Accelerating Boolean Satisfiability with Configurable Hardware

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

Department of Electrical Engineering, Princeton University
*NEC CCRL
{pzhong, mrm, sharad}@ee.princeton.edu, ashar@ccrl.nj.nec.com

Abstract

This paper describes and evaluates methods for implementing formula-specific Boolean satisfiability (SAT) solver circuits in configurable hardware. Starting from a general template design, our approach automatically generates VHDL for a circuit that is specific to the particular Boolean formula being solved. Such an approach tightly customizes the circuit to a particular problem instance. Thus, it represents an ideal use for dynamically-reconfigurable hardware, since it would be impractical to fabricate an ASIC for each Boolean formula being solved. Our approach also takes advantage of direct gate mappings and large degrees of fine-grained parallelism in the algorithm's Boolean logic evaluations.

We compile our designs to two hardware targets: an IKOS logic emulation system, and Digital SRC's Pamette configurable computing board. Performance evaluations on the DIMACS SAT benchmark suite indicate that our approach offers speedups from 17X to more than a thousand times. Overall, this SAT solver demonstrates promising performance speedups on an important and complex problem with extensive applications in the CAD and AI communities.

1 Introduction

For many problems, computing machines based on configurable hardware can provide significant performance improvement compared to conventional microprocessors. However, almost all of the prior work in this area targets relatively simple kernel algorithms or accelerates small segments of code. With the increase of FPGA logic capacity and availability of hardware systems with large number of FPGAs, it is now possible to map much more complicated problems to configurable hardware.

This paper presents our application of configurable computing to Boolean satisfiability (SAT). This is a core computer science problem which has important applications in CAD, AI and other computing subfields. Since SAT is an NP-complete problem, SAT solvers involve complicated control logic and time-consuming computation. Our approach takes advantage of FPGAs both for their configurability and their high logic capacity, to realize SAT solvers with massive, fine-grained parallelism.

We use the SAT problem as a case study to demonstrate a methodology to map complicated problems to configurable computing. Our configurable computing approach for SAT-solving can offer substantial speedups over traditional software approaches. Starting from a general, template design, we specialize the accelerator design to the particular SAT formula being solved. This allows us to even more tightly match the configurable hardware to the specific problem instance being solved. In our approach, the encapsulating control software creates a hardware description language (HDL) file automatically from the input formula and the template design. The file is then compiled to an hardware implementation. That resulting hardware will be used to solve the particular problem. Note that this formula-specific approach is *only* possible on reconfigurable hardware, since it would be impractical to fabricate an ASIC for each Boolean formula being considered.

Our project has led to a number of observations regarding FPGA-based accelerators for CAD problems and for applications in general. Most importantly, while many configurable computing applications currently implemented have been fairly simple data-centric systolic designs, our design helps demonstrate the potential for complex control structures. We have implemented a full tree search algorithm, with complex backtracking techniques. This implementation highlights configurable computing's potential for impact on a much broader set of applications than are often considered.

To demonstrate our design concretely, we compile problems for execution on an IKOS logic emulator [10] and Digital's Pamette board [13]. The IKOS system, with its large capacity and efficient partitioning, is particularly helpful in allowing us to compile very large, "real-life" problems. With these widely-available systems, our approach yields significant speedups (from 17X up to more than one thousand times) against the software-only solutions running on general-purpose workstations. Since we use a formula-specific approach, hardware compilation time must also be a consideration; when compile time is also taken into account, we still achieve speedups of 8X or more on difficult problems. It is crucial to note that many current SAT problems can take days to run in software; even with disappointing place-and-route times for current configurable hardware, our approach holds great promise. Furthermore, our regular, template-based design will be easily routable as faster place-and-route techniques become more common.

The remainder of this paper is structured as follows. Section 2 describes the SAT problem and its basic algorithm. Section 3 explains our methodology for mapping the problem to configurable hardware. Section 4 briefly discusses the hardware platforms we are using. In Section 5, we talk about the setup and the results of our experiments, including performance comparison and hardware usage. Section 6 discusses related work. Sections 7 offers discussion on configurable hardware and our future work. Finally, Section 8 gives conclusions.

