Journal of Automated Reasoning **24:** 5–36, 2000.

A SAT Solver Using Reconfigurable Hardware and Virtual Logic

MIRON ABRAMOVICI and JOSE T. DE SOUSA
Bell Labs - Lucent Technologies, Murray Hill, NJ 07974.
e-mail: miron/sousa@research.bell-labs.com

Abstract. In this paper, we present the architecture of a new SAT solver using reconfigurable logic and a virtual logic scheme. Our main contributions include new forms of massive fine-grain parallelism, structured design techniques based on iterative logic arrays that reduce compilation times from hours to minutes, and a decomposition technique that creates independent subproblems that may be concurrently solved by unconnected FPGAs. The decomposition technique is the basis of the virtual logic scheme, since it allows solving problems that exceed the hardware capacity. Our architecture is easily scalable. Our results show several orders of magnitude speedup compared with a state-of-the-art software implementation, and also with respect to prior SAT solvers using reconfigurable hardware.

Key words: satisfiability, reconfigurable logic, virtual logic, massive parallelism.

1. Reconfigurable Computing

Reconfigurable computing is an emerging computing paradigm using reconfigurable hardware based on Field-Programmable Gate Arrays (FPGAs) to implement computationally intensive applications by tailoring the architecture to the needs of each application. An FPGA is a universal logic device structured as an array of programmable logic cells, interconnected by a programmable routing network, and programmable I/O cells. There are several methods of programming FPGAs to implement a specific function. For SRAM-based FPGAs (the most popular type of FPGAs), programming is simply done by writing the contents of a configuration memory contained in the device. The set of all programming bits establishes the *configuration* that determines the function of the device. A SRAM-based FPGA may be reconfigured an arbitrarily large number of times. A reconfigurable processor combines hardware speed with the flexibility of software and is usually attached as a coprocessor to a general-purpose host computer. The host prepares and downloads the data needed to configure the hardware as an application-specific processor. The reconfigurable logic benefits from eliminating or reducing the overhead of a general-purpose computer (instruction interpretation, fixed-width data-paths, limited register set, fixed functional units, etc.) and can be optimally designed for the target application by exploiting fine-grain par-

allelism. Unlike special-purpose accelerators (such as those used for simulation), reconfigurable processors support many different applications. To pick only one example, the same DEC reconfigurable processor [34] has been used to implement (1) long-integer arithmetic, (2) RSA cryptography, (3) DNA sequencing, (4) neural networks, (5) multistandard video compression, (6) solutions of Laplace equations, (7) image acquisition, analysis, and classification, (8) stereo vision, (9) sound synthesis, and (10) Viterbi decoding. In most applications, the reconfigurable processor achieved supercomputer-level performance, while its cost is only a small fraction of a supercomputer cost.

Emulation is an application of reconfigurable computing increasingly used in design verification of complex systems (for example, see [16]). With emulation, one builds an FPGA-based hardware model of a new VLSI chip; typically, such a model is only about 10 times slower than the real hardware (hence many orders of magnitude faster than software simulation), which makes possible in-system verification and hardware-software co-design.

2. Motivation

SAT is a central problem in computer science, with many applications in many different domains [19]. Our main motivation in developing a reconfigurable hardware SAT solver was to obtain a low-cost tool for solving difficult SAT instances that occur in the area of Computer Aided Design (CAD) and test of electronic circuits. CAD and test problems that require solving large systems of Boolean equations include logic verification, timing verification [13, 22], layout and routability analysis [12, 35], and fault diagnosis [36]. The Automatic Test Generation (ATPG) problem may be formulated as a SAT problem [21, 30]. Computing the maximum circuit delay in the presence of false paths can also be solved by SAT [29]. Many problems in logic synthesis [8] – such as state assignment, state minimization, I/O encoding and asynchronous circuit design [20] – have SAT-based solutions. Many other computationally difficult problems, such as graph coloring, scheduling, theorem proving, and constraint satisfaction problems, have also been mapped into SAT. Because of the NP-completeness of SAT [10], even with the most advanced SAT algorithms, such as GRASP [28], difficult problems may require many hours of computation. In the DIMACS set of SAT benchmarks [14], 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. Note that in this area we cannot use approximate SAT algorithms, since unsatisfiable formulas occur often, and only an exact algorithm is guaranteed to recognize unsatisfiability.

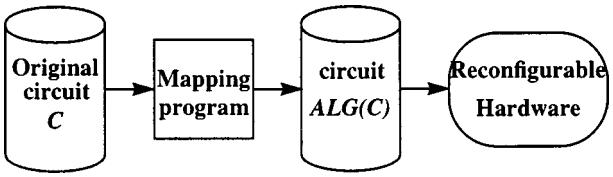

Figure 1. Speeding up algorithm *ALG* for circuit C.

3. Previous Work

Recently, several research groups have explored different approaches to implementing SAT on reconfigurable hardware [31, 2, 38, 25, 39, 24]. 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 would not be economically feasible to implement it in a conventional way. Using reconfigurable hardware allows one to "virtually" create the $ALG(C)$ circuit; then the algorithm is executed 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*. Similar paradigms based on reconfigurable computing techniques have also been used or proposed to accelerate other computationally difficult problems, such as finding independent sets [5] and transitive closure [9] in graphs.

In the following, we will refer to a $SAT(C)$ circuit as a *hardware SAT solver*, or a *satisfier*. Suyama et al. [31] were the first to create a satisfier, which is downloaded into an emulator for execution. Although this method has been reported to be more efficient than a software SAT solver, its applicability to CAD is limited because the generated vectors are always fully specified. For example, for a problem with 200 variables, all of them will be assigned binary values, even if the function may be satisfied, say, with only 20 assignments. This overspecification is detrimental in most CAD and test applications, where typically we would like to minimize the number of specified variables. For example, in ATPG, full specification would preclude test set compaction [1] and may also preclude the generation of the obtained vector by a different circuit.

The satisfier of Zhong et al. [38, 39] implements a version of the classical Davis–Putnam SAT algorithm [11]. For every variable x_i, they construct an implication circuit and a state machine to manage the decision process for x_i. The implication circuit detects the conditions when values of other variables imply a value for x_i, and detects a conflict when both 0 and 1 are implied for x_i. The decision state machine keeps track of the current value of x_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 x_i has an unknown value first assigns x_i to 1. All the implications of this assignment are determined (in several clock cycles), and if no conflicts are detected, the next cell to the right is enabled. If $x_i = 1$ leads to conflicts, the assignment $x_i = 0$ is tried next. When both 0 and 1 lead to conflicts, x_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 [39] implements a sophisticated nonchronological backtracking mechanism. The published results show a median speedup of 64 on a subset (about 60%) of the DIMACS SAT benchmarks [14].

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 [27, 28]. 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 [39] ranks variables based on their literal count and is used to arrange the variables in the serial chain. The only variables skipped at runtime 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 A literals appear in the currently unsatisfied clauses); this results in a lot of unnecessary backtracking. A satisfier architecture similar to [39] has been recently described by Platzner and De Micheli [24]. A proposed extension allows overcoming the problem of unnecessary decisions in [39] 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 [2] executes a *justification* algorithm patterned after PODEM [17]. Here, we have a combinational logic circuit that represents the Boolean function to be satisfied, and the goal is to set the primary output (PO) to 1 by a suitable assignment of the primary inputs (PIs). Central to this algorithm is the concept of *objective*, which is a desired assignment $l = v$ of value v to signal l, which currently has an unknown value x. Initially all signals are set to x. An objective may be achieved only by PI assignments. A *backtrace* procedure propagates an objective $l = v$ along a single path from l to a PI i, where all the signals 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 [2] 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 [39]. However, the central control unit appears difficult to scale up with the size of the target CNF. Rashid et al. [25] discuss an implementation of this architecture using Xilinx 6200 series FPGAs [37]. No significant results are reported in either [2] or [25].

An important issue affecting not only SAT but also 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 speedup can be achieved. To realize a proper balance between the runtime of the FPGA physical design tools and the computational savings resulting from accelerating SAT, Zhong et al. [39] suggest 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 (or days) of computation or if it allows solving problems that are impossible for a software SAT solver. Nevertheless, it would be a great advantage to have a more efficient design flow, less dependent on conventional FPGA design tools.

