3-valued CNF. The output signals displayed inside the states are the variable value, and the signals TT (to top) and TB (to bottom) that activate the previous and next FSM.

### 3.2 Architecture CEDC

CEDC (CE + don't cares) extends CE by introducing don't care variables. Don't care variables are unassigned variables that appear only in clauses that are already satisfied. For example, the assignment  $(x_1 \leftarrow 1)$  for the CNF

$$(\bar{x}_1 \vee x_2) \wedge (x_1 \vee \bar{x}_3 \vee x_4) \wedge (x_2 \vee \bar{x}_4) \quad (2)$$

makes  $x_3$  a don't care variable. Don't care variables cannot change the CNF value and should not be selected for assignment. This strategy is similar to *clause-order backtracking* in software.

The don't care condition for a variable is a Boolean function of  $x_1, \dots, x_n$  and can be easily derived from the CNF. In the above example, the don't care condition for variable  $x_3$  is  $(x_1 \vee x_4)$ . In the architecture CEDC, the datapath computes the CNF value and the don't care conditions for all variables in parallel. The FSM accepts an additional input, the don't care condition for its variable. If the FSM is activated while this condition is set, it passes control directly to the next or previous FSM.

### 3.3 Architecture IM

IM (propagation of implications) exploits logical implications that are caused by value assignments. For example, the assignment  $(x_1 \leftarrow 1)$  for the CNF in Formula 2 implies the variable  $x_2$ , i.e.,  $x_2$  must be assigned 1 to satisfy the first clause. An implied variable can in turn imply other variables. An implication condition can be derived from the CNF for every literal. In the above example, the implication condition for  $x_2$  is  $(x_1 \vee x_4)$ . If both literals of a variable are implied, a contradiction has occurred. To model the variables, IM uses 4-valued logic with the values  $\{0, 1, X, C\}$ , where  $C$  denotes the contradiction.

An FSM in this architecture sets its variable according to value assignments or value implications. If a contradiction occurs, the FSM sets a local contradiction flag. The datapath takes as input the variables as well as the local contradiction flags and generates as output the implications for all literals of the CNF and a global contradiction flag in parallel.

Resolving logical implications in CNFs is known as the *unit-clause rule* and is the basic mechanism in the DP algorithm. The iterative application of the unit-clause rule is called *Boolean constraint propagation*. Using this method in instance-specific hardware was first proposed by [5]; a detailed description of the datapath and the FSM can be found in [6].

### 3.4 IMCE, IMDC

IMCE and IMDC are combinations of the previous architectures. IMCE combines propagation of implications with the evaluation of the CNF expression. This can be helpful in cases, where a partial assignment already satisfies the CNF, but the IM strategy continues to assign values to unassigned variables. IMDC combines propagation of implications with don't cares, and has potentially the most deductive power.

## 4 Simulation

In order to compare the different architectures we have implemented a program in C, that solves SAT problems by simulating the different hardware architectures. The simulator estimates performance and hardware cost. The performance is measured in number of visited levels in the search tree, number of value assignments, and number of clock cycles. The hardware cost is estimated in number of gates (NOT and 2-input AND/OR) and flip-flops (FFs).

In this paper, we report on simulation results for three benchmark classes from the DIMACS satisfiability benchmarks suite [7]: class *par* (instances from learning the parity function), class *jnh* (randomly generated instances), and class *hole* (instantiations of the pigeon hole problem). These classes are well-suited for evaluation, as they include examples with long software runtimes.

Table 4 presents the simulation results for the architecture IM. The speed-up  $S_{raw}$  is defined as  $t_{sw}/t_{hw}$ , the ratio of software and hardware execution times and does not include the hardware compilation time. The speed-ups in Table 4 are remarkably high and motivate solving SAT in instance-specific hardware. Similar speed-up numbers have also been reported in [5]. Excellent candidates for instance-specific hardware are SAT problems, where high raw speed-ups are combined with long software runtimes. The estimation further shows that for the targeted FPGA line (Xilinx XC4K), the combinational logic dominates the hardware cost. Although the estimation of hardware cost is not very accurate, most of the examples in Table 4 should fit into one FPGA.