2 The SAT Problem

2.1 Motivation

Increasing density in programmable logic opens up exciting opportunities for hardware acceleration for CAD problems. As feature sizes shrink even further, 1M gate FPGAs will be feasible by roughly the year 2001 [12]. With programmability *and* high integration densities, we can create input-specific programmable solutions with previously-infeasible complexities. Our choice of SAT as a case study here was guided by two main factors:

- It is a search intensive application, as opposed to applications that compute intensively on a uniform data stream. Prior usage of configurable computing has often focused on the latter; the programmable logic frequently consists of a systolic array style or very deep pipeline data paths. Our new case study offers insight into how configurable computing can be best used for algorithms with complex control.
- The SAT solver uses a significant amount of bit-level operations. This aspect directly exploits the bit-level logic parallelism offered by programmable logic and thus holds great promise for computational acceleration and efficient hardware usage.

2.2 Problem Overview

The Boolean satisfiability (SAT) problem is a well-known subset of constraint-satisfaction problems. It is one of the basic NP-complete problems. It has many applications in computer-aided design of integrated circuits, such as test generation, logic verification and timing analysis. Given a Boolean formula, the objective is either to find an assignment of 0-1 values to the variables so that the formula evaluates to true, or to establish that such an assignment does not exist.

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. An n-clause is a clause with n literals. For example, $(v_i + \bar{v}_j + v_k)$ is a 3-clause. In order for the entire formula to evaluate to 1 each clause must be satisfied, *i.e.*, evaluate to 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. For example, an assignment of $v_i = 1$ would satisfy $(v_i + \bar{v}_j + v_k)$, while $v_i = 0$ leaves the clause 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. For example, in the above clause, the partial assignment $v_i = 0, v_k = 0$ implies v_j must be 0. A variable that is not assigned or implied is considered *free.* A conflict or *contradiction* arises if the value implied to a variable is different from the value previously implied or assigned to that variable. Detection of a conflict implies that the corresponding partial assignment cannot be a part of any valid solution.

Most current software SAT solvers are based on the Davis-Putnam algorithm [5] illustrated in Figure 1. This is a backtracking search algorithm. The basic algorithm begins from an empty partial 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 during the implication procedure, the algorithm picks the next free variable, and repeats the procedure. Otherwise, the algorithm attempts a new partial assignment by complementing the most recently assigned variable for which only one value has been tried so far. This step is called *backtracking.* The algorithm terminates when: (i) no free variables are available and no contradictions have been encountered, or (ii) all possible assignments have been exhausted. Case (i) indicates a solution has been found. Case (ii) indicates that no solution exists for this formula.

```
Initialize all variables to "free" value
do {
   Compute implications and check contradiction;
   if (contradiction)
     if (active_variable->assigned_value == 1)
        active_variable->assigned_value = 0;
      else
        backtrack();
      endif
   else
      active_variable = next_free_variable();
      active_variable->assigned_value=1;
   endif
} while ()
```

Figure 1: Pseudo-code for basic search algorithm.

Determining implications is crucial to pruning the search space since (1) it allows the algorithm to skip entire regions of the search space corresponding to contradictory partial assignments, and (2) every implied variable corresponds to one less free variable on which search must be performed. Unfortunately, detecting implications in software is very slow since each clause containing the newly assigned or implied variable is scanned and updated sequentially, with the process repeated until no new implications are detected.

Thus, configurable hardware's speedup potential in the SAT algorithm stems from the fact that the implication procedure central to the algorithm is both (i) highly parallelizable and (ii) easily mapped to basic logic gates. Our entire hardware architecture is designed to take advantage of this parallelism. Section 3 details our mapping of the Davis-Putnam algorithm onto reconfigurable hardware in a formula-specific manner.