Another problem in accelerating SAT with reconfigurable logic is the capacity and the cost of the hardware platform used for emulation. All previous SAT solvers using reconfigurable hardware are limited in the size of the problems they can handle by the available hardware capacity; hence any problem that does not fit is automatically "out-of-bounds." Current logic emulators have large capacity (up to several million gates), but they are quite expensive (about $0.50 per emulated gate) and hence not easily affordable; however, most other platforms that are reasonably priced do not provide sufficient capacity. The ideal solution would be a *virtual logic* mechanism that allows a reconfigurable logic platform to process circuits much larger than its available capacity at the cost of some additional processing time. Note the analogy with *virtual memory*, where the logical address space is much larger than the physical address space.

4. Main Contributions

In this paper we survey our work in creating a new satisfier architecture [3, 4]. We first describe a *new massively parallel fine-grain satisfier architecture*. Like [2], our satisfier selects the next variable to assign based on backtracing objectives. But while in [2] 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 a certain extent) to the multiple backtrace of the FAN algorithm [15]; 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 introduced the new concept of *objective propagation with several different priorities*.

Another significant contribution in our approach is the *identification of dynamically unate variables*. A *unate variable* has all its literals either complemented or not, and hence it can never cause a conflict. The literals of a unate variable are known in the SAT literature as *pure literals* [11]. 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 the satisfied 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 noninverting literals of A have died shows that A may be safely set to 0. In other words, although A is not unate, we can treat A as a variable that becomes unate in the current state (*dynamically unate*). Identifying and assigning dynamically unate variables represents an *opportunistic assignment* that is treated like a *new type of implication*. Increasing the number of implications reduces the search space and results in significant speedup. This new technique can be viewed as the hardware embodiment of the Pure Literal Rule [11].

When all the literals of an unassigned variable A have died (because all the clauses containing A are satisfied), A is said to be a *dead variable*. Our satisfier identifies dead variables and never assigns them, thus avoiding the unnecessary decisions taken in [39]. Our dead variables are equivalent to the *don't care*'s of [24] but are identified by a different mechanism.

We have implemented our satisfier using the ACE6264 reconfigurable processor board produced by TSI-TelSys Inc. This board employs the XC6200 series FPGAs [37], witch have been designed to support reconfigurable computing applications. To overcome the high computational costs of conventional FPGA physical design tools, we have developed *modular design techniques using iterative logic array* (*ILA*) *structures* that trade off some chip area and performance to obtain easy-to-construct circuits (a carry-ripple adder is a typical example of ILA). As in [25], we predesign several types of basic building blocks (including their placement and routing) and create a library of modules to be used by any satisfier. The novel aspects of our approach are the design of the library modules as ILA cells and the placement of these blocks using specific ILA structures parametrized for a target CNF. After the library modules have been created, *the complexity of the place-and-route algorithms grows only linearly with the size of the ILAs. The ILA-based approach is inherently scalable*, since adding more cells to an ILA does not change its regular structure. While an unstructured chip design ends up with

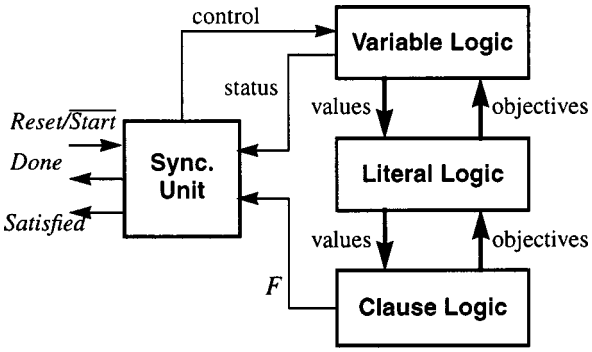

Figure 2. High-level view of the architecture.

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.

Multi-FPGA partitioning is necessary when the circuit to be emulated (in our case, the satisfier) does not fit into one FPGA. An additional complication is that such an FPGA partitioning algorithm must also deal with the constraints imposed by existing inter-FPGA board interconnect. Another significant contribution of this paper is *a method of decomposing a given SAT problem into several nondisjoint but nevertheless independent subproblems that may be solved in any order by unconnected FPGAs*, where every subproblem is assigned to one FPGA. Our decomposition technique combines partitioning into disjoint subproblems used in software SAT algorithms [27] with the simple variable decomposition previously used for running SAT on parallel processors [7]. Unlike circuit partitioning into multiple FPGAs, the unusual feature of our decomposition is that *inter-FPGA signals are not required.* This provides a *simple solution to the limited hardware capacity problem*: if we have more subproblems (n) than available FPGAs (k), we simply take turns in reusing every FPGA by reconfiguring it for n/k subproblems. This feature represents a *virtual logic mechanism*, which allows a reconfigurable logic platform to process circuits much larger than its available capacity. Additional advantages of our decomposition are discussed in Section 6.

The remainder of the paper is organized as follows. Section 5 presents the architecture of our satisfier. Section 6 describes our decomposition technique and the virtual logic scheme. Section 7 presents our experimental results. Section 8 discusses future research directions, and Section 9 concludes the paper.

5. The Satisfier Architecture

Figure 2 shows the high-level view of our satisfier. (More details will be given in the following subsections.) Variable Logic maintains the current values (0, 1, or x)

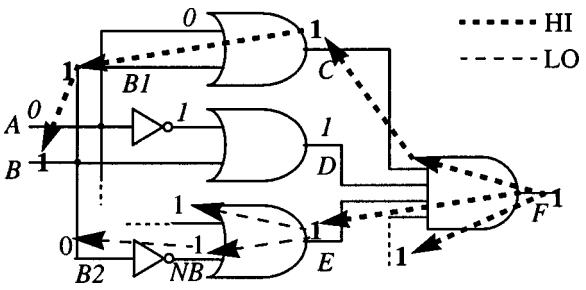

Figure 3. HI- and LO-objectives.

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 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.

5.1. 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. We treat satisfiability as a justification problem to set $F = 1$.

To justify $F = 1$, all the inputs of the AND gate must be 1. While in software justification algorithms [17, 15] (and also in the hardware scheme of [2]) 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 several objectives may reach the same variable, each one arriving from a different literal. Since these objectives may be conflicting, a first question should be: are all objectives equally important? The answer can be easily inferred by analyzing the objectives shown in Figure 3, where we assume that A has already been assigned value 0, which in turn satisfied clause D (logic values are italicized). Note that only signals with value x may have objectives. 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 $E = 1$). Thus 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

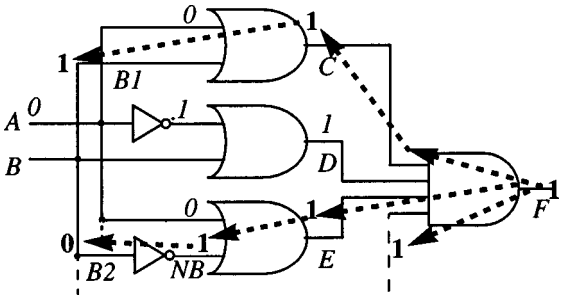

Figure 4. Conflicting HI-objectives.

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 PO 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 the stem 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 the PO; but for the sake of clarity, we will show them as being propagated.) While in [2], processing of the HI-1 objective of E would result in only one input with value x being assigned a LO-1 objective, in our approach LO-1 objectives are simultaneously sent to every input of E with x value. The importance of this difference will be explained shortly.

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 the PO. Figure 4 illustrates such a case where the objectives at *B1* and *B2* are, respectively, HI-1 and HI-0. In such a situation we would like to backtrack immediately to cut useless wandering in a no-solution area of the search space. This is achieved by the 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,

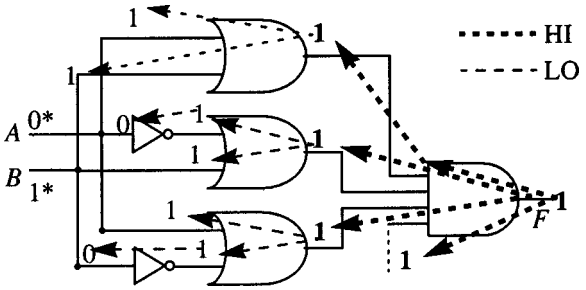

Figure 5. Potentially conflicting LO-objectives.

