Using Reconfigurable Computing Techniques to Accelerate Problems in the CAD Domain: A Case Study with Boolean Satisfiability

Peixin Zhong, Pranav Ashar, Sharad Malik and Margaret Martonosi
Princeton University and NEC CCRL
pzhong, sharad, mrm@ee.princeton.edu, ashar@ccrl.nj.nec.com

Abstract

The Boolean satisfiability problem lies at the core of several CAD applications, including automatic test pattern generation and logic synthesis. This paper describes and evaluates an approach for accelerating Boolean satisfiability using configurable hardware. Our approach harnesses the increasing speed and capacity of field-programmable gate arrays by tailoring the SAT-solver circuit to the particular formula being solved. This input-specific technique gets high performance due both to (i) a direct mapping of Boolean operations to logic gates, and (ii) large amounts of fine-grain parallelism in the implication processing. Overall, these strategies yields impressive speedups (>200X in many cases) compared to current software approaches, and they require only modest amounts of hardware. In a broader sense, this paper alerts the hardware design community to the increasing importance of input-specific designs, and documents their promise via a quantitative study of input-specific SAT solving.

1 Introduction

While hardware accelerators for CAD are not new, the increasing capacity and speed of field-programmable gate arrays (FPGAs) offers flexibility and computing power not available before. The hardware can be reused for different applications and can be easily updated to accommodate new developments in algorithms. This paper introduces an novel approach for accelerating the Boolean satisfiability (SAT) problem using configurable computing hardware. In our implementation of the SAT problem, we have harnessed this flexibility by implementing the SAT solver circuit on an "input-specific" basis. That is, a circuit is generated especially for each SAT formula to be solved. Since Boolean satisfiability approaches are logical-operation-intensive, the hardware approach can show large advantages by mapping the SAT expressions directly to logic gates, and by harnessing large amounts of parallelism in the evaluation of the logic.

While the bulk of the configurable computing applications currently implemented have been fairly simple data-centric systolic designs, our design shows the potential of custom computing on difficult problems with complex control structures. We have implemented a tree search algorithm, as opposed to the generally-simpler signal processing pipelines common to configurable computing thus far. In addition, we have also implemented a hardware version of nonchronological backtracking similar to that seen in software SAT solvers [11]. This requires complex control structures to manage the backtracking steps. These implementations demonstrate configurable computing's potential for impact on a much broader set of applications, particularly in CAD domain, than may initially have been considered.

2 The SAT Problem

The Boolean satisfiability (SAT) problem is a well-known constraint satisfaction problem with many applications in computer-aided design of integrated circuits, such as test generation, logic verification and timing analysis. 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 or contradiction 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 [4]. 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 contradiction 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 contradiction, 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 contradictions 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 entire regions of the search space corresponding to contradictory partial assignments. Every implied variable corresponds to one less free variable to branch on. Unfortunately, detecting impli-

cations in software is very slow. Each clause containing the newly assigned or implied variable is scanned and updated sequentially, with the process repeated until no new implications are detected.

Our intuition for hardware speedup potential in the SAT algorithm stems from recognizing that the implication procedure central to the algorithm is both highly parallelizable and easily mapped to basic logic gates. Our entire hardware architecture is designed to take advantage of this parallelism. Section 4 details our mapping of the algorithm onto reconfigurable hardware in a formula-specific manner.

Recent software implementations of the SAT algorithm have enhanced it in several ways while maintaining the same basic flow [8, 3, 12, 11]. The contribution of the GRASP work [11] is notable since it applies nonchronological backtracking and dynamic clause addition to prune the search space further. Very significant improvements in run time are reported. Recognizing nonchronological backtracking as an important feature, Section 5 shows how it can be mapped to reconfigurable hardware with significant improvements in run time.