The comparison of the different hardware architectures relative to each other revealed the following facts:

- The architectures can be divided into two groups, {CE, CEDC} and {IM, IMCE, IMDC}. Inside each group, the performance differences are below 1%.
- For classes *par* and *jnh*, IM performs at least 100 x better than CE; for class *hole*, IM performs about 10 x better than CE.
- The estimated hardware cost for IM is at most twice the cost for CE.

Hardware is used more efficiently by IM, as this architecture achieves with at most twice the hardware cost a performance at least 10 times better than CE. However, CE may be an option when the hardware resources are limited. Further, the performance measure counts clock cycles. Architectures that have more complex hardware designs will also have lower clock frequencies. CE is less complex than IM and will very likely lead to faster FPGA designs.

The results presented in this section depend strongly on the benchmark class. The exact trade-offs between the architectures in terms of performance and hardware cost must be evaluated for each new benchmark class.

## 5 Prototype Implementation

The design tool flow of our prototype implementation is shown in Figure 3 and consists of three parts: the front-end, the generator, and the back-end. The front-

<i>benchmark</i>	<i>variables</i>	<i>clauses</i>	$t_{sw}$ [s]	<i>simulated number of cycles</i>	$S_{raw}$ <i>at 10 MHz</i>	<i>hardware cost (Kgates)</i>
par16-1-c	317	1264	203.03	63171	32140	30 K
par16-1	1015	3310	321.25	158934	20212	87 K
par16-2-c	349	1392	3111.20	225408	138025	32 K
par16-2	1015	3334	1009.00	422295	23893	88 K
jnh16	100	850	2.11	20052	1052	26 K
jnh19	100	850	0.11	6432	171	26 K
hole7	56	204	4.56	351042	129	4.7 K
hole8	72	297	54.98	4342574	126	6.1 K
hole9	90	415	627.52	60162652	104	7.3 K
hole10	110	562	7616.40	922461250	83	9.7 K

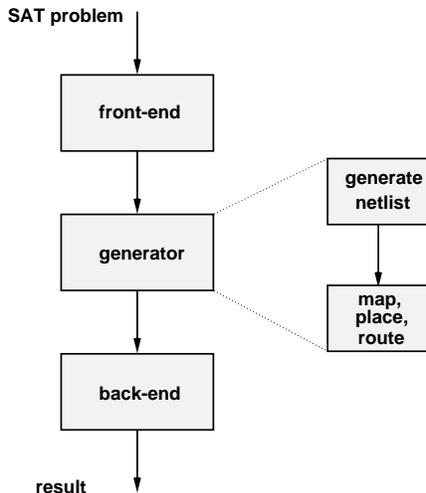
**Table 1.** Simulation results for examples from the DIMACS benchmark suite. The table shows the problem size in number of variables and clauses, the runtime of the software SAT solver GRASP, the simulated number of cycles for the IM architecture, the raw speed-up at an assumed clock frequency of 10 MHz, and the estimated hardware cost. GRASP was executed with parameters `+bD +dDLIS` on a Pentium-II/300MHz/128MB RAM PC platform running Linux.

end reads a SAT problem and checks for special cases, such as clauses that are always satisfied, reorders the variables and computes the assignment order. The generator compiles this modified SAT problem into a configuration bitstream for a Xilinx XC4K device. The back-end loads the bitstream onto the FPGA and waits for the end of the computation. If there is a solution, the back-end reads the FPGA register configuration, and extracts the variable values. The generator consists again of two blocks. The first block is the generation of the FPGA netlist, the second block invokes the Xilinx M1 design implementation tools for mapping, placement, and routing.

