A Massively-Parallel Easily-Scalable Satisfiability Solver Using Reconfigurable Hardware

Miron Abramovici **Jose T. de Sousa** **Daniel Saab**

Bell Labs - Lucent Technologies, Murray Hill, NJ 07974

Case Western Reserve University, Cleveland, Ohio 44106

miron@research.bell-labs.com sousa@research.bell-labs.com saab@alpha.cwru.edu

ABSTRACT: Satisfiability (SAT) is a computationally expensive algorithm central to many CAD and test applications. In this paper, we present the architecture of a new SAT solver using reconfigurable logic. Our main contributions include new forms of massive fine-grain parallelism and structured design techniques based on iterative logic arrays that reduce compilation times from hours to a few minutes. Our architecture is easily scalable. Our results show several orders of magnitude speed-up compared with a state-of-the-art software implementation, and with a prior SAT solver using reconfigurable hardware.

1. INTRODUCTION

The *satisfiability* (SAT) problem - given a boolean formula $F(x_1, x_2, \ldots, x_n)$, find an assignment of binary values to (a subset of the) variables, so that F is set to 1, or prove that no such assignment exists - is a central computer science problem[12][18]. Typically F is expressed as a product-of-sums which is also called *conjunctive normal form* (CNF). Here we review the terminology via an example: in the formula $F = (A + B) \wedge (\bar{A} + B) \wedge (A + \bar{B})$, we have two *variables* (A and B) and three *clauses*, each with two *literals*; the literals in the third clause are A and $\bar{B}$, where $\bar{B}$ is an *inverting literal* and A is a *non-inverting* one. The assignment (A=1, B=1) is a *satisfying assignment*, as it sets F=1. Hence F is *satisfiable*. The formula $F' = F \wedge (\bar{A} + \bar{B})$ is *unsatisfiable*.

SAT has many applications in CAD and test[11]; here we will mention only a subset. A system of boolean equations can be solved by SAT by merging all equations into one formula. CAD problems that require solving large systems of boolean equations include timing verification[7][15], layout, and routability analysis[6][23]. Automatic test-pattern generation (ATPG) may be formulated as a SAT problem[14][21]. Computing the maximum circuit delay in the presence of false paths can also be solved by SAT[20]. Many problems in logic synthesis[3] - such as state assignment, state minimization, I/O encoding and asynchronous circuit design[13] - have SAT-based solutions.

It is well-known that SAT is an NP-complete problem[4]. Even with the most advanced SAT algorithms, such as GRASP[19], difficult problems may require many hours of computation. In the DIMACS set of SAT benchmarks[8], there are still several problems so difficult that, to the best of our knowledge, no SAT algorithm has ever been able to solve them. Applied to complex VLSI circuits, SAT-based algorithms have long run-times. Thus speeding up SAT will result in improving the efficiency of many CAD and test algorithms relying on SAT.

2. PREVIOUS WORK

Recently, several research groups have explored different approaches to implement SAT on reconfigurable hardware[22][1][24][17][25][16][2]. Figure 1 illustrates the general data flow of such an approach, whose goal is to speed up an algorithm *ALG* working on a given circuit *C*. A mapping program generates the model of a new circuit *ALG(C)*, which executes *ALG* for *C*. Since *ALG(C)* will be used only once, it is not economically feasible to actually construct it. Using reconfigurable hardware allows one to “virtually” create *ALG(C)*, then execute the algorithm by emulating this circuit. In contrast to a hardware accelerator for *ALG* (for example, a simulation accelerator), where the same special-purpose hardware processes different circuits, in this approach the *ALG(C)* hardware is designed specifically for a single circuit. The advantage is that *ALG will run at emulation speed, without incurring the cost of building special-purpose hardware*.

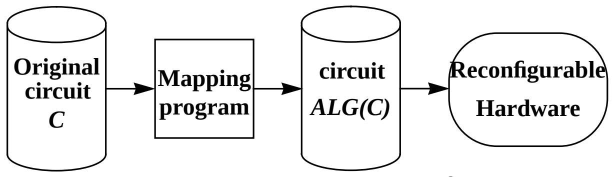

Figure 1. Speeding up algorithm *ALG* for circuit *C*

In the following, we will refer to a *SAT(C)* circuit as a *hardware SAT solver*, or a ***satisfier***. Suyama *et al.* [22] were the first to create a satisfier, which is downloaded into an emulator for execution. Although this method can be more efficient than a software SAT solver, its SAT algorithm is primitive and its applicability to CAD is limited because the generated vectors are always fully specified. This overspecification is detrimental in most CAD and test applications; for example, in ATPG, full specification would preclude test set compaction or generating the obtained vector by a different circuit.