3 The Promise of Configurable Computing

Special-purpose hardware accelerators for CAD problems are not new. This approach, however, usually suffers from long development time and high cost. The growth of field programmable logic devices allows application-specific hardware without the costs of fabrication. The SRAM-based programmable logic devices (such as Xilinx FPGAs) allow the same chip to be reprogrammed to different circuits an unlimited number of times. One successful application of such devices is logic emulation. A logic design is mapped to an FPGA array and simulated before it is actually fabricated. An emulator is many orders of magnitude faster than simulation on a general purpose computer.

The development of FPGAs has also led to the advent of field-programmable custom computing machines. The abundant programmable logic components and routing resources are used to form special-purpose computers exploiting fine-grain parallelism. Such custom computing machines have achieved very high performance in signal processing, genetic analysis, and cryptography applications. Previous work mostly concentrated on data-centric problems with relatively simple logic control.

Increasing density in programmable logic opens up exciting opportunities for hardware acceleration for complex computations. As feature sizes shrink even further, 1M-gate FPGAs will be feasible by roughly the year 2001 [10]. With programmability and high integration densities, we can create a machine specialized not only to the application but also to the input data associated with a specific problem.

Our choice of Boolean satisfiability (SAT) as a case study here was guided by two main factors:

- It is a search-intensive application, as opposed to data-intensive applications with simple control found in many previous studies of configurable computing. It is also a basic NP-complete problem that is core to many important problems. Gaining experience with this problem will help us accelerate a wide range of applications.
- It contains a significant amount of bit-level logic operations. This makes it very amenable for mapping to configurable hardware.

At this time, we know of three interesting proposals for solving SAT using reconfigurable hardware [13, 1, 9].

Suyama *et al.* [13] have proposed their own SAT algorithm - instead of the Davis-Putnam approach - for implementation in reconfigurable hardware. The search uses full assignment to explore the space. In their paper, they acknowledge that this is not as efficient as a backtrack search algorithm.

The proposal by Abramovici and Saab [1] is also not an implementation of the Davis-Putnam algorithm. Their proposal basically amounts to an implementation of a PODEM-based [6] algorithm in reconfigurable hardware. Therefore it is difficult to handle a generic CNF formula directly. On the other hand, our approach is more general and can exploit heuristics not available to PODEM.

Rashid *et al.* [9] implement small instances of the PODEM algorithm on a Xilinx XC6216. While mapping and compilation issues are presented in detail, there are currently no performance results for their approach.

4 FPGA Mapping: Basic Backtrack Search

4.1 Hardware Organization

Our hardware implementation has two parts: (i) the implication circuit (ii) a state machine to manage the backtrack search. Given a SAT formula, a VHDL description of the hardware is generated automatically.

The speedup over software arises from our implementation of the implication circuit. It finds all direct implications of newly-assigned or newly-implied variables in a single clock cycle. Consequently, all transitive implications of a new variable assignment can be determined in a few cycles. Since all clauses are examined in one cycle, parallelism is quite considerable in practice. For the aim-100-6_0-yes1-1 example in the DIMACS benchmark suite, this approach yields an average of 41.9 effective clause evaluations per cycle.

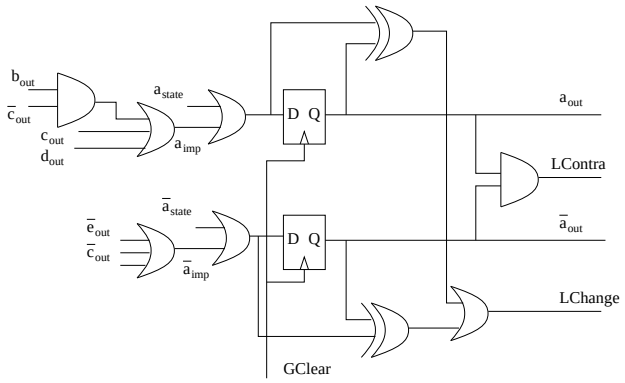

Figure 1: Circuit for implication and conflict detection.

Figure 1 shows the details of the implication circuit. The state of each variable in the circuit is encoded into two bits, $(v, \bar{v})$. A free literal is $(0, 0)$. The 1 value of the variable is represented by $(1, 0)$ and the 0 value by $(0, 1)$.