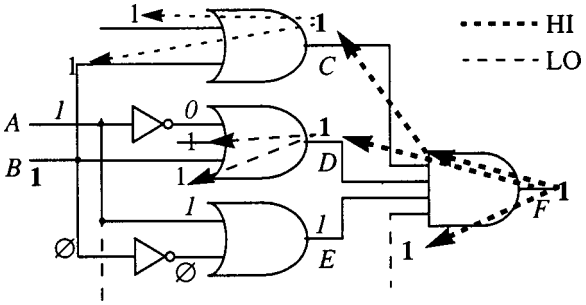

Figure 6. Dynamically unate variable.

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*.

Dead Literals and Dynamically Unate Variables. As a result of assigning $A = 1$ in Figure 6, clause E becomes satisfied. Then the value of its other input becomes a "don't care", and this input will be referred to as a *dead literal*. Although B is a binate variable, it may no longer cause a conflict in the current state; we say that B has become a *dynamically unate variable*. To transmit the information about dead literals from the satisfied clauses back to the variables generating the dead literals, we introduce a *dead objective*, denoted by $\oslash$ in Figure 6. Clearly, the priority of $\oslash$ should be the lowest. Then B is recognized as a dynamically unate variable because its fanout branches propagate only nonconflicting objectives (LO-1 and $\oslash$). Although the value of B is not implied by the current state, we will *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 nonconflicting LO-1 objective into a HI-1, which will generate the implication $B = 1$ in the next clock cycle. This process will continue, since the newly satisfied clauses will create

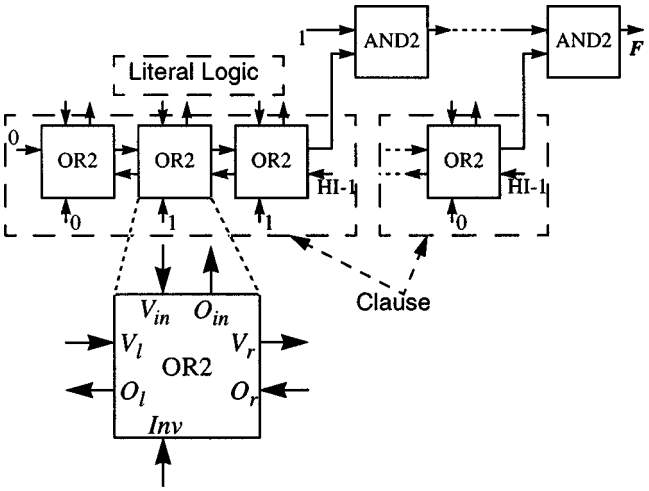

Figure 7. Clause logic.

new dead literals, which in turn may cause other variables to become dynamically unate, and so on.

In summary, our satisfier recognizes four priorities for objectives: HI, $*$ (potential conflict), LO, and $\oslash$ (dead). These are coded with two bits.

5.2. 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.

While values propagate left to right through the ILA, the computation of input objectives advances in the opposite direction. Every OR2 cell receives its output objective O_r from the cell on its right (O_r is initialized to HI-1 at the rightmost cell) 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 (nondead) objective is known a priori (1 for O_l and $\overline{Inv}$ for O_{in}) and it is hard-coded in the logic. Thus only the computation of priorities is relevant, as given in Table I, where H/L denotes a HI or LO priority and "-" denotes a don't care entry.

5.3. LITERAL LOGIC

Since the propagation of the variable values to literals is straightforward, we will discuss only the process of computing an objective for a variable from the ob-

Table I. OR2 objective priority computation.

O_r	V_l	V_{in}	O_l	O_{in}
⊘	—	—	⊘	⊘
	1	—		
—	—	1	⊘	⊘
	0	0		
	x	x	LO	LO
H/L	x	0	H/L	⊘
	0	x	⊘	H/L

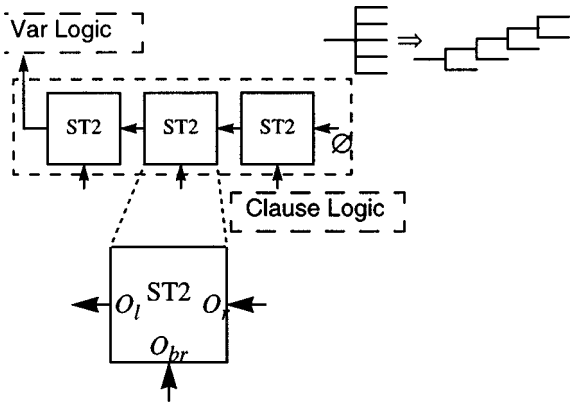

Figure 8. Literal logic.

jectives 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 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 II, where v and a are arbitrary objective values, and H/L/$^* \in$ {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 $F = 0$.

Table II. Variable objectives.

O_r	V_{br}	O_l	Notes
$\oslash$ H/L-v	$\oslash$ H/L-v	$\oslash$ H/L-v	same objective
— HI-v	HI-v LO-a	HI-v	HI overrides any objective
$\oslash$ H/L/*-v	H/L-v $\oslash$	H/L-v H/L/*-v	any objective overrides $\oslash$
LO-$\overline{v}$ *-v	LO-v LO-a	*-v	pot. conflict * overrides LO

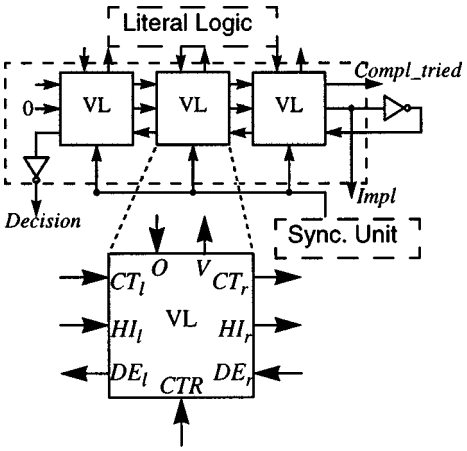

Figure 9. Variable logic.

5.4. VARIABLE LOGIC

As illustrated in Figure 9, Variable Logic is constructed as a bidirectional ILA of VL cells, where each cell corresponds to a variable. Every cell maintains its current 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). A HI-objective indicates that the objective value must be assigned to this variable as an implication in the next clock cycle. All cells with HI-objectives are concurrently assigned. 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. Finally, a dead objective denotes a variable that should do nothing.

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 ($O = \oslash$), because either its value is binary or its variable is dead, just passes the decision enable signal through, that is, $DE_l = DE_r$. 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. Both *Decision* and *Impl* being 0 means that all arriving objectives are $\oslash$, since any nondead objective would produce either an implication or a decision; this may occur only when the function F has a binary value.

Each VL cell is a state machine with four states: (1) Unset (initial state with $V = x$), (2) Implied (V implied from a HI-objective), (3) Assigned (V assigned from a *-objective by a decision), and (4) Complemented (V is the result of reversing the last decision of this cell). Being in the Complemented state results in asserting the output CT_r if the variable is the current decision variable. Thus one can OR the left input CT_l with a flag indicating that the cell is the current decision variable and is in the Complemented state, to obtain the output CT_r. The signal *Compl_tried*, obtained at the CT_r output from the right-most cell, reports whether the decision variable has been complemented. *Compl_tried*, *Decision*, and *Impl* are sent to Sync. Unit to report the status of Variable Logic. State transitions in a VL cell are controlled by its arriving objective O, its decision-enable input DE_r, and the 2-bit control bus *CTR*, which is generated by Sync. Unit and shared among all cells.

Unlike the central stack used in [2], in our architecture each VL cell has an up/down counter *DL* to keep track of the current decision level, and a register *AL* to store the level at the time when its variable has been assigned (by a decision or an implication). The state of a cell is encoded in a two-bit state register, which can only be updated if the cell is at the current decision level ($AL = DL$). All *DL* counters are concurrently incremented or decremented as specified by the *CTR* inputs. (The *DL* counters are identical, but their replication simplifies routing.) Such a distributed control mechanism is easily scalable.

5.5. SYNCHRONIZATION UNIT

Sync. Unit determines the state of the satisfier using the signal F from the Clause Logic, the signals *Impl*, *Decision*, and *Compl_tried* from Variable Logic, and an-

other replica of the *DL* counter. From these signals it can tell if the problem has been satisfied ($F = 1$), if the problem is unsatisfiable ($DL = 0$ and there is still need to backtrack), or if execution should continue. In the latter case, different actions are issued to the VL cells via *CTR*. $F = 0$ indicates the need to complement the current decision or backtrack to the previous one, if *Compl_tried* is asserted. If $F = x$, we can (1) have a new decision, (2) have implications resulting from the previous decision, or (3) be in between consecutive backtracks. In the first two cases new variable assignments will be caused, and in the third case we have to decrement *DL* again.

