Using Configurable Computing to Accelerate Boolean Satisfiability

Peixin Zhong, Margaret Martonosi, Pranav Ashar and Sharad Malik

Abstract— **The issues of software compute time and complexity are very important in current CAD tools. As FPGA speeds and densities increase, the opportunity for effective hardware accelerators built from FPGA technology has opened up. This paper describes and evaluates a formula-specific method for implementing Boolean satisfiability solver circuits in configurable hardware. That is, using a template generator, we create circuits specific to the problem instance to be solved. This approach yields impressive runtime speedups of up to several hundred times compared to the software approaches. The high performance comes from realizing fine-grained parallelism inherent in the clause evaluation and implication and from direct mapping of Boolean relations into logic gates. Our implementation uses a commercially-available hardware system for proof of concept. This system yields more than 100 times run-time speedup on many problems, even though the clock rate of the hardware is 100 times slower than that of the workstation running the software solver. While the time to compile the solver circuit to configurable hardware can be quite long on current platforms (20-40 minutes per chip), this paper discusses new approaches to overcome this compilation overhead. More broadly, we view this work as a case study in the burgeoning domain of high performance configurable computing. Our approach realizes large amount of fine-grained parallelism, and has broad applications in the VLSI CAD area.**

Keywords— **Boolean satisfiability, FPGA, parallel computing, hardware acceleration, VLSI CAD.**

I. Introduction

As the complexity of designing electronic systems increases, the effectiveness and efficiency of Computer Aided Design (CAD) tools become very important. Techniques that accelerate core CAD algorithms can bring about important changes in product design times.

This paper introduces an idea for accelerating automatic test pattern generation (ATPG), logic synthesis and verification by accelerating the Boolean satisfiability (SAT) problem at their core. In particular, we describe a configurable computing approach for SAT-solving that can offer substantial speedups over traditional software approaches. Configurable hardware techniques provide a new way of realizing large amount of fine-grained parallelism. Since Boolean satisfiability algorithms are logical-operation-intensive, the hardware approach garners good performance by mapping portions of the SAT expressions directly to logic gates, and by harnessing large amounts of parallelism in the evaluation of the logic.

While hardware accelerators for CAD are not new, using field-programmable gate arrays (FPGAs) provides a more cost-effective mechanism for implementing them by avoiding the high cost of building ASICs. We refer to this type of computation as configurable computing because the hardware is configured according to the application. A system based on configurable hardware can be used to accelerate different applications since the hardware can be tailored into an unlimited number of different configurations. Moreover, it even allows one to accelerate computations on an "input-specific" basis. That is, starting from a general solution template design, we specialize the accelerator design to the input data for a particular problem instance being solved. This allows us to reap potentially larger benefits from the hardware mapping than if it were a more general design.

We have implemented our SAT solver on an FPGA-based logic emulator and achieved significant runtime speedups. Our project has led to a number of observations regarding FPGA-based accelerators for compute intensive problems. Most importantly, while most configurable computing applications currently concentrate on data processing and use systolic designs (for example, see [3], [10]), our design explores the potential for more complex control structures and for speedups on non-systolic implementation. Our implementation demonstrates configurable computing's potential for impact on a much broader set of applications than may initially have been considered.

On the other hand, the SAT solver circuit we propose is generated according to each problem instance and its mapping time is also a part of problem solving time. Since current FPGA compilation tools are optimized for high density mapping, this compilation time can be quite long. This compilation overhead limits the overall performance of our SAT solver. We propose approaches to tailor the hardware compilation tool to greatly reduce the compilation time by utilizing the hierarchical structure and repetition of modules. This also serves as an alert to the CAD community to encourage development of faster mapping tools. With this overhead reduced, the instance-specific acceleration approach can be used for a much wider range of problems.

The remainder of this paper is structured as follows. Section II provides an overview of configurable computing and its promise for accelerating CAD applications. Section III then describes the SAT problem, and Sections IV presents our SAT-solver implementation in configurable hardware, giving performance results compared to software SAT solvers. In Section V, we discuss compilation time issues inherent in our approach and methods to reduce such overhead. Sections VI discusses related work. Finally, Sec-

P. Zhong, M. Martonosi and S. Malik are with Department of Electrical Engineering, Princeton University, Princeton, NJ 08544. P. Ashar is with NEC CCRL, Princeton, NJ 08540.

This work was supported in part by DARPA grant DABT63-97-1-0001, by a grant from the NSF Experimental Systems Program and by a research gift from NEC.

tion VII gives conclusions.