The satisfier of Zhong *et al.*[24][25] implements a version of the classical Davis-Putnam SAT algorithm[5]. For every variable *i*, they construct an implication circuit and a state machine to manage the decision process for *i*. The implication circuit detects the conditions when values of other variables imply a value for *i*, and detects a conflict when both 0 and 1 are implied for *i*. The decision state machine keeps track of the current value of *i* (0, 1, or *x* for unassigned/unknown), and of the way the value has been obtained (assigned or implied). The decision state machines of all variables are connected as cells in a serial chain, where only one cell at a time may be enabled. The leftmost enabled cell whose variable *i* has an unknown value first assigns *i*=1. All the implications of this assignment are determined, and if no conflicts are detected, the next cell to the right is enabled. If *i*=1 leads to conflicts, the assignment *i*=0 is tried next. When both 0 and 1 lead to conflicts, *i* is set to *x* and control is passed to the left (backtracking). A cell whose value has been implied simply passes control to the right for normal processing or to the left for backtracking. The architecture has been implemented using an IKOS emulator. A proposed extension[25] implements a sophisticated non-chronological backtracking mechanism. The published results show a median speed-up of 64 on a subset (about 60%) of the DIMACS SAT benchmarks[8], compared with a software SAT solver constrained to run a similar algorithm.

An important problem in any SAT solver is the method used to select the next decision variable and its value to be tried. In software, the most efficient techniques rely on dynamic (run-time) ordering of variables based on heuristic measures that allow selecting the best one[18][19]. Dynamic selection, however, has been considered too expensive to directly implement in hardware. What all hardware SAT solvers do is order the variables statically (as a preprocessing step) based on some heuristic functions. This predetermines the basic order in which the search space will be explored at run-time. The only way this ordering may be altered is by skipping certain variables based on run-time conditions.

The static order used in [25] ranks variables based on their fanout count and is used to arrange the variables in the serial chain. The only variables skipped at run-time are those already having a binary value, and for every decision variable the 1 value is tried before 0. This mechanism leads to unnecessary or detrimental decisions. For example, when variable *A* becomes enabled and *A* still has value *x*, *A*=1 will be tried, even if no assignment to *A* is really needed in the current state (if *A* feeds only already satisfied clauses), or even if 1 is the wrong choice for *A* (when only $\overline{A}$ literals appear in the currently unsatisfied clauses); this results in a lot of unnecessary backtracking. A satisfier architecture similar to [25] has been recently described by Platzner and De Micheli[16]. They overcome the unnecessary decisions in [25] by introducing additional clauses to identify the conditions that make a variable become a *don't care* and logic that avoids decisions on these variables.

The satisfier proposed by Abramovici and Saab[1] solves the *circuit SAT problem* - given a combinational circuit with output *F*, finding an input assignment to set *F*=1. SAT is solved by modeling a CNF by a 2-level circuit. The satisfier executes a line justification algorithm patterned after PODEM[10]. Central to this algorithm is the concept of *objective*, which is a desired assignment *l*=*v* of value *v* to line *l*, which currently has an unknown value *x*. Initially all lines are set to *x*. An objective may be achieved only by primary input (PI) assignments. A *backtrace* procedure propagates an objective *l*=*v* along a single path from *l* to a PI *i*, where all the lines along the path have value *x*, and determines a PI assignment $i=v_i$ that is likely to contribute to achieving *l*=*v*. A major innovation introduced in [1] is logic for backtracing of objectives, thus realizing a backward circuit traversal in hardware. The search process is performed in a central control unit, using a hardware stack to support the backtracking process. Backtracing inherently avoids both decisions on *don't care*'s and assigning wrong values as done in [25]. However, the lack of a regular structure makes this architecture difficult to scale with the size of the problem. Rashid *et al.*[17] discuss an implementation of this architecture using Xilinx 6200 series FPGAs. No results are reported in either [1] or [17].

An important issue affecting any algorithm implemented on reconfigurable hardware, is the *compilation time* spent in preparing the FPGA-based circuit to be emulated. This process involves multi-FPGA partitioning taking into account the existing board interconnect, and then, for every FPGA, technology mapping, placement, and routing. If these tasks require more time than solving the original problem in software, then no overall speed-up can be achieved. To realize a proper balance between the run-time of the FPGA physical design tools and the computational savings resulting from accelerating SAT, [25] suggests that one should use a satisfier only for really difficult problems for which it is worth spending even a couple of hours in compilation if this saves many more hours of computation.

The remainder of the paper is organized as follows. Section 3 summarizes our main contributions. Section 4 describes the architecture of our satisfier. Section 5 presents our preliminary results, and Section 6 concludes the paper.

3. MAIN CONTRIBUTIONS

In this paper, we first introduce a *new massively-parallel fine-grain satisfier architecture*. Like [1], our satisfier selects the next variable to assign based on backtracing objectives. But while in [1] only one objective is propagated along a single path and results in a unique variable assignment, the new architecture provides *new forms of massive parallelism - parallel backtracing of all objectives along all possible paths and concurrent assignments of several variables*. Conceptually, the parallel backtrace is similar to the multiple backtrace of the FAN algorithm[9]; but, unlike in software, where objectives must be processed serially, in hardware we backtrace all objectives concurrently. An important difference is that in places where FAN selects only one path to backtrace one objective, in hardware we allow *simultaneous exploration of all possible paths*. To make possible this massive parallel processing we introduce the novel concept of *objectives with different priorities*.

