A Configware/Software Approach to SAT Solving

J.T. de Sousa, J.M. da Silva

INESC-Lisboa/IST
Technical University of Lisbon
R. Alves Redol, 1000 Lisboa, Portugal
{jose.desousa,jpms}@inesc.pt

M. Abramovici

Circuits and Systems Research Lab
Agere Systems
Murray Hill, NJ 07974, USA
miron@research.bell-labs.com

Abstract

This paper presents the first configware/software solution for the Boolean satisfiability problem (SAT). Our partitioning assigns the two most difficult tasks (computing implications and selecting the next decision variable) for configurable hardware (configware) implementation. Conflict diagnosis, backtrack control, and clause database management are performed in software. Our approach avoids instance-specific place and route, and features a virtual hardware scheme with context switching which allows solving problems that are larger than the configware capacity. Some preliminary results show the feasibility of the approach.

1 Introduction

1.1 Definitions and Motivation

The satisfiability (SAT) problem — given a Boolean formula $F(x_1, x_2, \ ..., \ x_n)$, 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 [1]. Typically F is expressed as a product-of-sums, which is also called conjunctive normal form (CNF). The terminology is reviewed via an example: in the formula $F = (x_1 + x_2)(\bar{x_1} + x_2)(x_1 + \bar{x_2})$, we have two variables (x_1 and x_2) and three clauses, each with two literals; the literals in the third clause are x_1 and $\bar{x_2}$, where $\bar{x_2}$ is an inverting literal and x_1 is a non-inverting one. The assignment $(x_1 = 1, \ x_2 = 1)$ is a satisfying assignment, as it sets $F = 1$. Hence F is satisfiable. The formula $F' = F(\bar{x_1} + \bar{x_2})$ is unsatisfiable. The number of variables of a formula is denoted n and the number of clauses m. A k-clause is a clause that has k literals. A k-SAT problem is one where clauses have at most k literals.

SAT has many applications in CAD and test [2]; 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 and test problems that require solving large systems of Boolean equations include timing verification [3, 4], layout, and routability analysis [5, 6]. SAT is also related to the automatic test pattern generation (ATPG) problem, as the entire ATPG problem may be formulated as a SAT problem [7, 8]. Computing the maximum circuit delay in the presence of false paths can also be solved by SAT [9]. Many problems in logic synthesis [10] — such as state assignment, state minimization, I/O encoding and asynchronous circuit design [11] — have SAT-based solutions.

It is well-known that SAT is an NP-complete problem [12]. Even with the most advanced software SAT algorithms, such as GRASP [13], 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.

1.2 Previous Work

Several research groups have recently explored different approaches to implement SAT on configurable hardware (configware) [15, 16, 17, 18, 19, 20, 24], as an alternative to software SAT solvers. The main idea is to map a formula F into an instance-specific circuit C that solves F. Since C will be used only once, it is not economically feasible to actually fabricate it. Using configware allows one to *virtually* create C, then execute the algorithm by emulating this circuit. In this approach circuit C is designed specifically for a single formula. The advantage is that C will run at emulation speed, without incurring the cost of building special-purpose hardware.

The satisfiers implement variations of the classical Davis-Putnam SAT algorithm [21]. For example, the satisfier of Zhong et al. [18] implements the following algorithm. For every variable v_i, they construct an implication circuit and a state machine to manage the decision process for v_i. The implication circuit detects the conditions when values of other variables imply a value for v_i, and detects a conflict when both 0 and 1 are implied for v_i. The decision state machine keeps track of the current value of v_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 left most enabled cell whose variable v_i has an unknown value first assigns $v_i = 1$. All the implications of this assignment are determined, and if no conflicts are detected, the next cell to the right is enabled. If $v_i = 1$ leads to conflicts, the assignment $v_i = 0$ is tried next. When both 0 and 1 lead to conflicts, v_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 in [18] implements a sophisticated non-chronological backtracking mechanism that allows further performance improvement.