Implications are easy to determine. For a clause with n literals, if n-1 literals are 0, the last literal is implied to 1. In our encoding, given the clauses $(\bar{c} + a)(\bar{d} + a)(\bar{b} + c + a)(e + \bar{a})(c + \bar{a})$, a is implied by $c + d + b\bar{c}$ and $\bar{a}$ is implied by $\bar{e} + \bar{c}$, as shown in Figure 1. If both v and $\bar{v}$ are implied or assigned to be true, there is a contradiction. No solution exists with this partial assignment.

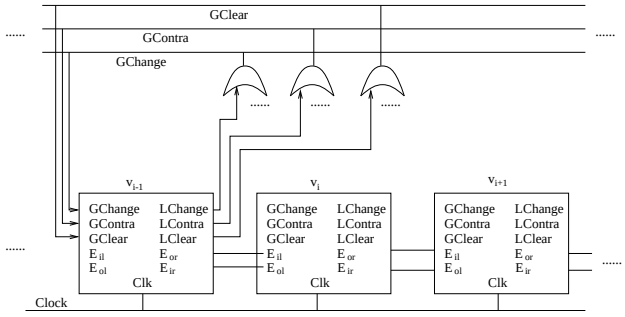

Figure 2: Global Circuit Topology

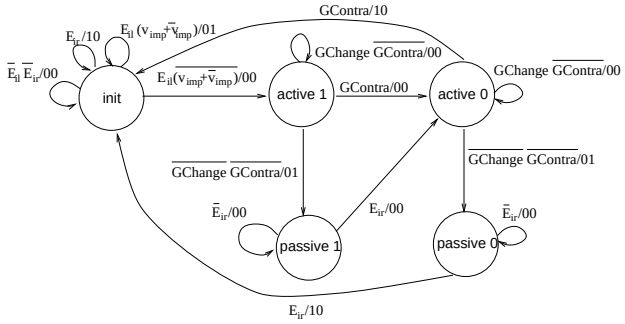

Figure 3: State Diagram for Backtracking Machine

The transitive implication may include cyclic loops. Since the implication is monotonic, changing from 0 to 1 due to our 2-bit encoding, it will never cause oscillation. A D flip-flop is used with each literal, so new implications will propagate one level in one cycle. The LChange signal checks new value changes. Implications finish when no new changes are found.

In order to avoid heavy communication with a host computer, search control is also implemented in hardware. A state machine is implemented for each variable and they are connected according to the order of search, as in Figure 2. Each box in the figure contains the implication circuit and the state machine for each variable. This distributed topology keeps global signals to a minimum and reduces hardware costs. At any instant, only one state machine is in control. Once that state machine has finished processing, it asserts E_{or} to transfer control to the state machine on the right (if progressing forward in the computation) or it asserts E_{ol} to pass control to the left (if backtracking). Each state machine is aware of whether its variable has been assigned, implied or is free.

The state machine for a single variable is shown in Figure 3. The five states in the state machine are encoded by three bits. Two bits correspond to the values of the positive and negative literals of its variable. The third bit indicates whether this particular state machine is active. The inputs to each state machine are the Enable signals from its left and right neighbors, and the global contradiction (GContra) and change (GChange) signals. State machine outputs are the enable signals, E_{ol} and E_{or} that pass control to the left or right.

System operation is fairly simple. After initialization, all the state machines are in the `init` state. When a state machine receives an enable signal, it changes its states. If control is transferred from the left, it can only be in the `init` state. If this variable's value is not implied, it asserts value 1 (`active 1`) and determines all the transitive implications of that assignment. If a contradiction is detected, it will try the value 0 instead (`active 0`) and repeat the implication step. If the implication settles without a contradiction, it transfers control to the state machine on its right and it transitions to the `passive 1` state. From `active 0`, if a contradiction is detected, it will backtrack. Its state is reset to `init` and the control is passed to the left. On the other hand, if implications settle without a contradiction, then it moves from `active 0` to `passive 0` and passes the control to the right.