A *unate variable* has all its literals either complemented or not, and hence it can never cause a conflict. Another significant contribution in our approach is the *identification of dynamically unate variables.* After a variable is assigned, certain clauses become satisfied. All the unassigned literals of an already satisfied clause are said to be *dead*, because in the current state, their values can no longer influence that clause. When our satisfier detects that all inverting literals of a variable *A* have died, it immediately assigns *A*=1, because this will satisfy all the clauses containing *A* without causing any conflicts. Similarly, detecting that all non-inverting literals of *A* have died shows that *A* may be safely set to 0. Although *A* is not unate, we treat *A* as a variable that becomes unate in the current state *(dynamically unate)*. Identifying and assigning dynamically unate variables represents an *opportunistic assignment* which is treated like a *new type of implication*. Increasing the number of implications reduces the search space and results in significant speed-up.

An unassigned variable becomes *dead* when all its literals have died. Our satisfier identifies dead variables and never assigns them, thus avoiding the unnecessary decisions taken in [25]. Our dead variables are equivalent to the *don't care*'s of [16], but are identified by a simpler mechanism.

To overcome the high computational costs of conventional FPGA physical design tools, we have developed *modular design techniques using iterative logic array (ILA) structures* to obtain easy-to-compile circuits. Like in [17], we design several types of basic building blocks (including their internal placement and routing), and create a library of modules as ILA cells to be used by any satisfier. After the library modules have been created, *the complexity of the place-and-route procedure for an ILA grows only linearly with the size of the ILA.* For inter-ILA connections we use a conventional router, which works in an area of the FPGA where no logic has been placed. While an unstructured chip design ends up with many unrouted nets after 10-12 hours of CPU time, the same *circuit using ILA-based design techniques takes only a few minutes to successfully compile*. These techniques are applicable to any type of FPGA, and the reduction in compilation time they provide becomes even more significant when we need to compile a large number of FPGAs. *The ILA-based approach is inherently scalable*, since adding more cells to an ILA does not change its regular structure.

4. THE SATISFIER ARCHITECTURE

Figure 2 shows the high-level view of our satisfier. Variable Logic maintains the current values (0, 1, or *x*) of all variables. Their values are sent to Literal Logic, which distributes them as literal values to Clause Logic. Clause Logic computes the value of every clause and of the output function *F*, and also determines objectives for all the literals. These objectives are sent back to Literal Logic, which merges objectives arriving from different literals of the same

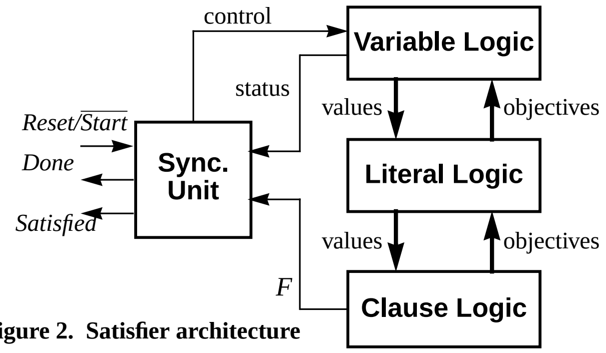

Figure 2. Satisfier architecture

variable into one objective for that variable. Both Clause Logic and Literal Logic are combinational blocks. Variable Logic maps the objectives arriving from Literal Logic into assignments (implications or decisions). The Synchronization Unit initiates backtracking when the function *F* becomes 0, performs some timing and control functions, and provides the interface with the outside world: *Reset/$\overline{Start}$* initiates the execution, *Done* is the completion signal, and *Satisfied* indicates whether the SAT problem has been successfully solved.

4.1 Concurrent Objectives

We model a CNF as a two-level circuit, as illustrated in Figure 3. Each PI represents a variable, each OR input represents a literal, and each OR gate represents a clause. Structurally, a variable is also referred to as a fanout stem.

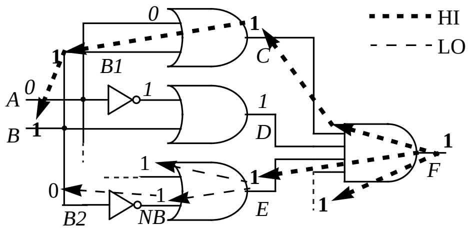

Figure 3. HI- and LO-objectives

To justify *F*=1, all the inputs of the AND gate must be 1. While in software line justification algorithms[10][9], these simultaneous requirements are processed one objective at a time, in our massively parallel approach all the objectives are concurrently processed. Allowing concurrent objective propagation along several paths means that different objectives may reach the same variable. Figure 3 shows that not all objectives are equally important. Here we assume that *A* has already been assigned value 0, which in turn satisfied clause *D* (logic values are italicized). All the objectives shown in bold are necessary to set *F*=1, while the 1 objectives at the inputs of *E* are not (since there are two alternative ways of setting *E*=1). We say that an objective has **high priority** if it *must be achieved* (in the current state) to set the function *F* to 1; the other objectives are said to have **low priority**. We will abbreviate "high- and low-priority objective" as "HI- and LO-objectives" respectively. Note that HI-objectives always form *continuous implication chains* (represented by bold arrows) starting with the *F* objective. Clearly, a HI-objective on a fanout branch of a stem should override any LO-objectives on other branches of the same variable. In Figure 3, the HI-1 objective on *B1* overrides LO-0 on *B2* and it is transmitted to *B*. HI-objectives reaching PIs denote *implications* and are mapped into value assignments for the corresponding variables in the next clock cycle. (Since every clause has always a HI-1 objective, these objectives are fixed in the logic and not propagated from *F*; but for the sake of clarity, we will show them as being propagated.)