Recent software implementations of the Davis-Putnam algorithm have enhanced it in various ways [4, 11, 14, 15] while maintaining the same basic flow. These enhancements may significantly improve the performance. We have had preliminary success on implementing some of these improvements [18]. Due to space limitations, this paper will concentrate on the methodology of configurable computing solutions for SAT problem and we will only explain the implementation of the basic algorithm in detail.

3 Hardware Mapping of Backtracking Search Algorithm of SAT

3.1 Hardware Organization

Analogous to software, our hardware implementation of the Davis-Putnam algorithm has two parts: (i) the implication circuit, and (ii) state machines to manage the backtracking search. Given a SAT formula, both hardware modules are generated automatically.

The speedup over software arises from our implementation of the implication circuit. Unlike software, this circuit finds all direct implications of newly-assigned or all newly-implied variables in a single clock cycle. Consequently, all transitive implications of a new variable assignment can be determined in a small fraction of the clock cycles required by software.

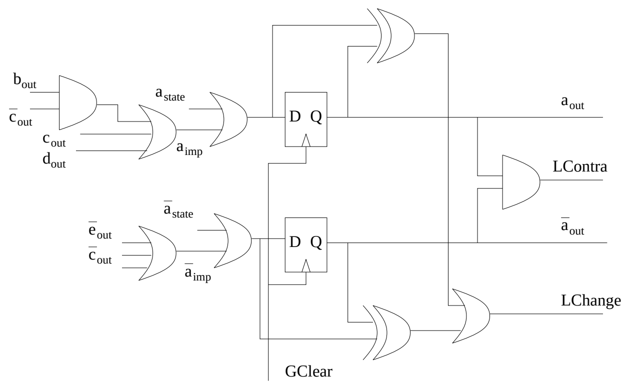

Figure 2: Circuit for implication processing and conflict detection.

3.1.1 The Implication Circuit

Figure 2 shows the details of a portion of the implication circuit. The state of each variable in the circuit is encoded into two bits by the values of its literals, $(v, \bar{v})$. If it has the value $(1,0)$ or $(0,1)$, the variable v is understood to have the value 1 or 0, respectively. For the state $(0,0)$, the variable v has no value assigned or implied; it is free. Finally, if $(v, \bar{v})$ takes the value $(1,1)$, we have a contradiction since the variable v cannot take both the values 1 and 0 simultaneously.

The role of the implication circuit is to determine the implied value of each literal and detect the contradiction from an assignment. A literal evaluates to 1 if it is either implied to 1 or assigned to 1. Consequently, its complement will be 0. In the circuit, the value of each literal is, therefore, the Boolean OR of its implied and assigned values. A contradiction arises when both literals of any variable become 1. A global contradiction is the sum of all the local contradictions.

In order to determine the implied value of a literal, the circuit must encapsulate the relationships between variables arising from the clauses in the CNF formula. For example, given the clauses $(\bar{c} + a)(\bar{d} + a)(\bar{b} + c + a)(e + \bar{a})(c + \bar{a})$, we can conclude that if $c = 1$, or if $d = 1$, or if $b = 1$ and $c = 0$, a must take on the value 1, *i.e.*, a is implied to 1. Similarly, if $e = 0$ or $c = 0$, a is implied to 0. Our encoding allows efficient implementation of implications. In this example, the equation for the implied value of literal a would be $a_{imp} = c + d + b\bar{c}$, as shown in Figure 2. Similarly, the equation for the literal $\bar{a}$ would be $\bar{a}_{imp} = \bar{e} + \bar{c}$.

Even the small example above has a loop in the implication dependencies, since the values of literals $\bar{a}$ and $\bar{c}$ are dependent on each other. It will not cause oscillation because no inverter exists in the circuit. However, we felt this might complicate the timing analysis because the path length can not be determined.

To avoid these problems, we propose a circuit in which each literal is latched by a clocked D flip-flop. During each clock cycle, the circuit determines in parallel the direct implications of all the variables that were assigned or implied in the previous cycle. The procedure completes when no new implication appears during a clock cycle. New implications are detected by taking the difference of the input and output of the D flip-flop for each literal. This parallel implication is much faster than in software. Profiling one typical problem (aim-100-6_0-yes1-1) showed an average of about 42 clauses are evaluated in one hardware clock cycle.