Another function of Sync. Unit is to provide clocking signals to the rest of the circuit. After new variables are set, the longest delay until *Decision* becomes stable involves two traversals through the VL ILA. This delay T can be determined from post-layout timing analysis, and is programmed into Sync. Unit. We call $f_1 = 1/T$ the main clock frequency. We also have a secondary, faster frequency f_2, which we use for consecutive decrement operations of *DL*. Note that no variable assignments are made between consecutive *DL* decrement operations; we just need to wait for the *Compl_tried* signal to propagate through the VL array. Usually $f_2 \cong 2 \times f_1$. If $N1$ and $N2$ are the respective numbers of clock pulses of frequencies f_1 and f_2 used during a run, we will report the result by the number of equivalent main clocks $CK = N1 + N2/2$. Because of the ILA nature of our satisfier, f_1 is quite low (below 5 MHz), compared with more conventional designs.

6. SAT Decomposition

6.1. SIMPLE DECOMPOSITION

In [7], an original SAT problem is decomposed into n subproblems, which are then distributed to n conventional processors, and interprocessor communications are used for load balancing. We reuse this technique for the case of reconfigurable hardware satisfiers. The principle underlying this decomposition is well known: we split the problem $F(x_1, x_2, \ldots, x_n) = 1$ into two subproblems obtained by setting one variable to 0 and 1, respectively. If we split on x_1, the two subproblems will be $F(0, x_2, \ldots, x_n) = 1$ and $F(1, x_2, \ldots, x_n) = 1$. Clearly, any solution to the first (second) subproblem is also a solution to the original problem if we add $x_1 = 0(1)$. To conclude that the original problem is unsatisfiable, every subproblem must be proven unsatisfiable. The CNF for a subproblem is a subset of the original CNF; the simplification process based on assigning one variable will be described in the next section. Thus any subproblem is smaller than the original problem. We can recursively repeat this splitting process for k variables. If we represent this decomposition process by a binary OR tree with k levels, the subproblems to be solved are obtained at the terminal nodes of the tree. The OR relation between the two branches of a node means that solving one of the two subproblems is sufficient to solve the original problem. The more variables we assign, the smaller the subproblems become. Of course, k should be limited since the number of subproblems

is 2^k. *The subproblems share all the variables not assigned, so they are not disjoint. Nevertheless, they are independent*, since the values of the shared variables do not have to be correlated, as different subproblems may obtain different satisfying vectors for the original problem.

Let us assume that we continue to assign variables until each subproblem fits into one FPGA. Unlike circuit partitioning into multiple FPGAs, the unusual feature of this decomposition is that *inter-FPGA signals are never required.* The advantages of not having any inter-FPGA interconnect are very significant. First, the conventional *multi-FPGA partitioning step is no longer needed,* and hence the design flow becomes simpler and faster. Second, the *large delays* associated with signals propagating off-chip, on the board, and back on-chip *are avoided,* which allows a faster clock speed. Third, *we can use a reconfigurable hardware platform much simpler, smaller, and cheaper than an emulator*, where inter-FPGA communication is a complex problem requiring expensive solutions, such as hierarchical crossbar structures [33], or dedicated field-programmable interconnect chips [23], or time-multiplexing hardware [6]. Unlike multi-FPGA partitioning techniques, which have to deal with a specific board interconnect scheme, *our decomposition is independent of the architecture of the target hardware platform.*

The lack of any inter-FPGA interconnect also provides the basis for the virtual hardware mechanism: if we have more subproblems (n) than available FPGAs (g), we simply *take turns in reusing every FPGA by dynamically reconfiguring it for different subproblems.* An FPGA that has finished a subproblem may be immediately reconfigured for the next one. The reconfiguration time is in most cases negligible compared with the computation time. In the worst case where all n subproblems must be solved, every FPGA will process, on the average, n/g problems. Thus the worst-case loss of efficiency caused by the time-multiplexing of FPGAs is proportional to n/g, which also represents the *overflow ratio* between the "size" of the problem and the available hardware capacity. Note that, in extreme, this scheme works even with a single FPGA, which must be reconfigured at most n times. For example, assume that we decompose a large SAT problem into 1,000 subproblems so that each one fits into one FPGA, but our reconfigurable hardware platform is limited to 50 FPGAs. So the problem size is significantly larger (20 times) than the available capacity. Nevertheless, we can still solve it by simply reconfiguring every FPGA for at most 20 subproblems. Since we may stop as soon as one FPGA finds a solution, it is likely that only an unsatisfiable original problem will require all 20 runs to complete.

Another benefit of the decomposition is the additional speedup obtained by the g FPGAs working concurrently on different subproblems. Since each subproblem is simpler than the original and is restricted to a smaller search space, and also because we can stop all computations as soon as one solution is found, the average speedup is usually greater than g (this observation is also supported by the results in [7]).

The drawback of this simple decomposition method is that in some CNFs the reductions in the problem size are relatively small, and we need to assign too many variables until we can make every subproblem fit into one FPGA, which in turn results in too many subproblems. This occurs in CNFs characterized by a small ratio between literals and variables, because setting one variable in such a formula results in a small reduction in the size of the problem. The next section provides a more general solution applicable to any CNF.

6.2. DECOMPOSITION FOR DISJOINT PARTITIONING

One type of partitioning used in software SAT solvers [27] is finding subsets of clauses that depend on disjoint subsets of variables. The resulting subproblems are independent and may be separately solved. If one subproblem is unsatisfiable, then so is the original problem. The solution to the original problem is obtained by concatenating the individual solutions of all subproblems. A disjoint partition provides an exponential reduction of the worst-case search space of a SAT solver, but it is seldom found in practical problems. However, such partitions may be created dynamically, as a consequence of the formula simplifications caused by assignments to variables.

The simple decomposition method described in the preceding section is inefficient when variable assignments result only in small reductions in the size of subproblems. In this section, we introduce a *new decomposition technique, whose goal is to assign variables so that the resulting subproblems are disjoint* [4]. This is motivated by the fact that disjoint partitioning significantly reduces the size of the resulting subproblems. The process of simplifying a formula based on an assignment $Var = val$ consists of removing those parts of the formula (variables, literals, and clauses) that become inactive, or *dead*, as a result of implying $Var = val$. The pseudocode of *Simplify* in Figure 10 outlines the simplification procedure. *To_imply* schedules assignments to be processed. If *val* conflicts with the current value of *Var*, the formula is recognized as unsatisfiable. All clauses satisfied by setting $Var = val$ are removed from the formula, along with all their literals. All remaining literals of *Var* and the variable itself are also removed. After this set of simplifications, we analyze all variables that lost some literals. A variable all of whose literals have died is also removed. If the remaining literals of a variable are all inverting or all noninverting, this dynamically unate variable may be safely assigned to the value that will satisfy all its clauses. Next we analyze all clauses that lost some literals (these are unsatisfied). If all the literals of a clause have been removed, this clause can no longer be satisfied, so the formula is recognized as unsatisfiable. If a clause is left with only one literal, its variable is assigned the value that satisfies the clause. We repeat this process until no more variables are assigned. If at this point all clauses are satisfied, the formula is recognized as satisfied.

```
Simplify (Var = val) {
To_imply (Var = val)
for every assignment Var = val to be implied {
  if value of Var is ¬val then return "UNSAT"
  if value of Var is val then continue
  assign Var = val
  for every clause cls satisfied by Var = val
    for every literal lit of cls Remove_literal (lit)
    Remove_clause (cls)
  for every literal lit of Var Remove_literal (lit)
  Remove_variable (Var)
  for every variable Var who lost literals {
    if Var has no literals then Remove_variable (Var)
    if Var has only non-inverting literals then To_imply (Var = 1)
    if Var has only inverting literals then To_imply (Var = 0) }
  for every clause cls who lost literals {
    if no literals left then return "UNSAT"
    if cls has only one literal lit of variable Var then
      if lit is inverting then To_imply (Var = 0)
      else To_imply (Var = 1) }
  } // no more assignments to imply
  if no clause left then return "SAT"
  return "SIMPLIFIED"
}
```

Figure 10. Formula simplification.