If, as above, control is transferred to a state machine from the left, but its variable's value has already been implied by a previous assignment, then it merely passes control to the state machine on its right on the next clock cycle and remains in the `init` state. Its output value will be maintained by the implication.

If control is transferred to a state machine from the right, the system is backtracking. If the variable has been implied by some previous assignment, the state machine is in the `init` state now. It simply passes control to the variable on its left. If the variable currently has the assigned value 1 (`passive 1`), then it is changed to 0 (`active 0`). The implications are computed. If the variable already has the assigned value 0, we have already tried both possible values for this variable. In this case, the state machine resets to `init` and backtracks further by transferring control to the left.

Finding a Solution A solution has been found when the rightmost state machine further attempts to pass control to the right. No solution exists if the leftmost state machine attempts to backtrack by further passing control to the left.

4.2 Performance Results

In this section we compare the performance of our hardware implementation of the basic Davis-Putnam SAT algorithm to its implementation in GRASP [11]. In order to estimate the hardware performance, we use simulation to count the exact number of hardware clock cycles needed to solve each problem and divide by clock rate to compute the run time. We have used a VHDL model to verify design correctness on moderate examples. However, since VHDL simulation can be slow, we have generated a C-language hardware model that is significantly faster then general VHDL simulation. We use this for performance estimation of long-running problems. Like a VHDL simulation, the C-language simulation is also capable of counting the number of hardware clock cycles required exactly.

The number of hardware clock cycles is translated into run time based on a clock rate of 1.33 MHz. This is a clock rate achieved on a large example on an IKOS FPGA board [7] as detailed in the section on hardware implementation issues. The GRASP runs were timed on a Sun 5 workstation with 64 MB of RAM and a 110 MHz processor.

As discussed earlier, our hardware implementation currently requires variables be ordered statically prior to the hardware implementation. Our ordering strategy places earlier the variables with more appearances in the formula.

We compared the performance for all the examples in the DIMACS SAT benchmark suite [5]. There are 240 problems in the DIMACS suite. Some of them take a very long time to solve and neither GRASP nor our C-model simulation gets

a result in reasonable amount of time (several hours). There are 134 instances in which at least one program finished.

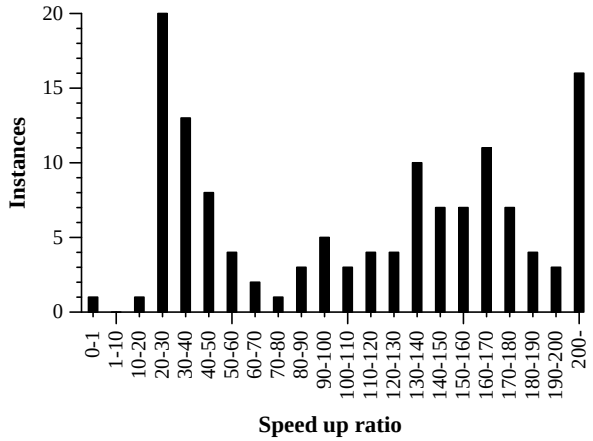

Figure 4: Speedup ratio histogram comparing our FPGA approach to a GRASP run, both with basic backtracking.

In all examples for which the implementation of the basic Davis-Putnam algorithm completes in GRASP, our hardware implementation also completes — usually in much shorter time. Figure 4 shows a histogram in which each bar corresponds to a speedup range. Speedup is defined as the ratio of the GRASP run time to the hardware run time. (That is, a 10X speedup means that our approach is ten times faster than GRASP.) The height of the bar corresponds to the number of DIMACS SAT examples for which the corresponding speedup is obtained.

The histogram indicates that *more than 90% of the examples have speedup greater than 20X* and *more than 45% of the examples have speedup greater than 100X*. The results clearly demonstrate that utilizing parallelism by direct logic mapping in the SAT algorithm achieves significant speedup even when the hardware clock is much slower than that of the general purpose computer.