Conflicts: Conflicting HI-objectives arriving at the same variable indicate an inconsistent state, because any binary value assigned to that PI in the current state would set the function *F* to 0 by reversing at least one of the implication chains arriving at the stem from *F*. Figure 4 illustrates such a case where the objectives at *B1* and *B2* are, HI-1 and HI-0.

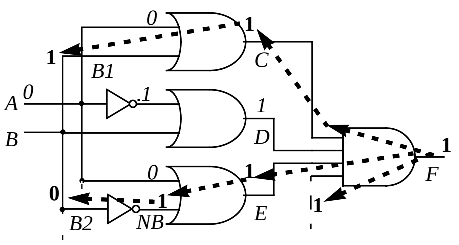

Figure 4. Conflicting HI-objectives

Here we would like to backtrack immediately to cut useless wandering in a no-solution area of the search space. This is achieved by same technique described above that maps any HI-objective reaching a PI into its corresponding variable assignment, because no matter which binary value is assigned to that PI, one of the implication chains will be reversed and the result will be *F*=0. This value of *F* is used in the Sync. Unit to initiate backtracking.

Potential conflicts: An additional priority level is useful to differentiate among LO-objectives. A LO-objective means that this objective is useful to achieve an upstream HI-objective, but in the current state there are alternative ways of achieving the same HI-objective. In Figure 5, all the OR inputs have LO-1 objectives, and both *A* and *B* receive two LO-1 and one LO-0 from their fanout branches. Clearly, this indicates potential for conflicts in the future. Here the choice of the objective value to propagate to the variable is arbitrary (*A* gets 0 and *B* gets 1), but we flag the variable objective as a *potential conflict* using a *. Variable Logic will select the next decision variable among the PIs with a potential conflict flag. Propagating LO-1 objectives from all inputs of the OR gates guarantees that *all potential conflicts are identified*.

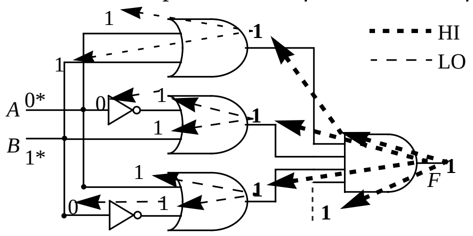

Figure 5. Potentially conflicting LO-objectives

Dead literals and dynamically unate variables: As a result of assigning *A*=1 in Figure 6, clause *E* becomes satisfied, and its other input becomes a *dead literal*. Although *B* is a binate variable, it may no longer cause a conflict; we say that *B* has become a *dynamically unate variable*. A dead literal is identified by a *dead objective*, denoted by ∅ in Figure 6. Clearly, the priority of ∅ should be the lowest. Then *B* is recognized as a dynamically unate variable because

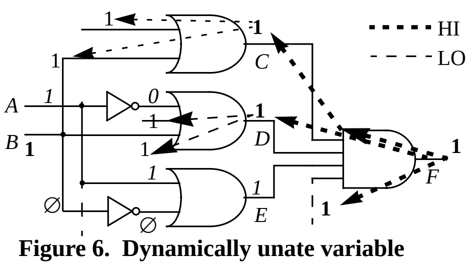

Figure 6. Dynamically unate variable

its fanout branches propagate only non-conflicting objectives (LO-1 and ∅). Although the value of *B* is not implied by the current state, we *opportunistically assign B*=1 to satisfy clauses *C* an *D* without causing any conflicts. To effectively treat *B*=1 as an implication, Literal Logic converts the non-conflicting LO-1 objectives into a HI-1, which will generate the implication *B*=1 in the next clock cycle.

In summary, our satisfier recognizes four priorities for objectives: HI, * (potential conflict), LO, and ∅ (dead).

4.2 Clause Logic

As illustrated in Figure 7, every clause with *m* literals is implemented by a bidirectional ILA of *m* OR2 cells. Every cell receives the value of one variable (V_{in}), a flag (*Inv*) indicating whether V_{in} should be inverted, and the partial OR result from the cells on its left (V_l), and computes $V_r = (V_{in} \oplus Inv) + V_l$ for the next cell on its right (using 3-valued logic). The V_r value obtained at the right-most OR2 cell is the clause output value, which is sent to an AND ILA to iteratively compute the value of the function *F*.