II. The Promise of Configurable Computing

Electronic design automation has long faced a challenge of meeting the computational demands of solutions to a wide range of problems. Typically this challenge has been dealt with using efficient software algorithm implementations running on new generations of faster microprocessors. General-purpose computing resources are not, however, sufficient in all cases. This motivates the development of special-purpose hardware accelerators geared towards solving individual applications.

Traditionally, logic simulation has been the primary target of special-purpose accelerators. Here the logical behavior of the circuit is simulated for a large set of test vectors to assure the correct behavior of the circuit. The growth in VLSI and the advent of field programmable gate arrays (FPGAs) have made it possible to use the basic, input-specific hardware idea for logic emulation. Several commercial emulator systems based on FPGAs are available, such as the Quickturn System Realizer [20], [15] and the IKOS VirtuaLogic Emulator [11]. These systems use FPGAs as the basic hardware and the targeted circuit is compiled to the FPGAs. Running the circuits on FPGAs provides much higher simulation speed than software approaches using general-purpose computers.

Given the success of logic emulation, it is natural to raise the question: can we use similar technology to accelerate other CAD problems? Since SRAM-based FPGAs can be programmed an unlimited number of times, a carefully designed system can be used to solve many different problems.

We identify the following features of configurable hardware as important characteristics for high performance computing:

- Faster function execution: Since the hardware is generated according to the application, the hardware functionality can be tailored accordingly. Instead of using a set of general-purpose instructions, specialized functional units can perform operations in one cycle where a conventional CPU may take many cycles.
- Fewer external memory accesses: Since the instructions are translated into circuit configurations, there is no need for a sustained instruction stream. Furthermore, the data flow can be customized according to the application, and intermediate results can be directly passed to the next stage. Many load/store operations are saved and the memory accesses can be localized, reduced, or even eliminated in some cases.
- More parallel processing elements: Since only necessary functions are mapped to the hardware, more functional units can be allocated on a single chip. For example, if the application only involves very short integers, there is no need to build floating-point units or 64-bit ALUs into the circuit. In this way, FPGA chip may contain more useful processing elements than a conventional processor.
- Lower communication overhead: In a conventional multiprocessor computer, the communication between different processors is often expensive. In configurable hardware, the communication channel is customized and sometimes direct wires can be used for fast communication. This makes the configurable hardware more suitable for exploiting parallelism on a fine-grained level.

Since an FPGA-based system typically has a much slower clock rate than a modern microprocessor, it is especially important to aggressively exploit parallelism to achieve higher performance. Although most of the previous research on configurable computing concentrates on data-intensive applications such as digital signal and image processing, we feel the characteristics of FPGAs are also suitable for implementing control intensive algorithms as found in many CAD problems. FPGAs are by design well suited for implementing random logic and finite state machines (FSM), which do not make good use of the long data words provided by modern processors. FPGAs are suitable for bit-level operations and provide more efficient hardware utilization. For these reasons, we believe configurable computing may find many suitable applications in the CAD domain.

Our focus on the Boolean satisfiability problem (SAT) is a step towards exploring the potential of configurable computing in CAD algorithms. Our choice of SAT as a case study here was guided by two main factors:

- SAT is a basic NP-complete problem, and it serves as the core problem for many CAD applications. A complete algorithm may have very long solution time even when the input is relatively small. Gaining experience on SAT will also help us accelerate other similarly compute-intensive problems.
- The algorithm involves many logic evaluation operations. These operations can be easily mapped to configurable hardware. We feel there is much potential for performance improvement by parallel logic evaluations.

Thus we implemented a backtrack search algorithm of SAT. Since many CAD problems are solved by search algorithms, experience on this problem may help us apply similar techniques to a broader class of applications.

III. 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 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, the partial assignment $v_i = 0, v_k = 0$ implies $v_j = 0$. A variable that is neither assigned nor implied is considered *free*. If all the literals in a clause are evaluated to 0, the Boolean formula cannot be satisfied. This is called a conflict or contradiction; it means the current partial assignment can not be a part of any valid solution.

Most current SAT solvers are based on the Davis-Putnam algorithm [6] illustrated in Figure 1. 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 no free variables are available and no contradictions have been encountered (implying that all the clauses have been satisfied and a solution has been found), or when all possible assignments have been exhausted. The algorithm is complete in that it will find a solution if it exists.

```
Initialize, all variables assigned 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 ()
```

Fig. 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 invalid 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 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 SAT solver is designed to take advantage of this parallelism. Section IV details our mapping of the Davis-Putnam algorithm onto reconfigurable hardware in a formula-specific manner.