5 FPGA Mapping: Nonchronological Backtrack

5.1 Algorithm

When the Davis-Putnam algorithm backtracks to the most recently assigned variable, its backtracking is said to be *chronological*. On the other hand, when an algorithm jumps over several previously-assigned variables to a variable more than one level above the current variable, the backtracking is said to be *nonchronological*. In order to jump directly to a previous level, the algorithm must first determine that no combination of values on the skipped variables will result in a satisfying assignment. GRASP is a recent SAT implementation with nonchronological backtracking. The GRASP work demonstrated that nonchronological backtracking can lead to significant reductions in run time. We describe an algorithm to achieve a similar goal in hardware, but with less control complexity than GRASP.

GRASP maintains a data structure called the implication graph, from which one derives the transitive implications leading to the contradiction. From the contradiction, it traces back to a set of assignments contributing to the contradiction. It jumps to the most recently assigned variable in the set. It is difficult, however, to traverse the implication

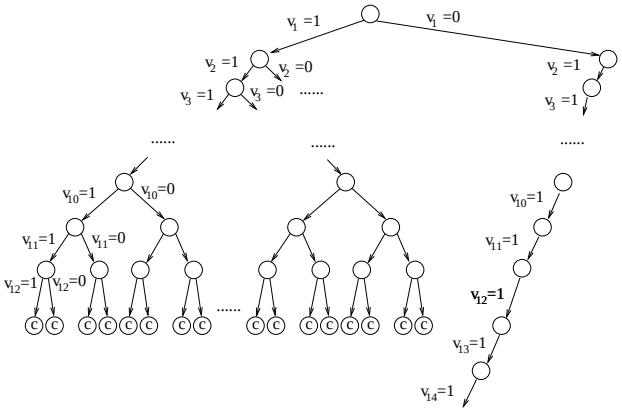

Figure 5: Search example for basic backtracking.

circuit in hardware. As a result, we devised the following alternative.

Rather than analyzing the implications, we take advantage of the fact that determining implications is very fast in our hardware. When a contradiction occurs for both assignments of the variable, it should backtrack. If we reset this variable to be free, the observed contradiction will disappear, but may recur when the same variable is assigned a value in the future. We want to change the value of the variable that really contributes to the observed contradiction.

If a contradiction was detected for both assignments to v_i, with i being its level, our procedure works its way back up the levels, one at a time. At each level j ($< i$), the procedure calls the implication routine twice, once for each v_i value, while v_j has a flipped value. The variables v_k between i and j ($j < k < i$) are left unassigned when the implication routine is called, i.e. both literals v_k and $\bar{v}_k$ are set to 0. The algorithm must backtrack to v_j if a no-contradiction case was found for one of the two assignments. If both cause contradiction, the variable v_j can be skipped, and the procedure repeats this step for variable v_{j-1}.

This procedure requires only $2n$ calls to the implication routine if the algorithm reverts back by n levels. Since the procedure does not call the implication routine an exponential number of times, and since the implication hardware is very fast, this analysis for nonchronological backtracking can be expected to be very fast also. In comparison, GRASP has the overhead of maintaining the implication graph data structure and analyzing it when a contradiction is found, but it does not do the $2n$ implications.

The following simple example shows how our approach works. Suppose we have the following formula: $(v_1' + v_{12}' + v_{13})(v_1' + v_{12}' + v_{13}')(v_1' + v_{12} + v_{14})(v_1' + v_{12} + v_{14}') \cdots$. Assume that no assignment for variables $v_2 - v_{11}$ leads to a contradiction. The basic search tree for this formula is shown in Figure 5. In the figure, each solid arrow represents assigning a new value and determining its implications. It begins by assigning $v_1 = 1$ and goes on until $v_{11} = 1$. Finally, $v_{12} = 1$ is tried. This leads to a contradiction since v_{13} is required to be both 0 and 1 simultaneously. Similarly, $v_{12} = 0$ also leads to a contradiction. The normal backtrack procedure would reset v_{12} to unknown and it would backtrack to v_{11}. The conflict disappears temporarily but will appear again whenever v_{12} is set.