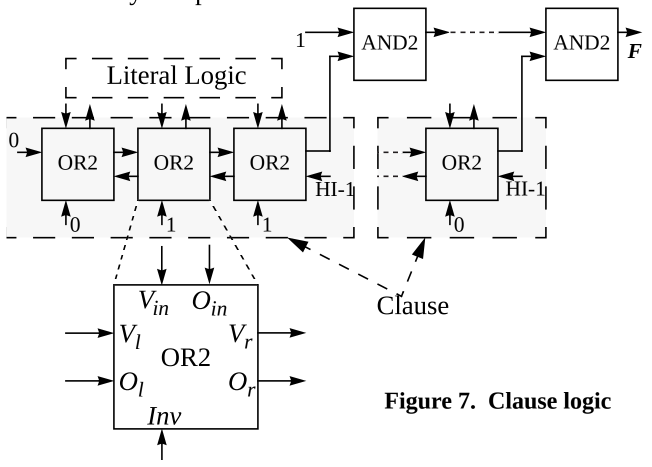

Figure 7. Clause logic

While values propagate left-to-right through the ILA, objectives advances in the opposite direction. Every OR2 cell receives its output objective O_r from the cell on its right (the right-most O_r is set to HI-1), and determines the objective for its input O_{in} and the output objective O_l for the cell to its left. In fact, the binary value of any (non-dead) objective is known apriori (1 for O_l and $\overline{Inv}$ for O_{in}) and it is hard-coded in the logic. The computation of priorities is given in Table 1, where H/L denotes a HI or LO priority.

O_r	V_l	V_{in}	O_l	O_{in}
∅	–	–	∅	∅
–	1	–	∅	∅
	–	1		
	0	0		
H/L	*x*	*x*	LO	LO
	x	0	H/L	∅
	0	*x*	∅	H/L

Table 1. OR2 objectives

4.3 Literal Logic

Since he propagation of the variable values to literals is straightforward, we will discuss only the process of computing an objective for a variable from the objectives of its literals. As illustrated in Figure 8, we model a stem as a sequence of stems with two fanout branches. The stem objective is computed by an ILA composed of ST2 cells which

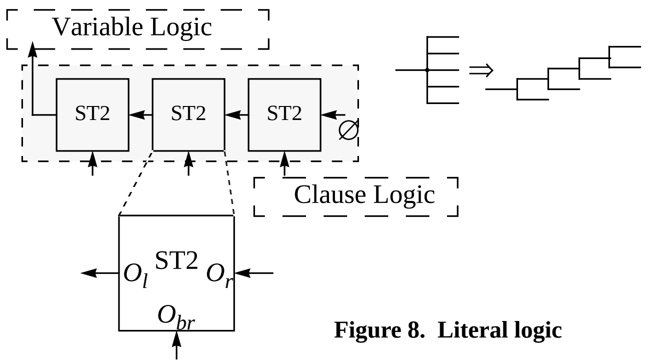

Figure 8. Literal logic

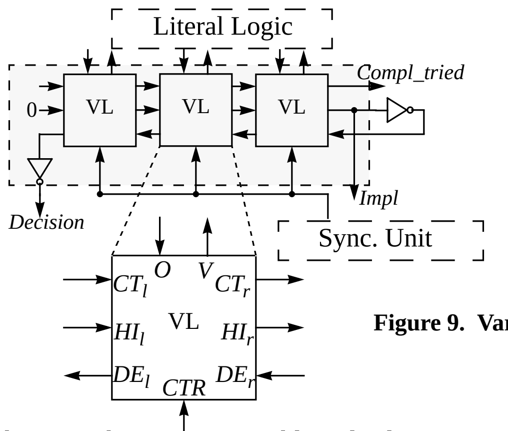

Figure 9. Variable logic

iteratively merge the objectives arriving on fanout branches from Clause Logic. Every ST2 cell receives the partial result O_r from the cell on its right and the objective of one fanout branch O_{br}, and computes the objective for its stem O_l, which is sent to the next cell on its left. The O_l output from the left-most cell is the PI objective sent to Variable Logic. The rules for computing objectives are given in Table 2, where v and a are arbitrary objective values, and H/L/*∈ {HI, LO, *}. Note that O_{br} cannot be a potential conflict (which may be generated only by Literal Logic), and that O_{br} is given priority over O_r in the case both have HI-priority. A conflict between HI-objectives will be detected because any value assigned to the stem will cause Clause Logic to output F=0.

O_r	O_{br}	O_l	*Notes*
∅	∅	∅	same objectives
H/L-v	H/L-v	H/L-v	
-	HI-v	HI-v	HI overrides any objective
HI-v	LO-a		
∅	H/L-v	H/L-v	any objective overrides ∅
H/L/*-v	∅	H/L/*-v	
LO-$\bar{v}$	LO-v	*-v	pot. conflict
*-v	LO-a		* overrides LO

Table 2. Variable objectives

4.4 Variable Logic

As illustrated in Figure 9, Variable Logic is constructed as a bidirectional ILA of VL cells. Every cell maintains its variable value V in a 2-bit register, and receives the objective O from Literal Logic as a 3-bit field (2 bits for priority and 1 for value). All cells with HI-objectives are assigned in the next clock cycle. A LO-objective shows that this variable has become dynamically unate; its priority is immediately converted to HI so that it will be treated as an implication. A *-objective denotes a potential conflict; if no variable must be implied, one of the variables with a *-objective will be selected as the next decision variable.

Since all implications must be done before any decision is tried, the ILA iteratively determines whether HI-objectives are present anywhere in the array. For this, every cell computes a HI_r flag signaling whether its objective O or any of the objectives of the cells to its left has HI-priority; the result from the preceding cells is brought in by the input HI_l. The signal *Impl* obtained at the HI_r output from the right-most cell reports whether at least one variable is being implied. Since no decisions should be made while implications are in progress, *Impl* is complemented and fed back to the right-most cell as a decision-enable input (DE_r). A cell receiving a dead objective (O=∅), because either its value is binary or because its variable is dead, just passes the decision enable signal through, i.e., DE_l=DE_r. If the objective O of a cell denotes a potential conflict and the cell is enabled to take a decision (has DE_r=1), then its variable will be the next decision variable, so it disables decisions for the cells to its left by setting its DE_l output to 0. The complement of the DE_l output from the left-most cell is the signal *Decision* which indicates whether any cell is taking a decision.