Our decomposition relies on a simple clause bipartitioning algorithm, which starts by creating from each clause a one-component cluster, then repeatedly merges clusters (based on a "closeness" measure reflecting the number of common variables), until two final clusters are obtained such that their number of common variables is minimized. To separate these two clusters, we assign their common variables. Note, however, that we may not need to assign all the common variables; as some of them may be implied or become dead after other common variables are assigned. Figure 11 outlines the algorithm to select the best variable to assign. It tries tentative formula simplifications for both 0 and 1 assignments to each common variable, and evaluates them based on the reduction obtained in the set of common variables (as a tie-breaker, we use the reduction in the number of literals). These simplifications are tentative since the changes they introduce may be erased by the procedure *Restore*. This analysis also does some learning; namely, when one assignment $Var = val$ falsifies the formula, then *Var* is set to $\overline{val}$ on a permanent basis (without *Restore*). The analysis is restarted after each instance of learning-induced simplification.

Figure 12 outlines the process of recursive decomposition trying to achieve a disjoint partitioning of the formula. MAX_SIZE is the capacity of the target FPGA and *Size*() provides the size of the current formula, both expressed as number of literals (after the library modules are defined, we can estimate how many literals

```
Select_best_var () {
RESTART: for every variable Var to be analyzed {
   lowest_penalty = 0
   status = Simplify (Var = 0)
   if (status = "SAT") then return "SAT"
   if (status = "UNSAT") then {// Var must be 1
     Restore()
     status = Simplify (Var = 1)
     if (status = "SAT") then return "SAT"
     if (status = "UNSAT") then return "UNSAT"
     go to RESTART }
   penalty = number of unassigned common variables
   Restore()
   if (penalty < lowest_penalty) then {
     lowest_penalty = penalty
     best_var = Var }
   status = Simplify (Var = 1)
   if (status = "SAT") then return "SAT"
   if (status = "UNSAT") then {// Var must be 0
     Restore()
     status = Simplify (Var = 0)
     if (status = "SAT") then return "SAT"
     if (status = "UNSAT") then return "UNSAT"
     go to RESTART}
   penalty = number of unassigned common variables
   Restore()
   if (penalty ≥ lowest_penalty) then continue
   lowest_penalty = penalty
   best_var = Var
} // all variables analyzed
return "DONE"
}
```

Figure 11. Variable selection.

will fit in the target FPGA [3]). The recursion terminates when the current formula fits into one FPGA. If the formula is too large, *Bipartition* clusters the clauses in two subformulas as disjoint as possible, and also computes the set *Common_vars*. If there are no common variables, the process continues recursively with the two disjoint subformulas. Note that here we need to satisfy both subformulas in order to satisfy the original one, so when one subformula is identified as unsatisfiable, the original one is also unsatisfiable.

When there are common variables, we select the best variable to assign using the procedure given in Figure 11, then recursively continue the decomposition with the two simplified formulas obtained by assigning the best variable. Here it is sufficient to satisfy one of the two subproblems, so when one subformula is satisfied, the original one is also satisfied.

Figure 13 shows the decomposition tree for *dubois*26 – one of the SAT benchmarks in the DIMACS set [14]; this formula has 78 variables, 208 clauses, and 624

```
Decompose_to_Partition (formula) {
if (Size() ≤ MAX_SIZE) then { // it fits
  ++nparts
  return "DONE"}
Bipartition (first_subformula, 2nd_subformula)
if (Common_vars = ∅) then { // disjoint subformulas
  status1 = Decompose_to_Partition (first_subformula)
  if (status 1 = "UNSAT") then return "UNSAT"
  status2 = Decompose_to_Partition (2nd_subformula)
  if (status2 = "UNSAT") then return "UNSAT"
  if (status1 = "SAT" and status2 = "SAT") then return "SAT"
  return "DONE"}
// we have common variables
status = Select_best_var()
if (status ≠ "DONE") then return status
Simplify (best_var = 0)
status0 = Decompose_to_Partition (formula)
if (status0 = "SAT") then return "SAT"
Restore()
Simplify (best_var = 1)
status1 = Decompose_to_Partition (formula)
if (status1 = "SAT") then return "SAT"
if (status0 = "UNSAT" and status1 = "UNSAT") then
     return "UNSAT"
return "DONE"
}
```

Figure 12. Decomposition for partitioning.

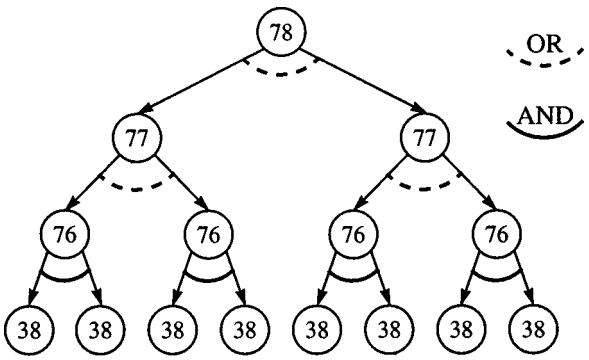

Figure 13. OR/AND decomposition tree for *dubois*26.

literals. The number inside a node is the number of variables for the corresponding subproblem. Setting MAX_SIZE = 300 literals, and assigning 2 variables, we obtain 8 subproblems, each one having 38 variables, 100 clauses, and 296 literals. (In general, the subproblems have different sizes, and the OR/AND graph is not symmetric.) The dotted arcs denote OR decomposition, where it is sufficient to solve only one subproblem, while the heavy arcs denote AND decomposition

Table III. Execution times for two subproblems.

A	B	OR	AND
SAT	SAT	$\min(a, b)$	$\max(a, b)$
SAT	UNSAT	a	b
UNSAT	SAT	b	a
UNSAT	UNSAT	$\max(a, b)$	$\min(a, b)$

obtained by disjoint partitioning, where both subproblems must be solved and their solutions concatenated. Thus *dubois*26 can be solved with 8 unconnected FPGAs. An additional advantage of the decomposition is the speedup resulting from the 8 FPGAs working concurrently solving subproblems much simpler than the original one (only 38 variables instead 78), and from the fact that we can stop execution as soon as the first pair of ANDed subproblems found a solution (or as soon as one FPGA from each pair identifies an unsatisfiable subproblem). If we have only 2 available FPGAs, we can solve the problem in at most four rounds of computation.

Table III summarizes the execution time for a pair of subproblems, A and B, obtained by decomposing a common parent problem. The execution time depends on the decomposition type (AND or OR), their results (SAT or UNSAT), and their execution times (denoted by a and b).

Although this is a more powerful decomposition technique, just like simple decomposition, when the target subproblem size is much smaller than the original problem size, the procedure may take an excessive time and/or produce too many subproblems. Decomposition for disjoint partitioning attempts to find the leaves in the decomposition tree that contain subproblems with the required size or smaller; in the worst case, we may need to explore an exponential number of branches before finding such leaves. However, increasing the subproblem size significantly speeds up the decomposition and reduces the number of subproblems to solve.

6.3. HARDWARE IMPLEMENTATION

The block diagram shown in Figure 14 illustrates a possible architecture for a PCI board to implement a satisfier using our virtual logic mechanism. The board is attached to a general-purpose host processor via a PCI bus. The host will use this board to solve computationally expensive SAT problems. The board features a microprocessor (μP), several FPGAs, and RAM for storing configurations.

The μP controls the virtual logic system, based on the OR/AND decomposition tree. It assigns subproblems to available FPGAs, starts their reconfiguration process, initiates the FPGA operation, and monitors the FPGA completion signals. Every FPGA communicates with the μP using four control signals: (1) *Start*; (2) *Done*; (3) *Satisfied*; and (4) *Stop*. When *Done* is asserted, *Satisfied* indicates

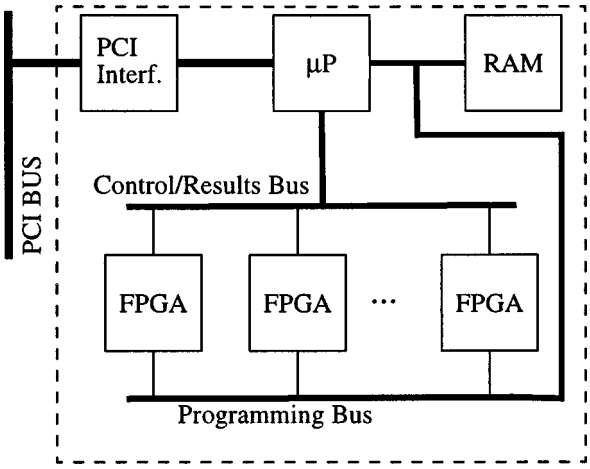

Figure 14. Satisfier board architecture.

whether the subformula assigned to that FPGA has been satisfied or it was proven to be unsatisfiable. *Stop* is used to abort a computation in progress in one FPGA when the μP has already determined that the result of that FPGA is no longer needed in computing the overall result. The four control signals of all FPGAs are bussed. Each FPGA has an address used by the μP to select that FPGA as the destination for *Start* or *Stop*, or to recognize the source of *Done* and *Satisfied*. The same *Control/ Results Bus* is used to retrieve the satisfying vectors computed by different FPGAs. The shared *Programming Bus* carries the configuration data from RAM to the selected FPGA. Since the computation does not require data path connections among FPGAs, this architecture is much simpler than an emulator.