3.1.2 Global Circuit Topology and the Backtrack State Machine

In addition to the implication processing, there should be a control unit to maintain the backtrack search. Though it is easy to design a centralized control unit on paper, the large number of variables (more than one hundred for interesting problems) means that such a unit may be difficult to implement on configurable hardware. We instead have designed a novel distributed control system for this application.

Two important features of our overall circuit organization are that (i) a separate state machine is implemented for each variable, and (ii) the state machines are connected with nearest neighbors in a linear array. The order of the variables is determined a priori. Our circuit topology takes the form shown in Figure 3. Each box in the figure contains the literal computation 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 searching forward) 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.

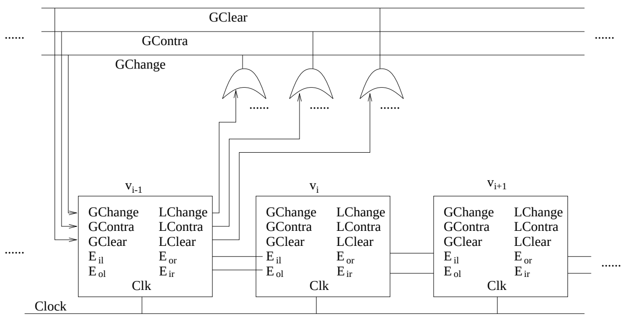

Figure 3: Global circuit topology

The state machine for a single variable is shown in Figure 4. 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), change (GChange) and clear (GClear) signals. State machine outputs are the enable signals, E_{ol} and E_{or} that pass control to the left or right as described below.

Forward Computation: After initialization, all the state machines are in the `init` state. When a state machine receives an enable signal, the current literal values and the enable's direction are used to determine whether to assert a new value, and which new value to assert. If control is transferred from the left and the variable's value is currently free (i.e., it is in the `init` state and no value is implied), then it asserts the value 1 (to `active 1`) and determines all the transitive implications of that assignment. While implications are still settling (i.e., GChange is asserted) and there is no contradiction, we wait by repeatedly

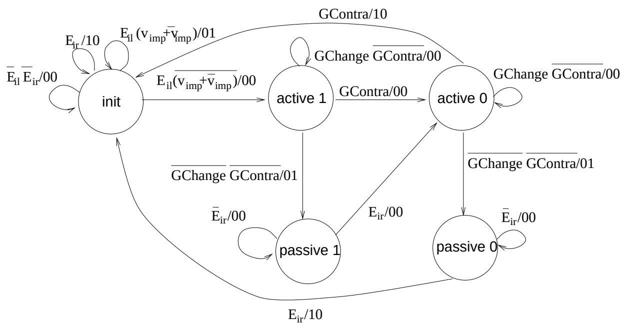
GContra/10
E_{ir}/10
E_{il} $(v_{imp}+\bar{v}_{imp})$/01
$\bar{E}_{il}$ $\bar{E}_{ir}$/00
init
$E_{il}\overline{(v_{imp}+\bar{v}_{imp})}$/00
active 1
GChange $\overline{\text{GContra}}$/00
GContra/00
active 0
GChange $\overline{\text{GContra}}$/00
$\overline{\text{GChange}}$ $\overline{\text{GContra}}$/01
$\overline{\text{GChange}}$ $\overline{\text{GContra}}$/01
$\bar{E}_{ir}$/00
passive 1
E_{ir}/00
passive 0
$\bar{E}_{ir}$/00
E_{ir}/10

Figure 4: State diagram for state machine of one variable.

transitioning to the same state. If contradiction signal is raised, it will try the value 0 instead (`active 0`) and repeat the implication step. If the implication settles without 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 for forward search.

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 value will be maintained by the implication.