The control of the satisfier is distributed among the state machines in the VL cells and Sync. Unit. Unlike the central stack used in [1], our architecture simply equips each VL cell with an up/down counter to keep track of the decision level. The state of a cell is encoded in a 2-bit state register, which can only be updated if the cell is at the current decision level. All counters are concurrently incremented or decremented as specified by the *CTR* inputs. Such a distributed control mechanism is easily-scalable.

5. EXPERIMENTAL RESULTS

Figure 10 shows the layout of a satisfier for a formula having 13 variables, 29 clauses, and 69 literals, using an XC6264 FPGA. The layout is organized in alternating columns of clause logic and variable/literal logic. The space between these columns is reserved for the routing between the ST2 blocks of literals and the OR2 blocks of clauses. It is this routing that takes most of the compilation time, since the routing of the ILAs themselves is trivial. Nevertheless, the empty space reserved between columns guarantees efficient routing. No automatic place and route tool would achieve the compilation speed and compactness of this layout. The block in the lower left corner is the Sync. Unit, whose size does not vary with the SAT instance being solved. The layout took about 3 minutes to be compile from its original CNF file. In contrast, the unstructured version of the same satisfier (that is, without using ILA-based design) could never be successfully compiled (after 10 hours of CPU time, there were still more than 200 of unrouted nets). We determined that a satisfier that occupies the whole chip area can be clocked with a main clock frequency f of about 3.5MHz.

We use examples extracted from the DIMACS set of SAT benchmarks[8]. In Table 3 we compare the results of our satisfier with GRASP[19], one of the most efficient software SAT solvers available. Because of the small capacity of our current hardware platform, the results of the satisfier are

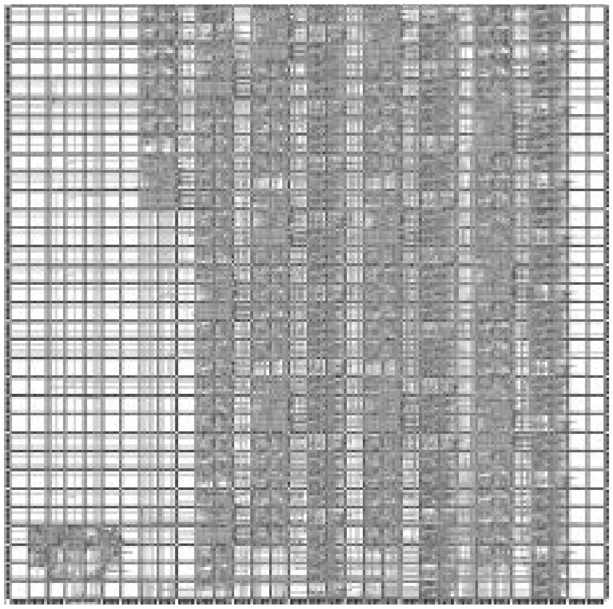

Figure 10. Satisfier layout on XC6264

obtained using a C model whose correctness was verified against a VHDL model which accurately represents the hardware implementation. The first 5 columns in Table 3 show the benchmark data: name, number of inputs #I, number of clauses #C, number of literals #L, and a Yes/No indication of satisfiability. The following columns show the time taken by our satisfier in number of main clock cycles *T*, the time taken by GRASP using the same time unit, and the speed-up *SU* gained with our approach. The main clock frequency is 3.5MHz. GRASP was run on a Sun Ultra Sparc workstation with 1026Mb RAM using a 248MHz clock. Thus GRASP is running with a clock about 83 times faster than the emulation clock.Unlike in [25], where GRASP was run in a restricted mode, so that it matched the features implemented in hardware, we allowed GRASP to run "full-speed," using all its sophisticated techniques. Our experience has shown that for many instances, the restrictive mode slows down GRASP by several orders of magnitude. GRASP was allowed to run each example for up to one hour (equivalent to 12.6G clock cycles on our satisfier), before aborting the execution. For 11 examples out of 20, our satisfier achieved significant speed-ups between 78 and 7,000, and for 3 instances the speed-up was in the 1.5 to 2.8 range. For 6 examples GRASP was faster than our satisfier. This is due to its sophisticated search features that do not have a match in our satisfier.