Finally, we note that recent software implementations of the Davis-Putnam algorithm have enhanced it in various ways while maintaining the same basic flow [5], [12], [16], [17]. In this paper we concentrate on the basic aspects of hardware acceleration. Our hardware design for accelerating more sophisticated SAT algorithms is reported elsewhere [22].

IV. Formula-Specific Mapping of SAT Solver

A. Design Overview

In this paper, we present an "formula-specific" SAT solver. That is, the solver circuit is generated according to the Boolean formula to be solved. This design realizes large amount of parallelism by creating a set of logic evaluation circuits specific to the formula being solved.

From analyzing software SAT solvers, we identify implication and conflict checking as the most compute-intensive portions; the hardware should try to accelerate these areas. These operations involve checking each clause of the SAT formula to determine new implications and identifying unsatisfied clauses at each step during the search process. All the clauses can be evaluated in parallel because there is no direct dependency among the clauses. In conventional computers, communication overhead makes such fine-grained parallelism difficult to utilize. It is also too expensive to allocate one CPU for each clause. However such parallel clause evaluation can be implemented in hardware using a small number of logic gates. We translate all the clauses into an implication circuit in configurable hardware. This is essentially a large combinational circuit that takes the current assignment of the variables, computes the implied values, and checks conflicts.

In addition to the implication circuit, a control unit is used to manage the backtrack search. Its main function is controlling the branch and backtrack depending on the output of the implication circuit. We implement such control functions on the configurable hardware so there is no communication between the host CPU and the configurable hardware during problem solving time. The control unit consists of a set of finite state machines connected as a linear array. Each state machine controls the value of one variable. This regular, distributed control makes it easier to partition large problems across multiple FPGAs. The details of the circuit are described in the following sections.

B. Hardware Organization

B.1 The Implication Circuit

The implication circuit can be automatically created based on the Boolean formula being solved. For a SAT clause with n literals, if n-1 literals evaluate to 0, the remaining literal must be 1 to satisfy the clause. Such implications are easy to map to logic gates. For a clause $(v_i + v_j + v_k)$, the implication will be $\bar{v}_i\bar{v}_j \rightarrow v_k = 1$, $\bar{v}_j\bar{v}_k \rightarrow v_i = 1$ and $\bar{v}_i\bar{v}_k \rightarrow v_j = 1$.

Fig. 4. State Diagram for Backtracking Machine.

state machine for each variable. This distributed topology keeps global signals to a minimum and reduces hardware costs.

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 two-bit encoding of values of the positive and negative literals of the 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.

At any instant, only one state machine is in control. This corresponds to the variable we branch on in the search process. Once that state machine has finished processing, it asserts E_{or} to transfer control to the state machine on the right (if branching forward in the computation) or it asserts E_{ol} to pass control to the left (if backtracking).

If the control is passed beyond the right most variable, all the variables are assigned and there is no conflict. A solution is found. If the control is passed beyond the left most variable, there is no solution to this problem.

As discussed earlier, our hardware implementation currently requires variables be ordered statically prior to the hardware implementation. The heuristic used is to place earlier the variables with more appearances in the formula. The value of such variables will be determined early in order to generate more implications or satisfy more clauses as soon as possible.

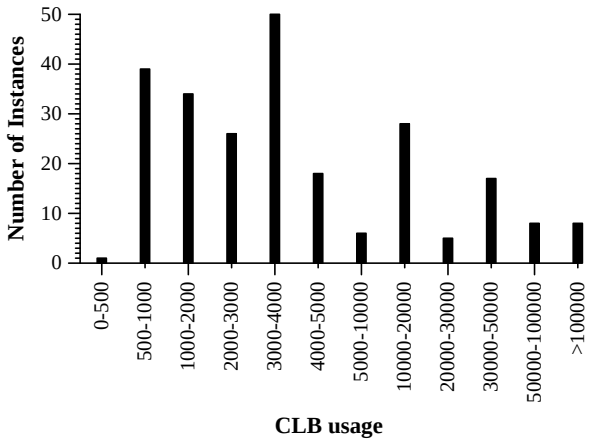

Fig. 5. A histogram of CLB usage for the DIMACS SAT benchmarks.