Backtracking: If control is transferred to a state machine from the right, then computation is backtracking due to a contradiction, and the goal is to try to find a new assignment that avoids the contradiction. In this scenario, there are three possible responses, again depending on the current local variable values. First, if the variable has been implied by some previous assignment, it is in the `init` state now. It simply passes control to the variable on its left.

Second, if the variable currently has the assigned value 1 (i.e., it is in the `passive 1` state), then it instead assigns the value 0 to it (moving to `active 0`), and determines the implications of that assignment. If no contradiction is found, then the conflict has been cleared, and computation can once again progress by having this state machine transfer control to its right. If, however, a contradiction is again found, then it is reset to `init` and transfers control left.

The third possible scenario is that this variable already has the assigned value 0. This indicates that we have already tried both possible values for this variable; in this case, it 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. On the other hand, if the leftmost state machine attempts to backtrack by further passing control to the left, then this indicates that no solution to the problem exists.

4 Implementation Platform

We use Digital's Pamette board [13] and the IKOS VirtualLogic Emulator [10] to implement our SAT algorithm on configurable hardware.

4.1 Pamette

The Pamette board is a scaled down version of PAM [17]. It has a PCI interface and four Xilinx XC4010E FPGAs available for user-defined functions. It comes with a set of tools for interface and control, and accepts VHDL designs. The tools do not perform automatic partitioning, however, so the design must be partitioned by the user if the problems do not fit on one FPGA chip. Xilinx tools are used to place and route the individual FPGAs. We have successfully mapped small SAT problems to a single FPGA of the Pamette board. However the capacity is too low to implement really interesting problems.

4.2 IKOS Emulator

We use the IKOS VirtualLogic SLI Emulator (Fig. 5) [10] to implement our SAT solver for larger problems where significant speedups are possible. The emulator consists of one system control board and 1 to 6 FPGA array boards. We have used 1 FPGA board for our work. Each FPGA board has an array of 64 Xilinx XC4013E FPGA chips. Time multiplexing using the Virtual Wires technique [3] can overcome the pin limitations of the interconnect between chips. The FPGAs run at an internal clock of 20MHz. Since the interconnect is multiplexed between several signals, it can take multiple cycles to transfer the signal between chips. Thus the emulated clock rate is 20MHz divided by number of internal clock cycles needs for each emulated cycle. The system board connects with a host workstation for downloading configuration and control functions.

Figure 5: Photograph of IKOS Virtualogic SLI logic emulator.

The system accepts structural Verilog as design input. The VirtualLogic compiler performs resynthesis and partitioning. It automatically adds the necessary logic for the interconnect multiplexing. Following place-and-route by Xilinx tools, the generated configuration can be downloaded to the emulator system. Each node in the circuit can be monitored by a logic analyzer when the emulator is running.

5 Experimental Results

5.1 Experimental Setup

To implement the SAT solving algorithm in configurable hardware, we first need to translate from the Boolean formula to a hardware description. The hardware is essentially a large number of state machines, one per variable, implication logic, and the global signals. Naturally, we use a hardware description language (HDL) to represent the circuit for easy portability. Since we are generating the HDL file on the per-SAT-formula basis, we have written a C program to translate from a SAT formula to a VHDL description of the projected hardware to solve the problem. This process takes a few seconds.

The VHDL can be simulated using a software simulator. Since the simulation is very slow, we have only used the VHDL simulator on relatively small problems to verify the correctness of the design. We have written a faster custom simulator, in C, especially for SAT problems. It takes the original SAT CNF formula and runs the same algorithm as the hardware. The program provides the actual solution after the relatively fast simulation and the correct cycle count as if it is running on hardware. The total running time can be obtained by multiplying the cycle count by the clock cycle time determined by the designer (Pamette) or the compiler (IKOS). The VHDL file can be subsequently mapped to a hardware platform. We have mapped them to both Pamette board and IKOS emulator.