7. Experimental Results

Figure 15 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 complex routing between the ST2 blocks of each variable and the OR2 blocks of each clause. It is this routing between variables and clauses 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. These new features are very important in reconfigurable computing applications. The block in the lower left corner is the Sync. Unit. As can be seen, it represents a small fraction of the whole area, and it does not vary with the SAT instance being solved. The layout took about 3 minutes to be compiled from its original CNF file. In contrast, the unstructured version of the same satisfier (that is, without using ILA-based design) could

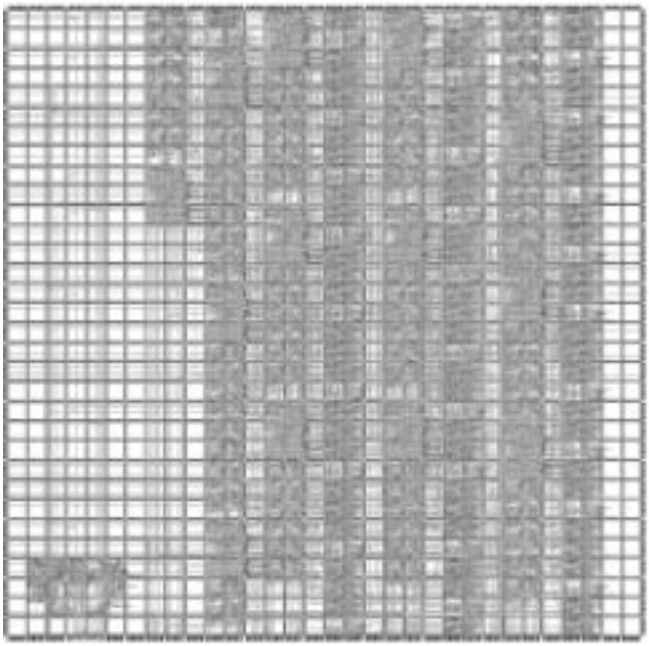

Figure 15. Satisfier layout on XC6264.

never be successfully compiled (after 10 hours of CPU time, there were still more than 200 unrouted nets). We determined that a satisfier that occupies the whole chip area can be clocked with a main clock frequency f of about 3.5 MHz. From the small size of this SAT instance we conclude that larger devices, virtual logic schemes, or multiple device architectures are in demand. All these are current research topics; the virtual logic scheme presented in this paper is a step toward these directions.

We ran satisfier circuits for instances that fit in the XC6264 in order to validate a software simulator of our architecture developed in the C language. With this program we were able to obtain the exact times that our satisfier circuits would take to solve larger instances. The simulation time is orders of magnitude slower, which limits the number of examples that can be analyzed in this way. Nevertheless, we managed to simulate enough examples to make the strong points that follow.

We use 25 examples extracted from the DIMACS set of SAT benchmarks [14]. In Table IV we compare the results of our satisfier with GRASP [28], one of the most efficient software SAT solvers available. We emphasize that these results are obtained without decomposition. The first 5 columns in Table IV 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 speedup SU gained with our approach. (For a meaningful comparison, we ignored instances for which runtimes were very

Table IV. Comparison with GRASP.

Benchmark	#I	#C	#L	Sat	Time[T]	Time$_G$[T]	SU
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
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

short.) The main clock frequency is 3.5 MHz. GRASP was run on a Sun Ultra SPARC workstation with 1026 Mb RAM using a 248 MHz clock. Thus GRASP is running with a clock about 83 times faster than the emulation clock. Unlike in [39], 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. These include clause addition based on conflict analysis, nonchronological backtracking, heuristics for decision making, etc. 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 (only *hole*10 was aborted by GRASP). For 10 examples out of the 15 included in Table IV, our satisfier achieved significant speedups between 78 and 7,000, and for 3 instances the speedup was in the 1.5 to 2.8 range. For 2 examples GRASP was faster than our satisfier. This is due to its sophisticated search features that do not have a match in our satisfier. However, if we also use decomposition, our satisfier will be faster in most cases. Nevertheless we envision our satisfier not as a replacement of, but as complement to, software SAT solvers.

Table V shows a comparison with the reconfigurable hardware satisfier described in [38]. To reproduce their results, we simply switched off our capabilities for identifying dead variables and implying dynamically unate variables, which are not implemented in [38]. As the variable order is the same, this would produce a fair comparison but for the mechanism for implication undoing. It is unclear in [38]

Table V. Comparison with [38].

Benchmark	#I	#C	#L	Sat	H	HO	Time[T]	SU
aim-50-1_6-no-3	50	80	240	N	12,854	—	51,900	80
aim-50-1_6-no-2	50	80	240	N	12,854	—	75,415	38
aim-100-1_6-yes1-3	100	160	480	Y	25,554	—	1,711K	>58
aim-100-1_6-no-2	100	160	480	N	25,554	—	6,570K	>15
aim-200-6_0-yes1-1	200	1200	3600	Y	165,354	1.6	942,830	1
hole8	72	297	648	N	32,113	—	259,519	9
hole9	90	415	900	N	44,099	—	2,336K	12
hole10	110	561	1210	N	58,751	2.6	23,357K	>4
ii8a2	180	800	2052	Y	98,166	2.5	60,587	71
ii32d2	404	5153	17940	Y	790,349	—	31,701	27
par16-1-c	317	1264	3670	Y	174,054	2.1	1,134K	1
ssa432-003	435	1027	2364	N	124,277	1.1	86,496	1

how implications are undone. It seems *all* implications are cleared, not only the ones resulting from the last decision, and all the previous decisions are re-implied when the last decision is modified. If this is the case, the comparison presented here is favorable to [38], since we undo only the implications of the last decision, and we do it in a single clock cycle. Column H represents the "hardware cost" of our satisfier, defined as in [38] as the total number gates and flip-flops. HO is the ratio of H to the corresponding cost in [38], 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 [38]. However, this buys us 1 to 2 orders of magnitude speedup, as illustrated by the last two columns in Table V. Again, these results did not take advantage of decomposition. Also, we are not including the speedup that results from our faster compilation time: according to [38], their compilation time for one FPGA is about 40 minutes, one order of magnitude greater than our 3 minutes.

Finally, we address one of the difficult problems in reconfigurable computing - size. In fact, none of the instances we analyzed can completely fit in a XC6264 device. In [38] an IKOS emulator was used, so that an instance can be mapped to multiple FPGA devices. This is an expensive solution given the price of emulators and the time required to partition the problem into multiple devices. In [38] it was reported that this step requires an additional compilation time in the order of tens of minutes. This severely restricts the range of instances where the approach can be useful compared to software SAT solvers.

In this work our goal was to find a cheaper and therefore more useful solution to this problem. The result was the decomposition techniques explained in Section 6. The maximum FPGA size MAX_SIZE was set to 300 literals, and we looked at examples that could be quickly (10 minutes at maximum) decomposed in at

Table VI. Results with decomposition.

Example	#I	#C	#L	Sat	#P$_d$	#P$_{dp}$
dubois50	150	400	1200	N	>1024	46
dubois100	300	800	2400	N	>1024	320
hole7	56	204	448	N	12	13
hole8	72	297	648	N	99	128
ii8a1	66	186	450	Y	2	3
par8-2-c	68	270	780	Y	1	10
par8-5-c	75	298	864	Y	0	1
pret60_25	60	160	480	N	>1024	64
pret60_40	60	160	480	N	>1024	64

most 1024 parts. The decomposition results are summarized in Table VI, where #P$_d$ is the number of parts (subproblems) obtained with the simple decomposition technique, and #P$_{dp}$ is the number of parts obtained with decomposition for disjoint partitioning, As we can see, in four examples the simple decomposition was aborted when the number of FPGAs passed the 1,024 limit, while decomposition for disjoint partitioning completed in all cases with a reasonably small number of parts. The problems where the simple decomposition obtained few parts are characterized by the presence of variables whose assignment contributes to significantly simplifying the formula without contributing toward achieving disjoint subproblems. In the future, we plan to combine the two methods to create an adaptive procedure that will dynamically switch between the two types of decomposition to take advantage of the benefits of each.