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 [13]. Dynamic selection, however, has been considered difficult to directly implement in hardware. What all hardware SAT solvers do is order the variables statically (as a pre-processing step) based on some heuristic functions. This pre-determines 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.

Another important issue affecting any algorithm implemented on configware, 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, Zhong et al. [18] 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 of computation. Abramovici et al. [20] propose structured layout techniques that can reduce the compilation effort to a few minutes, enlarging the scope of problems for which the approach is competitive. To completely avoid instance-specific layout compilation, Boyd and Larrabee [24] have proposed an architecture of a SAT-specific programmable logic device (PLD). This device can be configured in polynomial time to solve any k-SAT problem. However, its scalability and area efficiency depend heavily on the distribution of the number of literals per clause, which varies a lot among problem instances.

All the approaches described assume that the problem fits the available configware. Abramovici and de Sousa were the first to propose a virtual hardware scheme [25] that allows solving problems that exceed the configware capacity. They used formula decomposition techniques to partition a formula in several independent sub-formulas which can fit the available hardware. These subproblems are successively run by the hardware using a simple procedure to control the execution of the subproblems, and to monitor whether the original formula has been solved or proven unsatisfiable. The main limitation of this approach is that the efficiency of the decomposition techniques depends on the nature of the formula, and decomposing some formulas may increase the compilation time to unacceptable levels.

1.3 Main Contributions

Although the existing configware approaches to SAT solving can speed up the execution of some problems, in many other instances (especially the CAD ones) they are not successful. One reason for this is that instances derived from real world problems usually have a structure that the sophisticated conflict diagnosis and clause recording techniques of software solvers can recognize and exploit. Despite their massive parallelism, the basic search algorithm of configware solvers limits their ability to intelligently explore the search space. The control techniques required by sophisticated diagnosis engines would take a lot of space if implemented in hardware, and would hardly give any contribution to acceleration, because control is inherently sequential, and has little or no parallelism to exploit.

Therefore, we introduce a **configware/software**

approach. Conflict diagnosis and derivation of new clauses are done in software, whereas the *massively parallel computation of logical implications and decision variable selection heuristics are performed by the configware*. This is also the first reported **configware implementation of a decision variable selection heuristic**. *Communication between configware and software is minimized*: the host receives from, and sends to, the configware only the variables that have changed, and updates the configware clauses by adding or removing clauses from time to time. We also propose a **virtual hardware scheme** that allows processing instances that exceed the configware capacity. This is accomplished by dividing the original circuit in hardware pages which are successively run, with intermediate results stored in external RAMs. To speed-up page swapping we use a **context switching** PLD: the next hardware page is accessed in one clock cycle. This is the first virtual hardware approach for SAT solving where *the partition in hardware pages runs in (almost linear) polynomial time*, independent of the formula being compiled.

We also **convert the original k-SAT problem into a 3-SAT problem** as a preprocess step. This can be done in linear time and solves the scalability and area efficiency problems of [24] with an acceptable are overhead, since the problems of interest (CAD problems are good examples) consist of mainly 2 and 3-clauses anyway.

The remainder of the paper is organized as follows. Section 2 overviews the SAT algorithm, and discusses the configware/software partioning used. Section 3 describes the detailed architecture of the system. Section 4 presents our preliminary results. Section 5 outlines our conclusions and describes our current and future work.

2 Algorithm Overview

Figure 1 shows the overall structure of a generic SAT algorithm, capturing the organization of several of the most competitive algorithms [13, 26, 27]. The algorithm conducts a search through the space of possible variable assignments, and has three main engines: the **Decision, Deduction** and **Diagnosis** engine. The *decision* engine, grades potential decision variables using a selection heuristic, and one is elected to be assigned a tentative value. Its value, 0 or 1, is also chosen by the same heuristic. The *deduction* engine, identifies implied necessary assignments resulting from the last decision variable using the unit clause rule — whenever an unresolved clause has a single unassigned literal, the corresponding variable is