We are mainly targeting difficult SAT problems, since these will have very long runtimes (hours or days). As discussed in section 5.3, solvers for such problems normally do not fit on on a single chip. The IKOS logic emulator has much higher logic capacity than Pamette, and its compiler eases the partitioning process. For these reasons, our remaining experimental results will concentrate on the IKOS emulator.

5.2 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 [14]. We use benchmark problems from the DIMACS SAT challenge suite [6]. As discussed earlier, our hardware implementation currently requires that the order in which the search algorithm considers variables be chosen statically prior to the hardware compilation. The heuristic we use here places variables earlier in the

Problem name	Software Runtime	Emulator Clock Rate	Emulator Runtime	Speedup ratio
aim-100-6_0-yes1-1	0.57	1.67 MHz	0.0050	114
aim-200-6_0-yes1-1	128.63	0.95 MHz	1.24	104
dubois20	986.44	2.22 MHz	56.9	17.3
hole10	28859	1.82 MHz	565	51
ii8a2	117470	1.33 MHz	102	1152
ii32c1	0.01	0.77 MHz	0.00036	27.8
jnh1	0.37	0.87 MHz	0.0036	103
par8-1-c	0.02	2.00 MHz	0.00009	222
par16-1-c	202.41	1.00 MHz	1.336	152
pret60_25	695.75	2.50 MHz	14.7	44.9
ssa0432-003	5.2	1.11 MHz	0.19	27.4

Table 1: Configurable SAT solver performance on a collection of DIMACS SAT problems.

search if they have more appearances in the formula.

Table 1 shows the performance comparison and the hardware clock rate on a collection of benchmark problems. Since these problems have been compiled for the IKOS system using a Virtual Wires approach, the emulation clock rates differ for each problem depending on how many signals are multiplexed on the pins in each case. The clock rate tends to be slower for larger, more complex problems. In general, clock rates range from several hundred KHz to a few MHz, much slower than the several hundred MHz of a typical microprocessor. On the other hand, our approach can process up to several hundred clauses in each cycle.

Because of the extensive parallelism in our design, our configurable SAT-solvers are 17X to more than one thousand times faster than a general purpose workstation (e.g. a Sun 5 with 64MB of RAM and a 110MHz processor). This is achieved by exploiting massive, fine-grain parallelism and by calculating an entire level of implications in a single cycle rather than the many instructions required in the software approach.

With faster FPGAs of higher capacity, more I/O and routing resources, and with better placement and routing, it is very reasonable to expect emulation clock rates of 10 to 20MHz on future platforms.

5.3 Hardware Resources

Since we are mapping each input formula to the hardware, the hardware usage depends on the size and complexity of the input formula. Our template-based design for SAT-solving requires no memory, nor any other input data before or during problem solution. Instead, the full problem is embedded into the FPGA array.

Table 2 shows the hardware cost (i.e. total gate equivalents plus flip-flop usage) reported by the IKOS compiler when mapping to the IKOS emulator. The total cost is the summation of the hardware cost of each FPGA chip used to solve the problem. With our current compilation setup, the upper cost limit for each XC4013E chip is 5000. Therefore the upper limit for total hardware cost on a board with 64 chips is 320000. All the problems shown in the table can fit on one board. Among the 240 problems in the DIMACS suite, roughly 170 can currently be mapped to one or two boards. The chips used in our emulator are at the lower end of the current Xilinx product line. Therefore it is reasonable to expect a 10-20X capacity increase in the near future. With this scaling, almost all of the benchmark problems would fit in a system with e.g., less then ten boards.

5.4 Compilation issues

Depending on the problem characteristics, the configurable computing implementation can clearly 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 compilation 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 software runtimes (hours or more).