Another benefit of decomposition for disjoint partitioning is the fact that the resulting subproblems are much simpler and can run much faster than the original problem. Note that the number of variables in each disjoint part is significantly lower than the number of variables in the original problem. We observed that a subproblem can run 3 orders of magnitude faster than the original problem. Since there is a cost associated with decomposition, this acceleration can help compensate the cost incurred in decomposition. When the original problem is hard to solve, the speedup due to decomposition may decrease the overall computation time, even when the decomposition time is accounted for. However, further investigations are needed to determine which type of instance will benefit from this potential speedup.

8. New Directions

The satisfier architecture and the virtual logic scheme contributed by this research are just one of the few achievements in a promising and exciting new area: algorithms on reconfigurable hardware. In particular for SAT, many interesting aspects

are still to be explored. It is the objective of this section to discuss some of these new directions.

8.1. NONDISJOINT PARTITIONING

Both simple decomposition and decomposition for disjoint partitioning aim at producing independent subproblems that can be run by independent devices, concurrently or serially. However, the ability to do so depends on the *personality* of the instance. Some instances lend themselves better to these decomposition techniques than others. Not being able to handle instances whose personality is averse to these techniques represents a limitation in the usefulness of our approach.

To handle instances that resist decomposition by our techniques, we consider nondisjoint partition, that is, the decomposition in dependent subproblems. Suppose in decomposition for disjoint partition we obtain a residual set of common variables but, because we have exhausted our decomposition resources, we cannot proceed any further. One possible solution is to put these two parts in separate devices, and connect the devices with the necessary wires to carry the assignments to the common variables. Because we assumed a small number of common variables, we know only a few connections will be needed. This is not as general as full multi-FPGA partitioning, but is cheaper and enables handling more examples than with just our decomposition techniques. However, we would still have the disadvantages of communication across chips, longer delays, more complex hardware, and so on.

We are also looking into solving the dependent problems using the same platform. This would be accomplished in the following way: we would run one of the subproblems until we need to assign some of the common variables. Then we would swap the subproblems, while leaving on the platform the data concerning the common variables. The second subproblem would start running using these data, and generating its own, until more common variables are hit again. The first subproblem would be swapped in again and so on until the problem is solved. However, the overhead of swapping contexts may be excessive if the whole device need be reconfigured (current devices take tens of milliseconds for complete reconfiguration). One technology advance that can make this solution extremely attractive is the emergence of context switching FPGAs [32, 26]. Such devices can store different configurations or contexts in RAM, and switch between them in one clock cycle. In our case, switching between different SAT subproblems would become a trivial matter.

8.2. ALGORITHMIC IMPROVEMENTS

The algorithm implemented by our architecture executes a basic SAT search, incorporating only the minimal features of trying variable assignments in a fixed order, computing implications (Boolean constraint propagations), avoiding branching on dead (irrelevant) variables and implying unate variables (pure literal rule). The

architecture exploits the sheer speed of hardware and massive parallelism, rather than algorithmic sophistication. Nonetheless, if more algorithmic refinements can be introduced, while still maintaining the advantages of simplicity and massive parallelism, even more speed can be attained.

Whenever several subproblems must be concurrently solved, a general principle is to first solve the most difficult one, since solving easier problems first may be wasted effort. Of course, applying this principle depends on having good heuristic measures of "difficulty". In SAT, we can regard satisfying each clause as a subproblem, and consider the number of unassigned variables of a clause as an inverse measure of difficulty. One algorithmic improvement that we have been working on is a heuristic that favors decisions on variables feeding clauses having a small number of nondead literals. The objective mechanism is used to implement yet another priority level that signals these short clauses. In the simplest case we can signal only clauses containing two (nondead) literals. The clause would send, say, an objective LO+ instead of LO. This LO+ objective is accordingly merged with other objectives at the stem circuits, finally reaching the variables that will be given priority over the potential conflicts in the selection of the next decision variable. Such a heuristic is a variation of the well-known Jeroslaw–Wang heuristic [19], which is known to produce good results. This heuristic will require more hardware resources, of course, but will enhance the characteristics of massive parallelism.

A powerful algorithmic improvement introduced with GRASP [28] is the diagnosis of conflicting variable assignments. This analysis permits determining the variables that are responsible for the conflict, and enables (1) backtracking on the last assigned of these variables rather than on the last variable assigned (nonchronological backtracking), and (2) creating additional clauses from these variables, which will prevent the conflict from happening again (clause addition). In [38] a weak version of nonchronological backtracking in reconfigurable hardware is discussed. A considerable hardware overhead is needed for this scheme. A more plausible implementation of a complete conflict diagnosis procedure leading to both nonchronological backtracking and clause addition is put forward in [40]. Nonetheless, that scheme sacrifices massive parallelism and ignores dynamic variable selection. To be able to match the algorithmic sophistication of GRASP and add hardware acceleration and massive parallelism, we also need clever heuristics for dynamic variable selection.

8.3. MULTIDEVICE ARCHITECTURES

The basic problem of SAT on reconfigurable hardware is a space problem. Solving large SAT problems requires more hardware resources than can be found in any single FPGA device. Thus a multiple device architecture for SAT appears necessary. This is not a substitute for the virtual logic schemes we have studied; it is rather a complement. The virtual logic mechanism can use a multidevice structure instead of one chip to solve one subproblem. This means that it would be able to

handle larger parts, exponentially increasing the usefulness of our decomposition techniques.

The size problem also motivated the architecture described in [40]. This architecture uses multiple FPGA devices assembled in a pipelined ring. Each FPGA in this ring constitutes a processing element (PE) that implements a set of clauses. The variables circulate around the ring and get updated as they go along. One of the elements in the ring is a main control unit responsible for monitoring the global state of execution and selecting the next decision variable. This architecture has several advantages. Each PE is a shallow stage of logic, which can be clocked much faster (a frequency of 20 MHz is estimated). The need to route variables to clauses disappears, as variables become abstract entities. This contributes to structuring the layout, greatly improving the compilation time. Finally, and most important, the architecture is suitable for implementing dynamic addition of clauses derived by conflict diagnosis, using dynamic configurations. Despite these advantages, this architecture sacrifices the massive parallelism available in reconfigurable logic, and thus its scalability is questionable.

Our approach is to keep the massive parallelism of our architecture and to solve the size problem by extending the applicability of our decomposition technique. Instead of assigning a subproblem to one FPGA, we will assign it to a group of connected FPGAs. The virtual logic feature will result from the lack of connections among these groups. This will keep the cost lower compared with an emulation architecture. Our goal is a low-cost FPGA board that users can simply plug into their personal computers.

9. 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*. One result of objective propagation is the identification of all the "dead parts" of a CNF formula, which in turn allows the satisfier to recognize *dynamically unate variables* and *dead variable*s. These techniques generate more implications, avoid wrong and unnecessary decisions, and reduce the amount of back-tracking, thus providing significant speedups. Our results show several orders of magnitude speedup compared with both a state-of-the-art software SAT solver and a previous satisfier.

We have developed *modular design techniques using ILA structures.* These techniques result in *compilation time that grows only linearly with the size of the problem,* hence 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*.

We have introduced a *new virtual logic system for solving satisfiability problems using reconfigurable hardware.* We combine a simple decomposition method with a formula partition technique to create a *decomposition for disjoint partition.* This novel technique can effectively divide a formula into independent subproblems that can be concurrently run by multiple FPGAs.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 a much faster algorithm, since it does not need to know how the FPGAs are interconnected in our platform. In fact, it is independent of the reconfigurable hardware platform, so it may be easily used on any emulator without using its inter-FPGA communication structures. But since it does not require any inter-FPGA routing, it can work on a platform much simpler and cheaper than an emulator. Also the clock frequency may be higher, because we avoid the delays involved in propagating signals off-chip, on the board, and back on-chip.