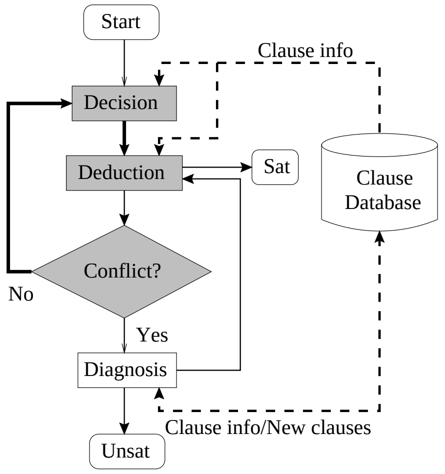

Figure 1: The satisfier algorithm. The shaded procedures are implemented in configware.

implied to the value that satisfies the clause. When one or more clauses are falsified, the deduction engine reports the existence of conflicts, which are then analyzed by the *diagnosis* engine. Diagnosis of a given conflict returns a set of variable assignments responsible for the conflict. The most recently assigned decision variable responsible for the conflict is complemented, and the implications of this and all following decisions are deleted. The implications of the new assignment are then computed, and the search proceeds in this way until a variable vector that satisfies the formula is found, or the formula is shown to be unsatisfiable.

The analysis of conflicts can also be used for identifying and recording new implicates of the Boolean function associated with the CNF formula. Clause Recording plays a key role in recent SAT algorithms [13, 26, 27]. For example, let us consider the formula

$$\begin{aligned} F = & (x_1 + x_4)(x_2 + x_4)(\bar{x_1} + \bar{x_2} + \bar{x_4}) \\ & (\bar{x_2} + \bar{x_5})(\bar{x_3} + \bar{x_5})(x_2 + x_3 + x_5) \\ & (\bar{x_4} + \bar{x_6})(\bar{x_5} + \bar{x_6})(x_4 + x_5 + x_6) \end{aligned}$$

where $x_1 = 1$, $x_6 = 0$ and x_2 is assigned value 1. Clearly, this assignment yields a conflict, since both x_4 and x_5 are implied to value 0, and these assignments are inconsistent with the last clause . This conflict will hold as long as the assignments $x_1 = 1$, $x_2 = 1$ and $x_6 = 0$ hold. In order to prevent this conflict at least one of the assignments must be complemented. As a result, the clause $(\bar{x_1} + \bar{x_2} + x_6)$ can be derived.

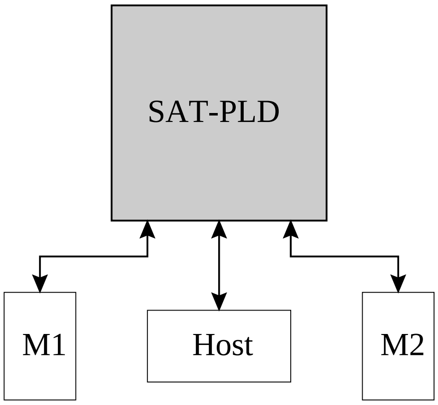

Figure 2: High-level view of the system.

Configware/Software Partitioning

After profiling GRASP [13], we realized that the decision and deduction engines take between 60 to 95% of the computation time in a set of benchmarks derived from hard problems. According to Amdahl's law, *the potential for acceleration ranges from 2.5 to 20X*. For the moment, our configware/software partitioning assigns the deduction and decision engines to configware while keeping the entire diagnosis engine in software. The data processing part (reverting implications) of the diagnosis engine could also be implemented in configware, further speeding the system. This issue is left for the future.

3 System Architecture

3.1 High-Level View

The top-level view of the system is shown in Figure 2. The system consists of a host computer connected to a reconfigurable platform containing a PLD and two single-port RAMs M1 and M2 (a single dual-port RAM can also be used). The RAMs contain the data associated with variables. A subset of the configurations (hardware pages) are stored in the deviced itself and are swapped in one clock cycle. A new page is preloaded while the current one completes. The device may be implemented with a multi-context FPGA, but, since our approach does not require instance specific place and route, a configurable device specifically designed for this problem would be more advantageous. Figure 3 shows the architecture of the PLD. The main block is the **clause pipeline**. This block processes the variables entering the input on the left and outputs the processed variables to the right. Data can be read from either M1 or M2, depending on state of the *connect* switch, which is controlled by a simple