Benchmark	*#I*	*#C*	*#L*	Sat	Clocks	Clocks$_G$	*SU*
aim-50-1_6-no-2	50	80	240	N	75,415	21,000	0.28
aim-50-1_6-no-3	50	80	240	N	51,900	17,500	0.34
aim-100-1_6-no-2	100	160	480	N	6,570K	52,500	0.01
aim-100-1_6-yes1-3	100	160	480	Y	1,711K	42,000	0.02
aim100-6_0-yes1-1	100	600	1800	Y	12,388	129,500	10.4
aim200-6_0-yes1-1	200	1200	3600	Y	942,830	2,033,500	2.2
dubois20	60	160	480	N	10,486K	199,500	0.02
hole8	72	297	648	N	259,519	302,834K	1166.9
hole9	90	415	900	N	2,336K	12.3G	5,270.0
hole10	110	561	1210	N	23,357K	>12.6G	>539.3
ii8a2	180	800	2052	Y	60,587	168,000	2.8
ii32c1	225	1280	6081	Y	38	150,500	3960.5
ii32d2	404	5153	17940	Y	31,701	5,586K	176.2
jnh1	100	850	4392	Y	3,879	304,500	78.5
par8-1-c	64	254	732	Y	118	28,000	237.3
par16-1-c	317	1264	3670	Y	1,133K	748,160K	660.0
par16-2-c	349	1392	4054	Y	703,007	4.93G	7,000.0
par16-5	1015	3358	8980	Y	1,750K	814,415K	465.4
pret60_25	60	160	480	N	28,512K	210,000	0.01
ssa432-003	435	1027	2364	N	86,496	126,000	1.5

Table 3. Comparison to GRASP

Table 4 shows a comparison against the reconfigurable hardware satisfier described in [24] and [25]. To reproduce their results, we simply switched off our features for dealing with dead variables and dynamically unate variables, which do not exist in their approach. Thus the compared satisfiers use the same static ordering and the same implication mechanism, which results in a fair comparison. Column *H* represents the "hardware cost" of our satisfier, defined as in [24] as the total number gates and flip-flops; H_P is the corresponding cost in [24], and *HO* is the ratio H/H_P, given only for the examples that are common in the two papers. Our hardware cost is between 1.1 and 2.6 times greater than [24]. However, this buys us a speed-up of 1 to 2 orders of magnitude on 8 out of the 20 examples from Table 3 (*SU*=1 for the examples not included in Table 4). Several runs bypassing a threshold of 100M clocks were aborted. Note that we are not including the speed-up that results from our faster compilation time: according to [25], their compilation time for one FPGA is about 40 minutes, one order of magnitude greater than our 3 minutes.

Benchmark	*H*	H_P	*HO*	Clocks	Clocks$_P$	*SU*
aim-50-1_6-no-2	12,854	---	---	75,415	2,885K	38
aim-50-1_6-no-3	12,854	---	---	51,900	4,162K	80
aim-100-1_6-no-2	25,554	---	---	6,570K	>100M	>15
aim-100-1_6-yes1-3	25,554	---	---	1,711K	>100M	>58
aim-200-6_0-yes1-1	165,354	100,453	1.6	942,830	942,830	1
hole8	32,113	---	---	259,519	2,226K	9
hole9	44,099	---	---	2,336K	28,912K	12
hole10	58,751	21,872	2.6	23,357K	>100M	>4
ii8a2	98,166	37,959	2.5	60,587	4,325K	71
ii32d2	790,349	---	---	31,701	854,807	27
par16-1-c	174,054	80,215	2.1	1,134K	1,134K	1
ssa432-003	124,277	103,709	1.1	86,496	86,496	1

Table 4. Comparison to [24]

6. CONCLUSIONS

In this paper we have introduced a *new satisfier architecture*, using *new forms of fine-grain massive parallelism* to accelerate a SAT solver implemented on reconfigurable hardware: *parallel backtracing of multiple objectives along all possible paths and concurrent assignments of several variables*. This massive parallel processing is facilitated by *objective propagation with several different priorities*. Our satisfier identifies *dynamically unate variables* and *dead variable*s. These techniques generate more implications, avoid wrong and unnecessary decisions, and reduce the amount of backtracking. Our results show several orders of magnitude speed-up compared with both a state-of-the-art software SAT solver and a previous satisfier. Objectives are a flexible mechanism that can be easily extended to support additional algorithmic improvements, such as dynamic variable selection, that we recently implemented (these results will be reported in the future).

We have developed *modular design techniques using ILA structures*, thus overcoming the high computational costs of conventional FPGA physical design tools. While an unstructured chip design ends up with many unrouted nets after 10-12 hours of CPU time, the same circuit using ILA-based design techniques takes only a few minutes to successfully compile. These techniques are applicable to any type of FPGA, and the reduction in compilation time they provide becomes even more significant when we need to compile a large number of FPGAs. Another advantage of the ILA-based approach is its *inherent scalability*.

In a related paper[2], we have introduced a *virtual logic system* that allows a reconfigurable logic platform to solve SAT problems much larger than its available capacity. It relies on *novel decomposition techniques* to divide a formula into independent subproblems that can be run by different FPGAs in any order. The unusual feature of our decomposition is that *inter-FPGA signals are never required.* Unlike the multi-FPGA partitioning used in a conventional design flow, our decomposition is independent of the reconfigurable hardware architecture. When the number of subproblems is larger than the number of available FPGAs, we simply reuse the same FPGA to solve in turn several subproblems. The FPGAs can be run concurrently, introducing a new level of course-grain parallelism, and resulting in *up to three orders of magnitude additional speed-up,* which compensates for the time spent in decomposition.