C. Hardware Usage

The hardware usage for our approach is a function of the size and complexity of the formula being solved. The full problem is embedded into the circuit instantiated on FPGAs.

A typical FPGA, such as Xilinx XC4000 series [21], contains an array of configurable logic blocks (CLBs). A CLB is a basic unit to be configured as logic gates. It contains two registers, two four-input lookup tables and one three-input lookup table. A big FPGA, such as XC40250XV, contains 8464 CLBs. As illustrated in Figure 5, the SAT solver's CLB requirement varies depending on the formula to be solved. This figure shows a histogram of the frequency that different quantities of CLBs are needed, for all the problems in the DIMACS SAT benchmarks [7], for our design. The count includes the logic gates but does not consider usage for routing. Across the DIMACS benchmarks, the median CLB requirement is 3655. Overall, more than half of these benchmark problems will fit within one large FPGA chip. More than 90% will fit on an array of 16 FPGAs.

D. Implementation Framework

D.1 Hardware

We use the IKOS VirtualLogic SLI Emulator [11] to implement our SAT solver. This is a commercial system that provides a generic circuit implementation platform on FPGAs. The emulator consists of one system control board and 1 to 6 FPGA array boards. Each FPGA board has 64 Xilinx XC4013E FPGA chips in an 8 by 8 mesh connected to four near neighbors. The SAT-solving circuit is mapped to these FPGA boards. The system board has an interface to the host computer and logic analyzer. It is connected to a host computer via a SCSI interface. It also contains several FPGAs for interfacing with the logic analyzer. A time-multiplexing technique called VirtualWires is used to overcome pin limitations on the interconnect between chips [4]. Thus, the number of logical wires between chips can exceed the available physical I/O pins. The FPGAs run at an internal clock of up to 26.7 MHz. Since the interconnect is multiplexed, it takes several internal cycles to send the signals to other chips. Thus the emulated clock rate is the internal clock rate divided by a multiplexing factor, which depends on the circuit implemented. Usually the emulated clock rate seen by the user's circuit is in the range of 0.7-2 MHz.

This system is used for proof of concept. Its mesh topology and relatively high capacity gives much flexibility in the design. However the clock rate is quite low. A more specialized hardware platform would provide higher speed but would have required more effort to design and implement [22].

D.2 Software

For a Boolean formula, the following tool flow translates it into the hardware configuration for a SAT-solver circuit:

- SAT-Solver Template Compiler: This C program transforms the Boolean formula into the VHDL hardware description of the solver circuit. Since the state machine for each variable is the same for every problem, it is designed as a component and is used in every circuit. The implication circuit is formula-specific and is generated for every problem instance. This is a fast process that takes only a few seconds.
- Synopsys Design Complier: The IKOS emulator only accepts structural Verilog mapped to its own gate libraries. We use the Synopsys Design Compiler to translate from VHDL to structural Verilog. This time could be avoided by directly generating the necessary Verilog file.
- IKOS compiler: The compiler performs resynthesis and partitioning from the Verilog to a multi-chip design for the emulator. The output is the netlist for the circuit. This process normally takes 10-20 minutes.
- Xilinx PPR Tools: The description of the FPGA is then compiled into the FPGA configuration bitstream. This is performed by the Xilinx partition, placement and routing tools (PPR). This is the most time consuming of the steps; each FPGA takes about 20-40 minutes to compile. This work can be farmed out to multiple computers.
- Emulator Runtime Tool: This tool controls the download and running of the emulator.

E. Performance Results

In this section we compare the run-time performance of our hardware implementation of the SAT algorithm to its implementation in GRASP [16]. The time to map the problem to the hardware is not included in this performance comparison, but will be discussed in next section.

E.1 Experiment Setup

In order to estimate the hardware performance, we have used simulation to count the exact number of hardware clock cycles needed to solve each problem. Our experience with general VHDL simulation has been that it is very slow. Although we have used it to verify design correctness on moderate examples, it is not practical to simulate

the large, difficult problems where our approach's performance advantages are most compelling. For this reason, we have generated a C-language model of the hardware that is significantly faster then general VHDL simulation. Like a VHDL simulation, the C-language simulation is also capable of counting the number of hardware clock cycles exactly. It also lets us profile performance in ways that are difficult in a direct hardware implementation. We have compiled a subset of the problems to the hardware to evaluate the correctness of the design. However, we have not compiled all 240 of the benchmark problems to hardware. Instead, the simulation results are used for performance comparison. The number of hardware clock cycles is translated into runtime based on a clock rate of 1.33 MHz. We use this single clock rate to simplify the data presentation; it is the median value of user clock rates for the problems we have compiled. The actual clock rates amongst this subset range from 0.7 MHz to 2 MHz, depending on the problem.