We use hole10, one of the DIMACS benchmarks, to illustrate the impact of compilation time on performance. (Its compile-time is similar to the other problems we have considered.) GRASP took more than eight hours of CPU time to solve this problem. Our

Problem name	number of variables	number of clauses	Total Cost
aim-100-6_0-yes1-1	100	600	33467
aim-200-6_0-yes1-1	200	1200	100453
dubois20	60	160	10398
hole10	110	561	21872
ii8a2	180	800	37959
ii32c1	225	1280	128064
jnh1	100	850	102663
par8-1-c	64	254	12220
par16-1-c	317	1264	80215
pret60_25	60	160	10673
ssa0432-003	435	1027	103709

Table 2: Hardware cost for a collection of DIMACS SAT problems.

configurable SAT solver completes this problem in 566 seconds. This leads to the 51X speedup ratio shown in Table 1. However, the compilation time should also be included. Using the same Sun workstation, the VirtualWires compilation takes 503.7 seconds and uses ten FPGA chips. The FPGA placement and routing for these 10 chips can be sent to multiple workstations. The place-and-route time is limited by the most difficult chip, which took 2400 seconds in this case. Loading the configuration to hardware take a few seconds. Thus, total solution time including both compilation and running is about 3470 seconds. With this adjustment, the speedup ratio is still a very significant 8.3X.

This experiment shows that SAT problems requiring more than a few hours of runtime with a software approach like GRASP will benefit from the application of configurable hardware, even when compilation time is taken into account. There are more than forty problems taking more than three hours to solve in the suite of 240 problems and there are many real world problems that may need too much computation for conventional approaches.

Current FPGA place-and-route software is more optimized to space utilization, rather than compilation speed. As faster place-and-route software is developed, the range of problems our SAT-solver accelerates will increase.

As an example of faster FPGA compilation on the horizon, consider the work by Gehring and Ludwig [7]. Their FPGA compiler achieves 10 to 100X speedup compared to Xilinx tools. In their case, the speed gain is mainly achieved by preserving the circuit's hierarchical information. Placement with arrays is simplified and routing for identical elements can be duplicated. Our SAT implementation also has these properties, so it will be an excellent candidate for speedup with these faster place-and-route tools. If the FPGA compilation time can be reduced from roughly one hour to a few minutes, there will be many problems, both SAT and otherwise, that can benefit from the input-specific solution using configurable hardware. We are also looking at rapid placement and routing algorithms that can trade spatial efficiency for compilation speed. Basically, as technology advances, more routing resources can be allocated on the FPGA. This mitigate the need for placement optimization. For our array styled structures, it is much more efficient to perform high level placement. The placement and routing of a single cell can be executed separately only once for the whole class of problems. With these approaches combined, we believe the compilation time can be dramatically reduced.

Overall, we envision a system where easy SAT problems are still solved in software. Very difficult problems, ones that often timeout today, can invoke the hardware compiler and configure an FPGA board to assist in solving them.

6 Related Work

At this time, we know of two interesting proposals for solving SAT using reconfigurable hardware [16, 1]. Both these proposals are recent and have overlapped with our work. To the best of our knowledge, only our work has implemented relatively large, difficult problems on hardware.

Suyama *et al.* [16] have proposed their own SAT algorithm, instead of the Davis-Putnam approach. Their algorithm is characterized by the fact that at any point, a full 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. In addition, their implementation requires a "max" calculator and a complex rule checker making it very hardware-resource-intensive. In comparison, the advantage of our approach is that it is an implementation of the Davis-Putnam algorithm resulting in many fewer states visited. We believe on the same hardware, our implementation will be much faster.

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 [8] algorithm in reconfigurable hardware. PODEM is used to solve test generation problems. It is targeted on multilevel circuits and it is not efficient on generic SAT problems, such as those in the DIMACS benchmark suite. Davis-Putnam's efficient data structures capture relationships between internal variables in a circuit; this reduces the state space visited and the run time significantly over PODEM [11, 4, 14]. Furthermore, they used centralized control for the input assignment. There is complex flow control in this part and it may need to manipulate up to several hundred signals. In contrast, our approach uses distributed control and only three global signals.

J. Babb *et al* reported solving graph problems using similar input-specific approach [2], which they called "dynamic computation structures". From an input graph, a circuit description is generated in Verilog. It is then compiled to Virtual Wires emulator systems (an earlier version of the IKOS emulator). The speedup against software is 10 to 400 times without compilation overhead. However, since these graph problems are relatively easy and short-running, the long compilation time makes the hardware acceleration useless. Contrary to their situation, there are many difficult SAT problems that make the compilation time of our implementations often acceptable.