The decomposition algorithm is such that often the resulting subproblems can be serially run much faster than the original problem. This is a consequence of the sophisticated dynamic variable selection used by the algorithm. It is also the key to our *virtual logic mechanism*, which allows a reconfigurable logic platform to process circuits much larger than its available capacity. Although the sum of the size of the subproblems is greater than the size of the original problem, the summation of the execution times is often smaller than the execution of the original problem. The FPGAs can be run concurrently, introducing a new level of coarse-grain parallelism, and resulting in up to three orders of magnitude additional speedup. This comes from the disjoint partitioning of the search space. Our architecture is easily scalable, since all we need to handle larger problems is additional or larger FPGAs. Combined with our fine-grain massively parallel satisfier architecture, the virtual logic for SAT has the potential to offer a low-cost solution for solving computationally difficult problems in a wide variety of fields.

Acknowledgment

We acknowledge useful discussions with Prof. D. Saab on early versions of these ideas.

References

1. Abramovici, M., Breuer, M. A. and Friedman, A. D.: *Digital Systems Testing and Testable Design*, IEEE Press, 1995.
2. Abramovici, M. and Saab, D.: Satisfiability on reconfigurable hardware, in *Proc. Intn'l. Workshop on Field-Programmable Logic and Applications*, Sept. 1997.

3. Abramovici, M., Sousa, J. and Saab, D.: A massively-parallel easily-scalable satisfiability solver using reconfigurable hardware, in *Proc. Design Automation Conf.*, June 1999.
4. Abramovici, M. and de Sousa, J. T.: A virtual logic system for solving satisfiability problems using reconfigurable hardware, in *Proc. Symp. on Field-Programmable Custom Computing Machines*, April 1999.
5. Babb, J., Frank, M. and Agarwal, A.: Solving graph problems with dynamic computation structures, in *SPIE Photonics East: Reconfigurable Technology for Rapid Product Development & Computing*, November 1996, pp. 225–236.
6. Babb, J., Tessier, R. and Agarwal, A.: Virtual wires: Overcoming pin limitations in FPGA-based logic emulators, in *Proc. Symp. on Field-Programmable Custom Computing Machines*, April 1993, pp. 142–151.
7. Bohm, M. and Speckenmeyer, E.: A fast parallel SAT solver: Efficient workload balancing, *Ann. Math. Artificial Intelligence* **17**(3–4) (1996), 381–400.
8. Brayton, R., Hachtel, G., McMullen, C. and Sangiovanni-Vincentelli, A.: *Logic Minimization Algorithms for VLSI Synthesis*, Kluwer Academic Publishers, 1984.
9. Chakradhar, S. T. and Agrawal, V. D.: A novel VLSI solution to a difficult graph problem, in *Proc. Intn.l. Symp. on VLSI Design*, January 1990, pp. 124–129.
10. Cook, S. A.: The complexity of theorem-proving procedures, in *Proc. 3rd Annual ACM Symp. on Theory of Computation*, 1971, pp. 151–158.
11. Davis, M. and Putnam, H.: A computing procedure for quantification theory, *J. ACM* **7** (1960), 167–187.
12. Devadas, S.: Optimal layout via boolean satisfiability, in *Proc. Intn'l. Conf. on CAD*, November 1989, pp. 294–297.
13. Devadas, S., Keutzer, K., Malik, S. and Wang, A.: Certified timing verification and the transition delay of a logic circuit, in *Proc. Design Automation Conf*., June 1992, pp. 549–555.
14. DIMACS Challenge Benchmarks, *ftp://dimacs.rutgers.edu/pub/challenge/sat/benchmarks/cnf/*.
15. Fujiwara, H. and Shimono, T.: On the acceleration of test generation algorithms, *IEEE Trans. Comput.* **C-32**(12), December 1983, pp. 1137–1144.
16. Gately, J. et al.: UltraSPARC-I emulation, in *Proc. Design Automation Conf*. June 1995, pp. 13–18.
17. Goel, P.: An implicit enumeration algorithm to generate tests for combinational logic circuits, *IEEE Trans. Comput.* **C-30**(3), March 1981, pp. 215–222.
18. Gu, J.: Satisfiability problems in VLSI engineering, in *DIMACS Workshop on Satisfiability Problem: Theory and Applications*, March 1996.
19. Gu, J., Purdom, P. W., Franco, J. and Wah, B. W.: Algorithms for the satisfiability (SAT) problem: A survey, in *DIMACS Workshop on Satisfiability Problem: Theory and Applications*, March 1996, pp. 19–51.
20. Gu, J. and Puri, R.: Asynchronous circuit synthesis with boolean satisfiability, *IEEE Trans. on CAD* **14**(8) (1995), pp. 961–973.
21. Larrabee, T.: Test pattern generation using boolean satisfiability, *IEEE Trans. on CAD* **11**(1) (1992), pp. 4–15.
22. McGeer, P. C. et al.: Timing analysis and delay-fault test generation using path recursive functions, in *Proc. Intn'l. Conf. on CAD*, November 1991, pp. 180–183.
23. *http://www.icube.com/prodpage/ProdPage.html*, I-Cube Co.
24. Platzner, M. and De Micheli, G.: Acceleration of satisfiability algorithms by reconfigurable hardware, in *Proc. Intn'l. Workshop on Field-Programmable Logic and Applications*, Sept. 1998.
25. Rashid, A., Leonard, J. and Mangione-Smith, W. H.: Dynamic circuit generation for solving specific problem instances of boolean satisfiability, in *Proc. Symp. on Field-Programmable Custom Computing Machines*, April 1998, pp. 196–204.

26. Scalera, S. M. and Vazquez, J. R.: The design and implementation of a context switching FPGA, in *Proc. Symp. on Field-Programmable Custom Computing Machines*, April 1998, pp. 78–85.
27. Silva, J. M.: An overview of backtrack search satisfiability algorithms, in *Proc. 5th Intn'l. Symp. on Artificial Intelligence and Mathematics*, January 1998.
28. Silva, J. M. and Sakallah, K. A.: GRASP – A new search algorithm for satisfiability, in *Proc. Intn'l. Conf. on CAD*, November 1996, pp. 220–227.
29. L. G. Silva et al.: Realistic delay modeling in satisfiability-based timing analysis, in *Proc. Intn'l. Symp. on Circuits and Systems (ISCAS)*, May 1998.
30. Stephan, P. R., Brayton, R. K. and Sangiovanni-Vincentelli, A.: Combinational test generation using satisfiability, *IEEE Trans. on CAD* **15**(9) (1996), 1167–1176.
31. Suyama, T., Yokoo, M. and Sawada, H.: Solving satisfiability problems on FPGAs, in *Proc. Intn'l. Workshop on Field-Programmable Logic and Applications*, 1996.
32. Trimberger, S., Carberry, D., Johnson, A. and Wong, J.: A time-multiplexed FPGA, in *Proc. Symp. on Field-Programmable Custom Computing Machines*, April 1997.
33. Varghese, J., Butts, M. and Batchller, J.: An efficient logic emulation system, *IEEE Trans. on VLSI* **1**(2) (1993), 171–174.
34. Vuillemin, J. et al.: Programmable active memories: reconfigurable systems come of age, *IEEE Trans. on VLSI Systems*, March 1996.
35. Wood, R. G. and Rutenbar, R. A.: FPGA routing and routability estimation via Boolean satisfiability, in *Proc. Intn'l. Symp. on FPGAs*, February 1997.
36. Wu, Y. and Adham, S.: BIST fault diagnosis in scan-based VLSI environments, in *Proc. Intn'l. Test Conf.*, October 1996, pp. 48–57.
37. *XC6200 Field-Programmable Gate Arrays*, Xilinx, June 1996.
38. Zhong, P., Martonosi, M., Ashar, P. and Malik, S.: Accelerating Boolean satisfiability with configurable hardware, in *Proc. Symp. on Field-Programmable Custom Computing Machines*, April 1998.
39. Zhong, P., Martonosi, M., Ashar, P. and Malik, S.: Using reconfigurable computing techniques to accelerate problems in the CAD domain: A case study with Boolean satisfiability, in *Proc. Design Automation Conf.*, June 1998.
40. Zhong, P., Martonosi, M., Ashar, P. and Malik, S.: Solving Boolean satisfiability with dynamic hardware configurations, in *Proc. Intn'l. Workshop on Field-Programmable Logic and Applications*, Sept. 1998.