The GRASP software runs on a Sun 5 workstation with 64 MB of RAM and a 110 MHz processor.[1] We have set the options so GRASP performs the algorithm comparable to our hardware implementation with the same variable ordering.

E.2 Runtime Speedup

Figure 6 shows a histogram of the run time performance. Each bar corresponds to a speedup range where speedup is the ratio of the GRASP run time to the hardware run time. That is, a 10X speedup means our approach is ten times faster than GRASP software. The height of the bar corresponds to the number of examples for which that corresponding speedup is obtained.

The problem instances come from the DIMACS SAT benchmark suite [7]. There are 240 problems in the DIMACS suite. Some of them take very long to solve and neither GRASP nor our C-model simulation gets a result in reasonable amount of time. There are 134 instances that at least one program finished.

In all examples that complete in GRASP, our hardware implementation also completes, in usually much better time. 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*. It shows the custom solver circuit achieves significant runtime speedup even though the hardware clock is much slower than that of the general-purpose computer. This clearly shows the performance potential of such approach. Section V will discuss how the compilation time issues impact the speedups.

E.3 Discussion of Run-time Performance and Parallelism

The speedup mainly comes from two factors: (1) evaluating implications in parallel, and (2) squeezing multiple operations into one cycle. In order to better understand the reasons for such speedups, it is necessary to examine how many operations are actually taken in parallel. In hardware, since the clauses are mapped into the implication circuit, all the clauses can be evaluated in each clock cycle. However, in software, there is no need to evaluate all clauses whenever a new value is assigned to a variable. Only the clauses containing the newly modified variable should be evaluated because only these clauses may be affected by the new assignment. The effective number of clause evaluations in one cycle is the number of clauses that contain the variables updated in the previous cycle. This can serve as a rough measurement of realized parallelism.

[1] Our department has faster SGI hosts, but this was the fastest SUN host that could run the SPARC executable that we were able to obtain. Scaling to current processors would lead to about 6 times performance improvement for GRASP, but the runtime speedups shown in Figure 6 would still be fairly significant. This is also the system we used to compile our hardware configurations.

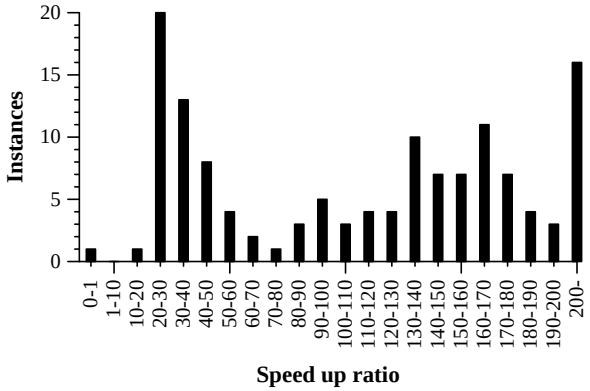

Fig. 6. Runtime speedup ratio histogram comparing our basic algorithm on FPGAs to a GRASP run with basic backtracking.

TABLE I

Comparison of acceleration in cycle counts. The numbers of clauses and average numbers of effective parallel clause evaluations are also included to show the correlations.

Problem name	Number of clauses	Parallel evals	SW/HW cycle count ratio
a50-2_0-y1-2	100	7.1	2750
a100-2_0-y1-4	200	8.4	2184
a200-6_0-y1-1	1200	62.3	12076
dubois20	160	8.0	859
hole7	204	18.3	2750
hole8	297	21.9	2818
hole9	415	25.9	2817
hole10	561	30.1	3077
ii8a2	800	15.8	94888
par8-1-c	254	29.4	12191
par-16-1-c	1264	60.4	17000
pret60_40	160	8.5	2093
ssa0432-003	1027	11.0	2860

Table I shows statistics for average number of actual clause evaluations per cycle for a range of SAT problems. The table also shows the total number of clauses, and cycle count ratio of the software and reconfigurable hardware. The total number of clauses ranges from 100 to 1264. The average number of clause evaluations ranges from 7 to 62. Larger problems tend to have more parallelism in the clause evaluation. For example the a200-6_0-y1-1 has higher speedup than a100-2_0-y1-4 because more clauses are evaluated in each cycle. The par8-1-c and par16-1-c problems show similar characteristic. This is good because

TABLE II

Comparison of run time and compilation time: software run time, hardware run time, IKOS compilation time, Xilinx FPGA placement and routing time and the total hardware solution time are show in the table. The unit for time is second.