Figure 6(a) shows our version of nonchronological backtracking. Figure 6(b) shows how GRASP directly backtracks to v_1. The dotted arrows indicate skipping and do

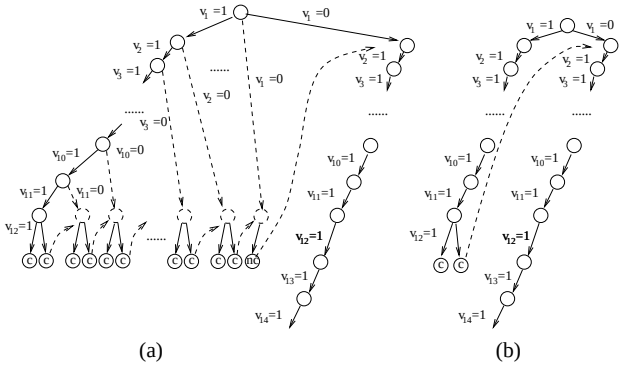

Figure 6: Search examples for nonchronological backtracking. (a) Algorithm in our hardware algorithm. (b) Software algorithm used in GRASP.

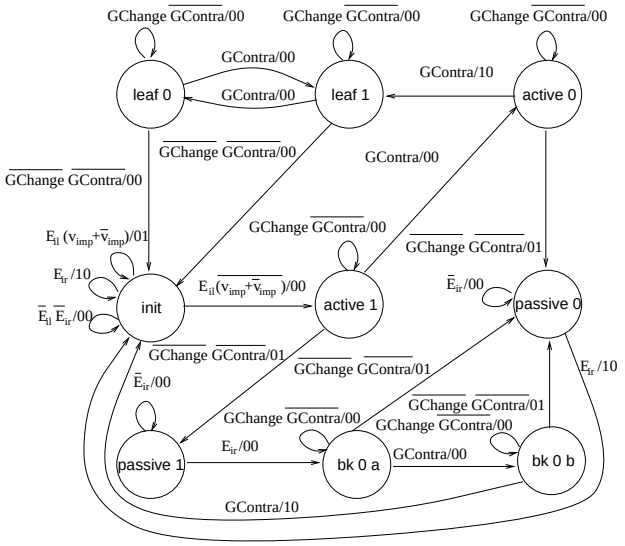

Figure 7: State diagram for nonchronological backtracking machine

not incur any computation; they simply show what values are assumed before an implication calculation. For both methods, the search is the same before reaching v_{12}. Backtracking differs, however. When backtracking at each level, our implication procedure is called twice. That is, we backtrack to v_1 after effectively calling the implication procedure only 22 times, rather than 4082 attempts with the basic backtracking approach. While this simple example is mainly pedagogic, dramatic speedup potential also exists in real benchmarks. For example, in the aim-100-1_6-yes1-1 benchmark from the SAT suite, chronological backtracking visits 606,578 partial assignments; the nonchronological backtracking algorithm only visits 1384 partial assignments.

5.2 Hardware Organization

The only modifications required to implement our nonchronological backtracking algorithm are to the state machine. No new global signals are added and the implication circuit is not modified.

The state machine to perform the nonchronological backtracking is shown in Figure 7. It has more states and transitions to reflect the complex backtracking procedure outlined

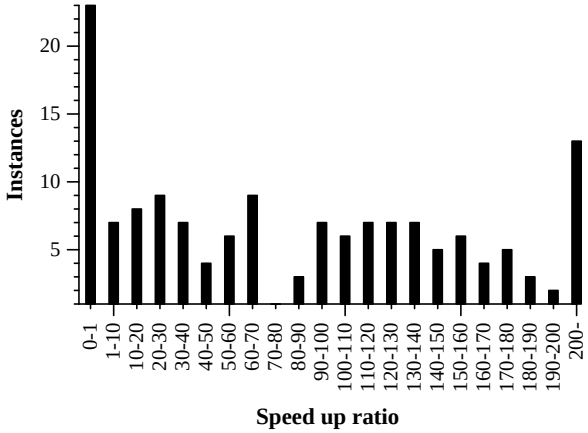

Figure 8: Speedup ratio histogram for nonchronological backtracking

previously. Although the state machine has become bigger, the extra states require only one more encoding bit. Since most of the circuit is the same and since the state machine is not instrumental in determining the clock cycle time, the cycle time from the basic algorithm remains valid here.