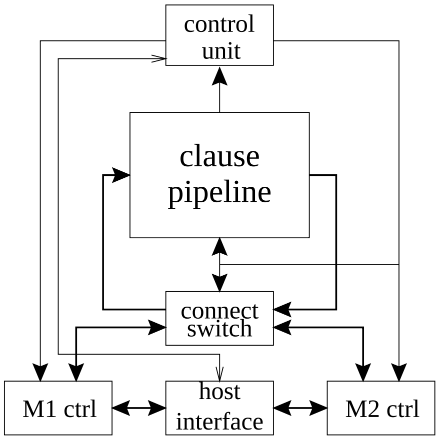

Figure 3: Inside the SAT-PLD.

control unit. The control unit responds to events issued by the clause pipeline and the *host interface* and manages data transfers.

Suppose the variables are currently in M1, and the PLD is configured with the formula corresponding to one page. The variables are retrieved from M1, streamed through the PLD, and stored in M2. The PLD grades incoming variables with respect to their usefulness for being the next decision variable, and also performs all implications. Then the PLD switches to the next hardware page, and the process continues with the data now moving from M2 back to M1. Running all hardware pages through the PLD is called a *pass*. If new implications are generated during a pass, another one is started. When no more implications are generated in a pass, we say that *stasis* has been reached. If after stasis is reached some conflicts are detected, the host reads the relevant variables from the configware and analyzes the conflict. Conflict analysis may produce new implicates (clauses) which are dynamically added to the clause pipeline to avoid those conflicts in the future. The conflict analysis also determines the variable assignment causing the conflict, and complements the previous decision. If no conflicts are present, the graded variables are passed to the host, who decides the variable to be next assigned.

The host is also in charge of verifying the state of the formula: unresolved, satisfied or unsatisfiable. The formula is satisfied when all clauses have been satisfied. The formula is unsatisfiable when the diagnosis engine returns no backtrack variables. In either case, satisfied or unsatisfiable, execution is complete. If the instance is unresolved and there are no conflicts, the host chooses a new decision variable based on the grading done in configware, and assigns it. After the new variable is assigned, a new implication/decision

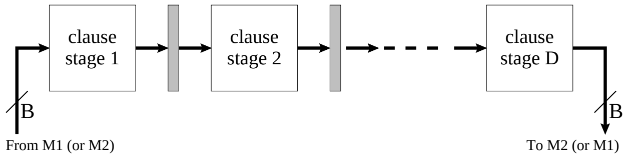

Figure 4: The clause pipeline.

pass is started, and the entire process described above is repeated.

3.2 Clause Pipeline

Figure 4 shows the clause pipeline, whose structure is inspired by the ring architecture presented in [19]. Like in that approach, the variables are the data circulating through the pipe, and are updated when they meet the clauses they feed in the various pipeline stages. An important difference is that in our approach, the output of last stage D does not feedback to the input of stage 1; stage 1 reads the variables from one of the memories, whereas stage D outputs the processed variables to the other memory. This enables a virtual hardware scheme, since the pipe can be reconfigured with the next hardware page while the data is safely kept by the memories. The paging scheme also allows an unlimited number of clauses to be dynamically added to the configware, without need to have spare hardware for that purpose as in [19] However, if the whole formula fits in the PLD, there is no need to use the external RAMs. The output of the pipe can be fed back to its input forming a ring structure similar to the one proposed in [19].

Every clause stage keeps the data for a subset of M clauses, and has a mechanism to select the variables feeding its clauses from the input stream. Because the bandwidth of the pipeline is B bits, only a slice of N variables can be processed in parallel by a stage. The relation between N and B depends on the number of bits k needed to encode a variable. Thus $B = kN + c$, where c bits are reserved for control. For example, if $B = 64$ bits and we encode each variable with $k = 6$ bits, then a stage can process $N = 10$ variables, and 4 bits are left for control. Figure 5 illustrates the layout of a slice of variables. The 4 bits in the control field have the following meaning:

I: the *implication* bit indicates that some variable in the slice has been implied.