Problem name	SW run	HW run	IKOS compile	ppr	HW of Total
a50-2_0-y1-2	0.05	0.0011	333	10440	10783
a100-2_0-y1-4	894	42	5460	84060	89572
a200-6_0-y1-1	128	1.35	2426	>100K	>100K
dubois20	986	70.8	996	10380	11447
hole7	10	0.22	343	8160	8503
hole8	128	2.8	518	11640	12170
hole9	1818	45.2	554	13740	14349
hole10	28859	695	721	18960	20386
ii8a2	117470	127	1475	35100	36712
par8-1-c	0.02	0.00011	464	12360	12834
par-16-1-c	202	1.3	3501	79680	83192
pret60_40	705	18	386	12000	12414
ssa0432-003	5.2	0.21	5280	83880	89160

our hardware approach targets larger and more difficult problems and these problems may have a higher speedup ratio.

In addition to the apparent parallel evaluation, another important factor is merging multiple instructions into one cycle. In a software version, evaluating a clause requires accessing memories for relevant pointers and variables. Then implications and conflicts can be evaluated accordingly. This is a process that requires many instructions and many memory accesses. In hardware, they are all performed in one cycle without accessing external memory. This is also a significant component for the acceleration.

With these two factors combined, the hardware is capable of performing the computation task worth of thousands of general-purpose computer cycles in one hardware cycle in the reconfigurable hardware.

V. Hardware Compilation Time Issues

A. Compilation Time of Current Platform

Since the circuit is generated according to the formula to be solved, the compilation time is part of the problem solving time and should also be considered. In this section, we describe the actual compilation time for the IKOS platform.

As previously mentioned, the following steps contribute to the total compilation time. (The Synopsys compilation is excluded because it can be skipped with direct Verilog generation.):

- Template Compilation: It only takes a few seconds to execute.
- IKOS compilation: It takes 10 to 20 minutes to compile most problems into xff files.
- Xilinx PPR: These xff files are subsequently compiled using the Xilinx partition, placement and routing (PPR) tools. Normally it takes 20 to 40 minutes to generate the configuration for one FPGA chip. For very lightly used chips the time is significantly shorter.
- Download configuration: It takes about 10 seconds to download the bitstream to the hardware.

Table II shows compile-time statistics for a subset of the SAT suite. We show the software and hardware runtime, the IKOS compilation time, Xilinx PPR time and the total time to generate a hardware solution for the problem. We assume no parallelism in PPR. The computer for the hardware compilation is the same as the one used to run software solver for comparison purpose. Since the IKOS software provides the mechanism to allocate the PPR tasks to a farm of workstations, however, the total compilation time could be about 1 hour for all problems if a large number of computers were used. From the table, we can see the hardware compilation time is often more than the time used to solve the problem in software, except for the very difficult problems.

B. Compilation Time vs Problem Difficulty

Since the compilation time is on the order of hours, our approach will not provide useful speedups for problems that can be solved in minutes or less by software approach. On the other hand, very difficult problems can still benefit from the hardware approach because the compilation time can be much shorter that software solver. This trend is clearly shown in the holeN series of problems from the DIMACS suite.

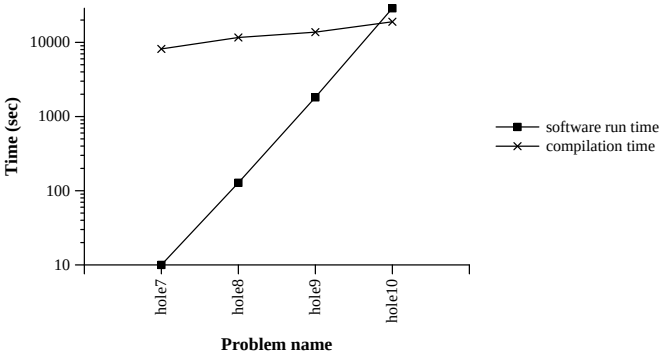

Fig. 7. Software run time and hardware compilation time versus the size of holeN problems.

In Table II, we include the data for problems hole7 to hole10. These are a group of problems of similar characteristics with different size and difficulty. Fig. 7 plots on a semilog scale of GRASP software run times and hardware compilation times for hole problems. Clearly, the software run time scales up much more rapidly than the compilation time.