7. REFERENCES

[1] M. Abramovici and D. Saab, "Satisfiability On Reconfigurable Hardware," *Proc. Intn'l. Workshop on Field-Programmable Logic and Applications*, Sept., 1997

[2] Miron Abramovici, J. T. de Sousa, "A Virtual Logic System for Solving Satisfiability Problems Using Reconfigurable Hardware," to appear in *Proc. Symp. on Field-Programmable Custom Computing Machines*, 1999

[3] R. Brayton, G. Hachtel, C. McMullen, and A. Sangiovanni-Vincentelli, *Logic Minimization Algorithms for VLSI Synthesis*, Kluwer Academic Publishers, 1984

[4] S. A. Cook, "The Complexity of Theorem-Proving Procedures," *Proc. 3rd Annual ACM Symp. on Theory of Computation*, pp. 151-158, 1971

[5] M. Davis and H. Putnam, "A Computing Procedure for Quantification Theory," *Journal of the ACM*, vol. 7, pp. 167--187, 1960

[6] S. Devadas, "Optimal Layout Via Boolean Satisfiability," *Proc. Intn'l. Conf. on CAD*, pp. 294-297, November 1989

[7] S. Devadas, K. Keutzer, S. Malik, and A. Wang, "Certified Timing Verification and the Transition Delay of a Logic Circuit," *Proc. Design Automation Conf.*, pp. 549-555, June, 1992

[8] DIMACS Challenge Benchmarks, *ftp://dimacs.rutgers.edu/pub/challenge/sat/benchmarks/cnf/*

[9] H. Fujiwara and T. Shimono, "On the Acceleration of Test Generation Algorithms," *IEEE Trans. on Computers*, vol. C-32, no 12, pp. 1137-1144, December, 1983.

[10] P. Goel, "An Implicit Enumeration Algorithm to Generate Tests for Combinational Logic Circuits," *IEEE Trans. on Computers*, Vol. C-30, No. 3, pp. 215-222, March, 1981.

[11] J. Gu, "Satisfiability Problems in VLSI Engineering," *DIMACS Workshop* on *Satisfiability Problem: Theory and Applications*, March 1996

[12] J. Gu, P. W. Purdom, J. Franco, and B. W. Wah, "Algorithms for the Satisfiability (SAT) Problem: A Survey," *DIMACS Workshop* on *Satisfiability Problem: Theory and Applications*, pp. 19-51, March 1996

[13] J. Gu and R. Puri, "Asynchronous Circuit Synthesis with Boolean Satisfiability", *IEEE Trans. on CAD*, Vol. 14, No. 8, pp. 961-973, August 1995

[14] T. Larrabee, "Test Pattern Generation Using Boolean Satisfiability," *IEEE Trans. on CAD*, Vol. 11, No. 1, pp. 4-15, January, 1992

[15] P. C. McGeer et al., "Timing Analysis and Delay-Fault Test Generation Using Path Recursive Functions," *Proc. Intn'l. Conf. on CAD*, pp. 180-183, November 1991

[16] M. Platzner and G. De Micheli, "Acceleration of Satisfiability Algorithms by Reconfigurable Hardware," *Proc. Intn'l. Workshop on Field-Programmable Logic and Applications*, Sept., 1998

[17] A. Rashid, J. Leonard, and W.H. Mangione-Smith, "Dynamic Circuit Generation for Solving Specific Problem Instances of Boolean Satisfiability," *Proc. IEEE Symp. on Field-Programmable Custom Computing Machines*, April 1998

[18] J. M. Silva, "An Overview of Backtrack Search Satisfiability Algorithms," *Proc. 5th Intn'l. Symp. on Artificial Intelligence and Mathematics*, January 1998

[19] J. M. Silva and K. A. Sakallah, "GRASP - A New Search Algorithm for Satisfiability," *Proc. Intn'l. Conf. on CAD*, pp. 220-227, November 1996

[20] L. G. Silva et al., "Realistic Delay Modeling in Satisfiability-Based Timing Analysis," *Proc. Intn'l. Symp. on Circuits and Systems (ISCAS)*, May 1998

[21] P. R. Stephan, R. K. Brayton, and A. Sangiovanni-Vincentelli, "Combinational Test Generation Using Satisfiability," *IEEE Trans. on CAD*, vol. 15, no. 9, pp. 1167-1176, Sept. 1996.

[22] T. Suyama, M. Yokoo, and H. Sawada, "Solving Satisfiability Problems on FPGAs," *Proc. Intn'l. Workshop on Field-Programmable Logic and Applications*, 1996

[23] G. Nam, K. A. Sakallah, and R.A. Rutenbar, "Satisfiability-Based Layout Revisited: Routing Complex FPGAs Via Search-Based Boolean SAT", *Proc. Intn'l. Symp. on FPGAs*, February 1999

[24] P. Zhong, M. Martonosi, P. Ashar, and S. Malik, "Accelerating Boolean Satisfiability with Configurable Hardware," *Proc. IEEE Symp. on Field-Programmable Custom Computing Machines*, April, 1998

[25] P. Zhong, M. Martonosi, P. Ashar, and S. Malik, "Using Reconfigurable Computing Techniques to Accelerate Problems in the CAD Domain: A Case Study with Boolean Satisfiability," *Proc. Design Automation Conf.*, June 1998