7 Discussion

Based on our experience with this case study of a SAT solver, we have some observations regarding complex problem solving on configurable hardware and the application of input-specific approach. In contrast to traditional applications (e.g. signal processing) on configurable hardware, our design is not built around one or several deep pipelined data paths. Instead, it has more control logic and global connections (despite our attempts to minimize them). The systolic array style hardware such as SPLASH [9] does not suit the SAT application well, because of limited interconnect between chips and the specialized topology of the array. A more flexible hardware system is desired. That is the reason behind our choice of a logic emulator as our implementation platform. The IKOS compiler also eases partitioning problems that would otherwise complicate our automatic design generator.

However, further improvements on the configurable SAT-solver, and other problems like it, will place further requirements on the configurable hardware platform used. For example, GRASP [14] has achieve dramatic improvements compared to the basic algorithm, by using a diagnostic engine to find the variable assignments that are responsible for conflicts. With this information, it applies non-chronological backtracking which can prune more of the search space than basic backtracking. It can also dynamically add clauses to prune the search space further. We have also implemented a preliminary variant of nonchronological backtracking [18] and are looking at further improvements. In order to make complicated diagnoses in support of non-chronological backtracking, the host computer may need to have quite low-latency communication with the configurable hardware. In an even more aggressive approach, we may also choose to modify the configurable hardware on-the-fly to add new clauses or new implications. This requires the techniques of dynamic (during the application) hardware reconfiguration. We believe that such future work will provide more insight on applications and techniques for configurable computing.

8 Conclusions

Overall, the contributions of this paper are twofold. First, we provide a system design for formula-specific Boolean satisfiability solutions based on a configurable hardware implementation. We have a fully-automatic procedure to generate the hardware configuration from the original SAT formula. The problems have been successfully mapped to Digital's Pamette and IKOS emulator. Moreover, our hardware performance results demonstrate that the configurable computing approach has the potential for dramatic improvements over general-purpose software-based techniques.

The second contribution of this paper is much broader. We present our SAT-solver as a case study of a complicated configurable hardware application. 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 relatively easy and fast to compile configurations for them. Experiments with current compiler and hardware platforms show that difficult problems can benefit from the application of configurable hardware, even including the compilation time. With improvements in FPGA compilation and other software techniques, the input-specific approach will have even broader appeal on many difficult problems; as such, it represents a new class of applications benefiting from configurable hardware.

9 Acknowledgements

The authors thank Joshua Toub for his help mapping this SAT algorithm to the Pamette board. This work was supported in part by DARPA grant DABT63-97-1-0001, by an NSF Career Award (Martonosi) and by a research donation from NEC.

References

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

[2] J. Babb, M. Frank, and A. Agarwal. Solving Graph Problems with Dynamic Computation Structures. In *SPIE '96: High-Speed Computing, Digital Signal Processing, and Filtering Using Reconfigurable Logic*, pages 225–236, 1996.

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

[4] 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.

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

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

[7] S. W. Gehring and S. H.-M. Ludwig. Fast Integrated Tools for Circuit Design with FPGAs. To be published.

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

[9] M. Gokhale, W. Holmes, A. Kopser, et al. Building and Using a Highly Parallel Programmable Logic Array. *IEEE Computer*, 24(1):81–89, Jan. 1991.

[10] IKOS Systems. Virtual Logic SLI Emulator. http://www.ikos.com.

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

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

[13] M. Shand. PCI Pamette V1. http://www.research.digital.com/SRC/pamette.

[14] 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.

[15] 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.

[16] 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.

[17] J. E. Vuillemin, P. Bertin, D. Roncin, et al. Programmable Active Memories: Reconfigurable Systems Come of Age. *IEEE Trans. on VLSI Systems*, 1996.

[18] 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*, June 1998.