The solution time is in the worst case exponential to the size of the problem. Although the hardware compilation problem also contains multiple subproblems that are NP hard, there is no need to obtain an optimum solution. Getting an acceptable placement and routing is much easier. From our experience, the time to place and route one FPGA chip takes 20-40 minutes. It does not vary much because the number of gates on each chip is always below an upper limit. Larger problem maps to more FPGA chips. This is more close to a linear correlation. With larger, more difficult problems, the hardware approach shows more promise, because the compilation time becomes

much less prominent. This demonstrates the potential for useful hardware acceleration on large, difficult problems.

C. Discussion on Reducing Compilation Time

As we have discussed before, the compilation overhead currently limits the usage of our approach. The current FPGA tools are designed to maximize hardware utilization and tend to use a lot of computation time to achieve this goal. The Xilinx tools, for example, use stochastic, iterative algorithms for placement. It spends large amount of time to iteratively search for a better placement. A new tool optimized for fast compilation will be more desirable for our application.

As an example of faster FPGA compilation on the horizon, consider the work by Gehring and Ludwig [8]. Their FPGA compiler achieves 10 to 100X speedup compared to Xilinx tools. In their case, the speed gain is mainly achieved by preserving the hierarchical information of the circuit. Placement with arrays is simplified and routing for identical elements can be duplicated. The Teramac project also uses hierarchical placement techniques for fast hardware compilation [2]. They have achieved 1 Million gate compilation in 2 hours. This is on a routing-rich FPGA structure. This shows tailoring the architecture of FPGAs will also make it more suitable for fast placement and routing.

Another investigation by Swartz *et al* [19] concentrates on a fast FPGA router. It can route 20,000 LUT/FF pairs in 23 seconds if routing resources are not stressed. Such approaches are especially useful in configurable computing because hardware utilization is often less important than rapid implementation.

Based on our observations regarding compile-time, we propose a framework to further improve the hardware compilation speed. The typical hardware compilation procedure includes synthesis, mapping, placement and routing. Of these, placement and routing are most time consuming.

One of the problems with the current tools is that the hierarchy information is not well preserved. In fact, the placement and mapping tools normally work on a fully flattened netlist. Thus, they cannot exploit the regularity in the design, such as our linear array of state machines.

In our proposed approach, the modules are mapped, placed and routed earlier and are build into a module library for later use. These modules can be highly optimized because they are compiled once and reused for many times because placing the module at different locations does not affect the internal structure of the module.

When a full circuit description is generated, most of the circuits are mapped into the pre-compiled modules. Gate level mapping is only activated for the inter-module connections and the instance-specific parts. Similarly, the placement is mostly based on the larger modules instead of gates. During the routing process, only inter-module signals should be routed. This approach may lead to great compilation time savings for a design that is mostly composed of highly repetitive structures.

We have also worked on a more regular hardware design for easier mapping [22]. In our newest design, a pipelined bus replaces the random direct connections. The clauses are parametrized modules connected to the bus. This design is amenable to direct circuit generation.

Fully elaboration of these approaches is beyond the scope of this paper. Preliminary work shows the compilation overhead of each problem instance can at least be reduced to the order of a few minutes, which will render the reconfigurable approach attractive to broader range of problems.

VI. Related Work

A. Hardware SAT Accelerators

There have been other recent studies of SAT on configurable hardware. For example, Suyama *et al* [18] have proposed their own SAT algorithm. Their algorithm is characterized by using full assignment of all the variables in the search. This approach is much less efficient that Davis-Putnam algorithm as stated in their paper.

Another proposal by Abramovici and Saab [1] is basically an implementation of a PODEM-based [9] algorithm. PODEM is developed for the ATPG problems. If extended to general SAT problems, it is normally less efficient than the Davis-Putnam algorithm [12], [5], [16]. This paper is mostly based on simulation and has not shown any detail of the implementation.

To our knowledge, neither of these groups has reported implementing problems of realistic sizes on actual systems. On the other hand, we are presenting our results based on experiments with commercial logic emulators.