The architectures presented in Section 3 consist of the three blocks array of FSMs, datapath, and global controller. The global controller and the single FSM depend only on the chosen architecture, they do not change with the problem-instance. Therefore, these components are pre-designed, i.e., they are specified in Verilog HDL, synthesized and optimized by Synopsys FPGA Express II, and stored in a library as FPGA netlists. At hardware generation time, the required number of FSMs is instantiated and placed. Placement is done for two reasons: First, placing the FSMs allows the back-end to extract the result of the computation by the read-back facility of the FPGAs. Second, placement of the FSMs results in faster designs. The datapath depends totally on the problem-instance and is generated directly as FPGA netlist.

Our prototype is implemented on a PC platform running Windows NT4.0. As reconfigurable resource we use a Digital PCI Pamette board, which is equipped with 4 FPGAs of the type Xilinx XC4020. In our current experiments, we use only one of these FPGAs for implementing the SAT architecture.

The overall runtime for computing a SAT problem in hardware consists of the hardware compilation time,  $t_{comp}$ , the time for configuring the FPGA,  $t_{config}$ ,



**Fig. 3.** Prototype design tool flow.

the actual hardware execution time,  $t_{hw}$ , and the time for reading back and extracting the result,  $t_{read}$ .

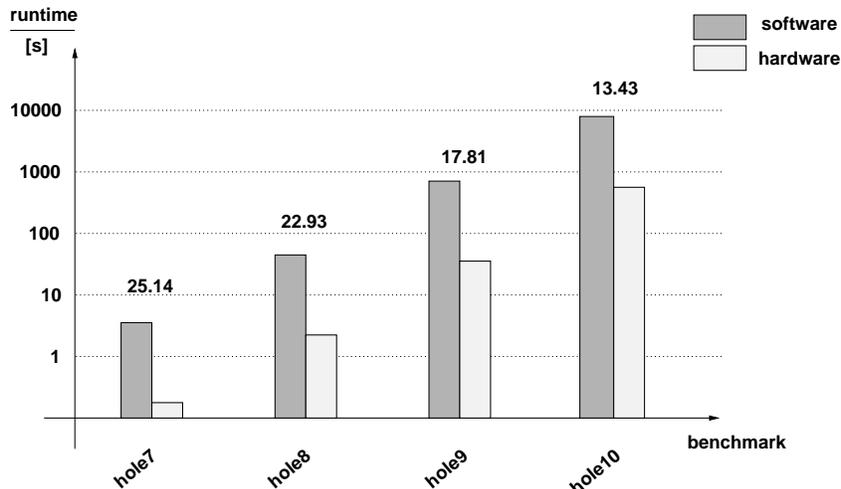
$$t_{overall} = t_{comp} + t_{config} + t_{hw} + t_{read} \quad (3)$$

The overall speed-up  $S_{overall}$  is then given by  $t_{sw}/t_{overall}$ . With our design tool flow, the times for FPGA configuration and read-back can be neglected compared to the hardware compilation time, which itself is strongly dominated by the Xilinx design implementation tools. At the time of this writing, we have successfully compiled and run CE architectures for the *hole* benchmark class. The examples *hole7* to *hole9* can be mapped onto one Xilinx XC4020, for *hole10* an FPGA of type XC4025/XC4028 is required. All the FPGA designs run at 20 MHz. Table 5 and Figure 4 present the experimental results

<i>benchmark</i>	$t_{sw}$ [s]	$t_{comp}$ [s]	$t_{hw}$ [s]	$S_{raw}$	$S_{overall}$
hole7	4.56	134	0.181	25.14	0.03
hole8	54.98	249	2.398	22.93	0.22
hole9	627.52	439	35.229	17.81	1.32
hole10	7616.40	597	567.255	13.43	6.54

**Table 2.** Experimental results for the CE architecture and the *hole* benchmark class. The table shows the GRASP runtime, the hardware compilation time, the hardware execution time, and the resulting speed-ups.