5.3 Performance Results

We compare our runtime with nonchronological backtracking to a GRASP implementation with nonchronological backtracking. Similar to Figure 4, the results are graphed in Figure 8. Due to the improved performance, there are 149 problems that finished with either GRASP of our approach.

While GRASP's more complicated nonchronological backtracking allows it to garner a bigger improvement from this technique than our hardware does, we still obtain a median speedup over GRASP of 63.5X. From the full DIMACS suite, we offer 100X or greater speedups on 63 of the examples.

6 Implementation Issues

6.1 Envisioned Usage

Depending on the problem characteristics, the configurable computing implementation can often offer significant speedups over a software SAT solver. In order for the acceleration to be useful, however, it must offer performance advantages even after hardware compile time and configuration time are considered. For this reason, we envision that the configurable hardware techniques will mainly be used on SAT problems with very long GRASP run-times (hours or days) or in cases where GRASP aborts. In such cases, the hardware synthesis times required will be acceptable. Furthermore, current commercial CAD tools for FPGAs are quite general and therefore quite slow in this usage; it is not uncommon for synthesis and place-and-route times to exceed one hour. Since our designs are quite regular and easy-to-route, however, compiler tools specialized for our template design could be much faster, and could thus broaden the range of problems for which hardware acceleration is useful. Overall, we envision a system where easy SAT problems are still solved in software. Very large problems, ones that often timeout today, will invoke the hardware compiler and configure an FPGA board to assist in solving them.

The envisioned hardware platform is a high-performance workstation with an FPGA board attached via an exter-

nal connection or the I/O bus. Easy problems are handled by software running on the workstation itself, while longer-running problems are handled by the FPGA board. Requirements for the FPGA board itself are partly a function of the hardware size of the problems being solved; this is discussed in the following subsection.

6.2 Hardware Resources

Hardware requirements for our approach are a function of the size and complexity of the formula being solved. The hardware usage is estimated in terms of CLBs of Xilinx XC4000 series FPGAs.[14]. The SAT solver's CLB requirements varies widely with the formula to be solved. Across the DIMACS benchmarks, the median CLB requirements are 3655. Overall, relatively few (less than 30%) of these problems will fit within the about 2000 CLB limit of current FPGA chips. However, 202 of 240 problems require fewer than 20000 CLBs and should fit on fewer than ten FPGAs.

Partitioning the design across FPGAs is straightforward because of the regular topology used. In our experiments, we have used the partitioning and pin multiplexing techniques in the Ikos SLI logic emulation system originally developed as part of the MIT Virtual Wires effort [2, 7]. This partitioning software can multiplex several inter-chip signals to use the same physical pins, thereby circumventing the pin limitations that can often limit the CLB utilization of FPGA designs. Furthermore, the software performs this multiplexing with minimal impact on hardware cycle times. We have compiled many of the DIMACS problems for the IKOS system. Their clock rate ranges from 700KHz to 2 MHz. We use a 1.33 MHz clock rate as a typical value for performance comparison. If SAT is implemented on a directly-connected array, the clock rate is expected to be 10 to 20 MHz.