D: the *decision* bit indicates that some variable in the slice has been graded as a potential decision variable.

C: the *conflict* bit indicates that some assigned variable in the slice has caused a conflict, i.e., has falsified a clause.

Z: the *local implication* bit indicates that some variable in the slice has been implied in the last round of implications [18].

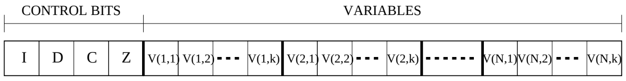

Figure 5: Structure of a slice of variables.

The set of slices form a train of variables which has length $L = \lceil n/N \rceil$. The control unit monitors the Z bit in the last stage to check for stasis. If the L slices of variables pass and no Z bits are asserted, then stasis has been reached. After stasis is reached there may or may not be conflicts; this is indicated by the C bit being set in at least one slice. If there is a conflict, only the slices where the I bit is asserted are passed to the host to update the state of its variables and initiate diagnosis. The host finds the variables responsible for conflicts by checking their encoding in the slices with C=1 (the encoding will be explained in the next subsection). The variables responsible for conflicts are used to determine the conflicting clauses, which are then sent to the diagnosis engine.

If there are no conflicts (C=0 for all slices), the variables of the slices where the I or D bits are asserted are passed to the host. Based on their encoding, the host finds the potential decision variables, and selects the next one to be assigned (the details will be provided in the next subsection).

Our clause design was greatly influenced by our previous architecture [20]. In fact, the previous design is almost equivalent to a one-stage pipeline in the current design. The pipeline structure is introduced in the new design to solve the problems of scalability, virtual hardware, and dynamic reconfiguration (adding and removing clauses on the fly).

3.2.1 Variable Encoding

The k bits used for variable encoding are divided in two equal fields, called the *0-priority level* l_0 and the *1-priority level* l_1. The priority level l_b of a variable is a heuristic measure that reflects how useful is to set the variable to value b for satisfying the formula. The highest priority level (all $k/2$ bits set to 1) means that the variable has already been *assigned* by a decision or an implication, and this level is denoted by A. A variable for which $l_0 = l_1 = A$ is responsible for a conflict since it has been implied to both Boolean values. The l_b levels between 0 and $A - 1$ indicate increasing degrees of eligibility of b as a decision value for the variable.

Initially we set $l_0 = l_1 = 0$ for all variables. The heuristic we use is to increment l_b whenever a variable passes by one of its unresolved clauses that is satisfied for value b. The incrementing stops when the level reaches $A-1$. All levels less than A are reset before each new pass of the formula (otherwise they would accumulate from pass to pass).

This encoding represents a generalization of the priorities used in [20]. The A level corresponds to the high priority HI. A variable that has $l_0 = l_1 = 0$ after passing all pipeline stages corresponds to a *dead* variable, whose literals feed only already satisfied clauses. The host should never assign a dead variable. A variable that has $0 < l_b < A$ and $l_{\bar{b}} = 0$ after passing all pipeline stages corresponds to a *dynamically unate* variable, whose literals in currently unresolved clauses have the same polarity. The host should set a dynamically unate variable to value b by an implication, since this assignment satisfies the clauses fed by the variable, without causing any conflicts.

Now we can explain how the host selects the next decision variable among the variables received from the clause pipeline. These are the variables of the slices where the I or D bits are asserted. Any variables which have not been implied ($l_0, l_1 < A$) are potential decision variables. There may be various heuristics for doing this. For example, we could choose the variable with the highest $l_0 + l_1$ score, and assign it the value corresponding to $\max\{l_0, l_1\}$. Ties may be resolved using a static order of the variables computed as a preprocess step. Another technique that we use is to introduce a little randomization in the process. Instead of selecting the variable with the best score we randomly select one variable among the top 10% variables, for instance. A certain degree of randomization has been reported an effective technique for SAT solving [2].