The work by Platzner and De Micheli is more recent [13]. It has compared several algorithm options. Simulation results are given on an algorithm similar to Davis-Putnam algorithm. However, the experimental implementation is based on a simpler backtrack algorithm. In this algorithm, backtrack occurs when a clause evaluates to false. Implications are not deduced. This algorithm needs more clock cycles to solve the same problem due to this less efficient search method. On the other hand, the hardware is simpler and uses fewer gates. Newer Xilinx tools are used and the hardware compilation time ranges from 2 minutes to 10 minutes. However, it is not shown how the more efficient algorithm is implemented. It is also unknown how the system performance will be affected if the design were spread across several FPGAs.

B. Software SAT Algorithms

SAT algorithms fall into two categories: complete or incomplete. Complete algorithms will exhaust the possible search space and will definitely find a solution if one exists. That is, they will exhaust the possible search space. In this paper, we have only considered complete algorithms.

The Davis-Putnam algorithm is a basis for many modern complete SAT solvers [6]. Newer software algorithms try to improve the efficiency by employing different techniques, including (1) preprocessing of the formula, (2) look-ahead techniques, such as heuristics in selecting the next branching variable, and (3) look-back techniques. Most

notably, two separate recent efforts, GRASP [16] and Relsat [14] have used look-back techniques to achieve significant speedups compared to the basic algorithm. They are based on conflict analysis that leads to run-time learning and thus to more effective backtrack.

VII. 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 modest. Moreover, our hardware performance results indicate that the configurable hardware approach has the potential for dramatic improvements over conventional computers. Experiments with an emulator system show up to several hundred times run-time speedup and there is much potential for further improvement as the capacity and performance of FPGAs improve.

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 data parallelism by performing template-driven hardware generation for each problem/data set being solved. Although current commercial software tools may take relatively long time to create the actual configuration, the parameterized modular design methodology creates well-structured circuits. Exploiting this structural regularity in hardware compilation can significantly reduce this overhead.

We believe our approach can also be extended to many other CAD problems. These problems often have significant fine-grained parallelism with frequent logic evaluations and simple arithmetic computations. These characteristics make them amenable to FPGA based accelerators. With the advent of million-gate FPGAs and much improved performance, FPGA based accelerator will provide an attractive solution to the increasing need of computation power for CAD algorithms.

References

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

[2] R. Amerson, R. J. Carter, W. B. Culbertson, P. Kuekes, and G. Snider. Teramac-configurable custom computing. In *Proceedings IEEE Symposium on FPGAs for Custom Computing Machines*, pages 32–38, Apr. 1995.

[3] P. Athanas and L. Abbott. Real-Time Image Processing on a Custom Computing Platform. *IEEE Computer*, 28(2):16–24, Feb. 1995.

[4] J. Babb, R. Tessier, M. Dahl, S. Z. Hanono, D. M. Hoki, and A. Agarwal. Logic Emulation with Virtual Wires. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 16:609–626, 1997.

[5] 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.

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

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

[8] S. W. Gehring and S. H.-M. Ludwig. Fast Integrated Tools for Circuit Design with FPGAs. In *FPGA'98 1998 ACM/SIGDA Sixth International Symposium on Field Programmable Gate Arrays*, pages 133–139, Feb. 1998.

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

[10] 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.

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

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

[13] M. Platzner and G. De Micheli. Acceleration of Satisfiability Algorithms by Reconfigurable Hardware. In *Field Programmable Logic and Applications, FPL '98*, Aug. 1998.

[14] R. J. Bayardo, Jr. and R. C. Schrag. Using CSP Look-back Techniques to Solve Exceptionally Hard SAT Instances. In *Principles and Practice of Constraint Programming - CP96*, pages 46–60, Aug. 1996.

[15] S. P. Sample, M. R. D'Amour, and T. S. Payne. US Patent US5329470, Reconfigurable Hardware Emulation System, July 1994.

[16] 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.

[17] P. Stephan, R. Brayton, and A. Sangiovanni-Vincentelli. *Combinational Test Generation Using Satisfiability.* Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, 1992. UCB/ERL Memo M92/112.

[18] 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.

[19] J. S. Swartz, V. Betz, and J. Rose. A Fast Routability-Driven Router for FPGAs. In *FPGA'98 1998 ACM/SIGDA Sixth International Symposium on Field Programmable Gate Arrays*, pages 140–149, Feb. 1998.

[20] J. Varghese, M. Butts, and J. Batcheller. An Efficient Logic Emulation System. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 1(2):171–174, June 1993.

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

[22] P. Zhong, M. Martonosi, P. Ashar, and S. Malik. Solving Boolean Satisfiability with Dynamic Hardware Configurations. In *Field Programmable Logic and Applications, FPL '98*, Aug. 1998.