7 Future Work

As evidenced by the organization of our paper, we began our work by implementing the basic Davis-Putnam procedure in reconfigurable hardware. Recent software implementations of the Davis-Putnam procedure have grown to be very sophisticated however [11]. To beat them consistently, our implementation should match that sophistication. We zeroed in on nonchronological backtracking as a key requirement and have been able to implement it with a small increase in hardware. The GRASP implementation still contains two additional features that our implementation does not:

1. The ability to add clauses on the fly plays a significant role in GRASP's ability to solve hard problems. Variable relationships derived in the search help prune search tree in the future. It is hard to change the circuit and reroute the FPGA in the middle of the algorithm. We are working on low cost ways of implementing this feature based on multiple configuration contexts.

2. The ability to choose the next decision variable on the fly also contributes to the efficiency of the software implementations. While this is very simple to do in software, it can be very hardware intensive in reconfigurable hardware, as evidenced by the Abramovici and Saab proposal [1]. In practice, we have found that a good static ordering of variables works quite well. If dynamic ordering can provide significant improvement, we will implement it as well.

8 Conclusions

Overall, the contributions of this paper are two-fold. First, we provide a system design for formula-specific Boolean satisfiability solutions based on a configurable hardware implementation. Our design's hardware requirements are quite modest compared to other recent proposals. Moreover, our hardware performance results indicate that the configurable hardware approach offers dramatic improvements over even the best current software-based techniques. We have observed speedups of well over 200X compared to software SAT solutions on several of the DIMACS SAT benchmarks.

The second contribution of this paper is much broader. We present our SAT-solver as a case study of a class of *input-specific* configurable hardware applications. These applications aggressively harness the increasing integration levels and reconfigurability of current FPGAs by performing template-driven hardware designs for each problem/data set being solved. Their amenability to parameterized, automated design makes it easy and fast to compile configurations for them. Overall, this paper alerts the hardware design community to an increasingly-important application style, and documents its significant promise via a detailed case study on formula-specific SAT-solving.

References

[1] M. Abramovici and D. Saab. Satisfiability on Reconfigurable Hardware. In *Seventh Intl. Workshop on Field Programmable Logic and Applications*, Sept. 1997.

[2] J. Babb, R. Tessier, and A. Agarwal. Virtual Wires: Overcoming pin limitations in FPGA-based logic emulators. In *Proc. IEEE Workshop on FPGA-based Custom Computing Machines*, pages 142–151, Apr. 1993.

[3] S. Chakradhar, V. Agrawal, and S. Rothweiler. A transitive closure algorithm for test generation. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 12(7):1015–1028, July 1993.

[4] M. Davis and H. Putnam. A Computing Procedure for Quantification Theory. *Journal of the ACM*, 7:201–215, 1960.

[5] DIMACS. DIMACS Challenge benchmarks and UCSC benchmarks. Available at ftp://Dimacs.Rutgers.EDU/pub/challenge/sat/benchmarks/cnf.

[6] P. Goel. An Implicit Enumeration Algorithm to Generate Tests for Combinational Logic Circuits. *IEEE Tran. on Computers*, C30(3):215–222, March 1981.

[7] IKOS Systems. Virtual Logic SLI Documentation. Version 1.6.

[8] T. Larrabee. Test Pattern Generation Using Boolean Satisfiability. In *IEEE Trans. on Computer-Aided Design*, volume 11, pages 4–15, January 1992.

[9] A. Rashid, J. Leonard, and W. H. Mangione-Smith. Dynamic Circuit Generation for Solving Specific Problem Instances of Boolean Satisfiability. In *Proceedings IEEE Workshop on FPGA-based Custom Computing Machines*, Apr. 1998.

[10] J. Rose and D. Hill. Architectural and Physical Design Challenges for One Million Gate FPGAs and Beyond. In *Proc. 1997 ACM/SIGDA Fifth Intl. Symp. on Field-Programmable Gate Arrays*, Feb. 1997.

[11] J. Silva and K. Sakallah. GRASP-A New Search Algorithm for Satisfiability. In *IEEE ACM Intl. Conf. on CAD-96*, pages 220–227, Nov. 1996.

[12] 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.

[13] 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.

[14] Xilinx Corp. The Programmable Logic Data Book. Xilinx Corp. San Jose, CA, 1996.