3.2.2 Clause Stage

Since any k-SAT formula ($k > 3$) can be converted to a 3-SAT formula, we use this property to create a regular structure where only 3-clauses are used. This conversion can be accomplished in linear time, allowing us to process any CNF formula.

A clause pipeline stage is detailed in Figure 6. The stage contains two types of blocks: the clause processor block (CL) and the variable merge block (VM).

Each stage contains M CL blocks. A CL block implements a 1, 2 or 3-clause. A CL block selects its 3 inputs out of the N incoming variables. Note that each k-width signal carries a series of time multiplexed variables. Thus a clause may select the same signal for its 3 inputs provided that the 3 input variables are in dif-

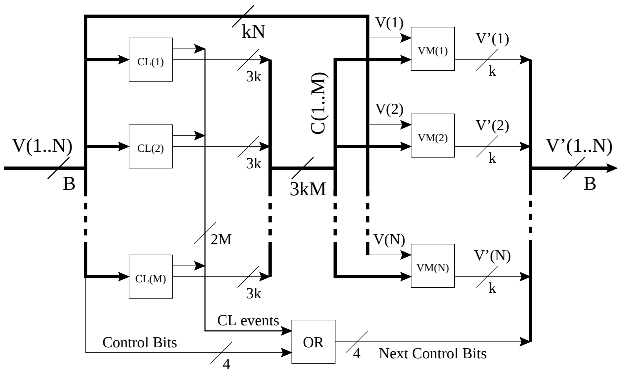

Figure 6: A clause pipeline stage.

ferent time slots. A 3-clause receives 3 variables and outputs the same 3 variables after processing them. Thus, the clauses' outputs form a bus C(1..M) of M signals, each of which has width $3k$. The control bits of a slice of variables are updated at each stage by the CL blocks. The next control bits simply result from ORing the incoming EVENT signals generated by the CL blocks with the control bits from the previous stage. The operation of the CL block is described in Section 3.2.3.

After being processed by a clause CL(i), a variable V(j) becomes one of the 3 vars of signal C(i). In order to be passed to the next stage, the original order of variables must be restored. This is the function of the VM blocks, which are described in detail in Section 3.2.4.

3.2.3 Clause Processor

The detailed implementation of a clause processor (block CL) is shown in Figure 7. The three N-input MUXs select the 3 input variables for the clause. The input MUXs are controlled by a ($\lceil \log_2(N+1) \rceil$)-bit configuration register. The MUXs before the **Clause Processor** select whether the variable from the input MUX or the variable stored in `Var Reg` should be used by the processor. If the variable is passing by the stage at that exact moment, then the variable from the input MUX should be used; otherwise, the stored variable is used. This MUX is controlled by a $\lceil \log_2 L \rceil + 1$-bit configuration register that contains the time slot `Vtime` of when the variable passes by, and the literal inversion `sign` in the last bit. For non inverting literals `sign`=0, and for inverting ones `sign`=1. We need $3M[\log(N+1) + \log_2(L+1)] \approx 3M \log_2 n$ configuration bits for each clause stage.

To implement a 2-clause, the `Var Reg` is preset with a zero-variable ($l_0 = A,\ l_1 = 0$) which is always selected by the MUX. For a 1-clause, we preset two in-

INP1		INP2 and INP3	OUT1		EVENT	COMMENT
$l'_0 = A$	$l'_1 < A$	both have $l'_0 = A$	$l'_0 = A$	$l'_1 = A$	C=1, I=1	Conflict
$l'_0 < A$	$l'_1 < A$	both have $l'_0 = A$	$l'_0 = l_0$	$l'_1 = A$	I=1	Implication
$l'_0 < A$	$l'_1 < A-1$	one has $l'_0, l'_1 < A$; other has $l'_1 < A$	$l'_0 = l'_0$	$l'_1 = l'_1 + 1$	D=1	Decision
All other situations			INP1		No events	