The results show that we could generate faster designs as assumed in Section 4. The execution times of hardware and software SAT solvers increase with the



**Fig. 4.** Software runtime  $t_{sw}$ , hardware runtime  $t_{hw}$ , and the resulting raw speed-up  $S_{raw}$  for the CE architecture and the *hole* benchmark class.

problem size more rapidly than the hardware compilation time. This leads to a *cross-over* point in the overall speed-up around *hole9*, i.e., here the SAT solver in reconfigurable hardware is for the first time faster than the software SAT solver. For *hole10* we achieve a speed-up of 6.54, which reduces the runtime from more than 2 hours in software to about 20 minutes in hardware.

For the *hole* benchmarks, the architecture IM requires about 10 times less clock cycles than CE. This would lead to an overall speed-up of 8.77 for *hole10*, assuming the same hardware compilation time than for CE. However, IM for *hole10* will not fit onto one FPGA XC4025.

## 6 Conclusion, Further Work

We have presented different architectures for solving SAT problems in instance-specific hardware. These architectures offer trade-offs between performance and hardware cost, depending on the benchmark class. Simulations revealed that for larger problems from the DIMACS benchmark suite, instance-specific SAT solvers can achieve significant raw speed-ups over software SAT solvers. We have implemented a prototypical design tool flow and discussed first experimental results. The results show that architectures with less deductive power can be competitive when the hardware compilation time is not neglectable compared to the hardware execution time. This is the case for all currently implemented benchmarks.

As the density of FPGAs increases, many interesting SAT problems can be accelerated by instance-specific hardware. Although FPGA-based computing machines still require relatively long compilation times, instance-specific archi-

techniques are promising for hard SAT problems, where software algorithms show a long runtime.

Further work includes:

- Implementation of instance-specific architectures for minimum-cost problems. A SAT problem with unit cost or integer cost values assigned to the variables forms a minimization problem. Finding minimum-cost solutions to SAT problems is a frequent task in CAD algorithms.
- Application of the instance-specific SAT solver to CAD tools. CAD applications have to be found, that generate SAT problems that are hard to solve in software, i.e., problems that have a relatively small number of variables but show long software runtimes.

## Acknowledgment

This work was partially supported by the Austrian National Science Foundation *FWF* under grant number J01412-MAT. We would also like to thank Alessandro Bogliolo and Luca Benini for their contributions and discussions in the early phases of this work.

## References

1. Jun Gu, Paul W. Purdom, John Franco, and Benjamin W. Wah. Algorithms for the Satisfiability (SAT) Problem: A Survey. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 35:19–151, 1997.
2. M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, (7):201–215, 1960.
3. M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Communications of the ACM*, (5):394–397, 1962.
4. J. Silva and K. Sakallah. GRASP – A New Search Algorithm for Satisfiability. In *IEEE ACM International Conference on CAD '96*, pages 220–227, November 1996.
5. Peixin Zhong, Margaret Martonosi, Sharad Malik, and Pranav Ashar. Implementing Boolean Satisfiability in Configurable Hardware. In *Logic Synthesis Workshop*, May 1997.
6. Peixin Zhong, Margaret Martonosi, Pranav Ashar, and Sharad Malik. Accelerating Boolean Satisfiability with Configurable Hardware. In *IEEE Symposium on FPGAs for Custom Computing Machines*, April 1998.
7. DIMACS satisfiability benchmark suite, available at <ftp://dimacs.rutgers.edu/pub/challenge/sat/benchmarks/cnf/>.
8. Takayuki Suyama, Makoto Yokoo, and Hiroshi Sawada. Solving Satisfiability Problems on FPGAs. In *International Workshop on Field-Programmable Logic and Applications (FPL)*, pages 136–145, 1996.
9. Miron Abramovici and Daniel Saab. Satisfiability on Reconfigurable Hardware. In *International Workshop on Field-Programmable Logic and Applications (FPL)*, pages 448–456, 1997.
10. Azra Rashid, Jason Leonard, and William H. Mangione-Smith. Dynamic Circuit Generation for Solving Specific Problem Instances of Boolean Satisfiability. In *IEEE Symposium on FPGAs for Custom Computing Machines*, April 1998.