Table 1: Clause processor operation

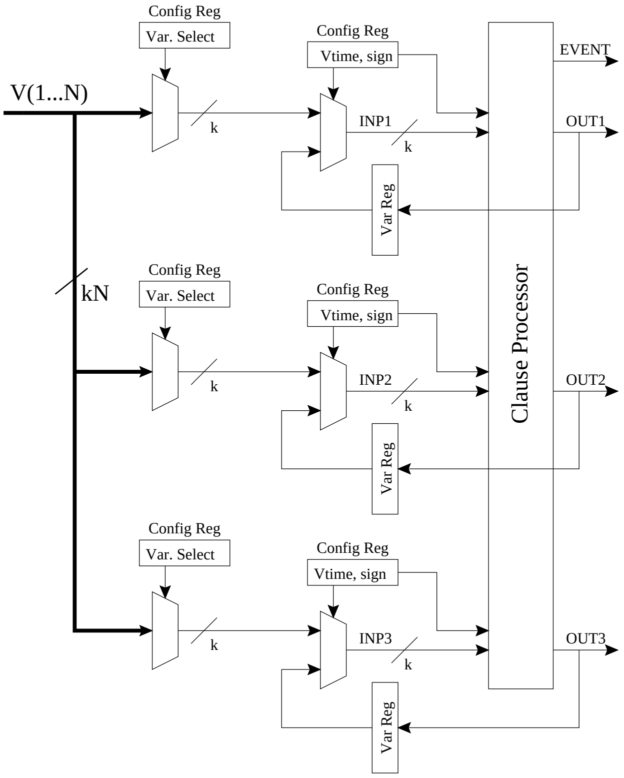

Figure 7: The clause processor (CL).

puts with the zero-variable.

The operation of the clause is described in Table 1. This table gives the output OUT1 as a function of the respective input INP1 and the two other inputs INP2 and INP3. It also shows the events that are generated by the clause. OUT2 and OUT3 are given by similar tables as a function of INP2 and INP3, respectively. Instead of the levels l_0 and l_1, Table 1 employs $l'_0 = l_{0 \oplus sign}$ and $l'_1 = l_{1 \oplus sign}$. That is, l'_1 refers to the literal that satisfies the clause, and l'_0 refers to the literal that does not satisfy the clause. When both l'_0 and l'_1 equal A, this means that the variable has caused a conflict at some clause.

The first row of Table 1 shows how conflicts are detected. A variable assigned to 0 traverses a clause

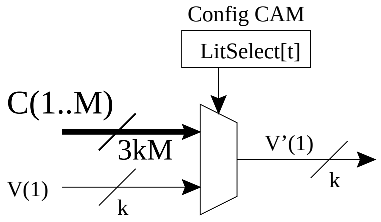

Figure 8: The variable merge block (VM).

where the other two inputs have value 0. The fact that the 3 inputs have $l'_0 = A$ falsifies the clause and causes a conflict event (C=1) to be generated. It also causes l'_1 to be implied to $l'_1 = 1$, so that in this way we can know that the variable has caused a conflict. The second row of the table illustrates a simple implication. INP1 has $l'_0 < A, l'_1 < A$ and INP2 and INP3 have $l'_0 = A$. In this case, OUT1 is assigned to 1 ($l'_1 = A$), since this is the only way to satisfy the clause; an implication event (I=1) is generated by the clause. The third row of the table shows a possible heuristic to increment the priority levels l_0 and l_1 (depends on which one is represented by l'_1). This heuristic increments l'_1 each time a variable passes by an unresolved clause; a decision event (D=1) is generated when this happens. Finally, the fourth row of Table 1 states that in all other cases OUT1=INP1 and no events are generated.

3.2.4 Output Variable Merge

The implementation of the VM block is detailed in Figure 8. A VM block either selects a variable V(j) directly from the V(1..N) bus, if the variable is not processed in any clause of that stage, or collects V(j) from the C(1..M) bus (see Figure 6). In this way, the variables in bus V'(1..N) will have the same order as in bus V(1..N).

If a variable is processed by more than one clause in the same stage, their priority levels should be merged using a block similar to the ST (stem) block proposed in [20]. However, we noticed that it is possible, with a simple almost linear heuristic, to distribute the clauses among the stages so that in each stage there are no clauses with variables in common. This observation permits avoiding ST blocks and their associated area and delay overheads. It simplifies the circuit and allows a higher clock rate in exchange for a fast and simple pre-processing procedure.

Therefore, our VM block merely consists of programmable MUXs, which, according to the time slot `t`, select the right literal for re-assembling variable V'(i). Therefore, we implement the configuration registers using local content addressable memories (CAMs) having $3M+1$ positions addressed by content `t`. The amount of configuration data per stage for the VM blocks is thus $N(3M+1)\lceil \log_2(L) \rceil$ bits.

3.3 Virtual Hardware

In our configware/software approach the hardware has to be configured according to the formula to be solved. However, the number of clauses that can be accommodated is finite. What can we do if the formula has more clauses than the ones that can fit in the hardware? Since SAT problem instances for real world problems are getting bigger and bigger [2] (especially in the CAD domain), not being able to deal with formulas that exceed the configware capacity would be a serious limitation. A solution to this problem is to divide the formula in hardware pages and run them as they are needed. This solution is called a *virtual hardware* scheme by analogy to the *virtual memory* concept. For a virtual hardware scheme to be efficient, the following is required: the algorithm to partition the virtual circuit in pages and schedule their execution should be (1) efficient and (2) exploit locality. In this paper we deal with issue (1), and we leave issue (2) for further investigations.

Our partitioning algorithm is straightforward. Since the hardware is organized in a pipeline structure, each page is a sequence of consecutive pipeline stages, as many as can fit in one device. The set of all pages forms a virtual clause pipeline of depth $D' = N_P D$, where N_P is the number of pages and D is the depth of the actual pipeline in each page. If the device has a gate capacity S_D, then $N_P = \lceil \frac{S_F}{S_D} \rceil$, where S_F is the size of the circuit that represents a formula F.

The scheduling of pages is fixed. Each page is loaded in its turn to process the variables. One limitation of this scheme is the fact that pages will be loaded even if not used (no relevant events are generated). A solution that we plan to investigate in the future is a **dynamic page scheduling mechanism**. We create a table that gives for each variable the pages where it occurs. Then, only the pages where recently updated variables occur will be scheduled; the others are skipped, contributing to speeding up the system.

4 Preliminary Results

This paper presents the first version of our configware/software system for SAT solving, and respective formula compilation techniques. So far we have developed a software simulator of the system and have obtained a preliminary evaluation. The algorithm of the simulator is based on simulating a one-stage pipeline and then extrapolating to a pipeline of depth D' by noticing that, in the worst case, each level of parallel implications in a one-stage machine corresponds do executing the D' stages of the virtual pipeline. Therefore, the time T_t for computing each implication level is the time for running all hardware pages given by

$$T_t = N_P(T_c + T_v + T_p),$$

where T_c is the time to configure one page, T_v is the time for processing the train of variables through one pipeline stage, and T_p is the pipeline latency.

This simulator has been incorporated in GRASP [13], which can thus simulate the whole configware/software approach. The results obtained for a set of problem instances are shown in Table 2. The `2dlx_cc_bugXX` instances are circuit equivalence checking problems; `ssa2670-1XX` and `bf0432-007` are digital circuit test problems; the remaining problems are well know instances from the DIMACS set [14].

Column 1 gives the name of the instance; columns 2 and 3 give the size in literals before and after conversion to 3-SAT (`Size_b` and `Size_b`, respectively); column 4 gives the number of hardware pages N_P; column 5 gives the CPU time `Tgrasp` taken by GRASP in seconds; column 6 gives the formula compilation time (3-SAT conversion + clause distribution among the stages); column 7 gives the total time taken by our configware system (`Tcw/sw`), including compilation and configuration times; column 8 gives the fraction of time `Frac` GRASP spends running the decision and deduction engines (this is the fraction that will be accelerated using the configware); column 9 gives the speed-up `SU_frac` of the accelerated fraction; finally, column 10 gives the overall speed-up `SU` of the system versus software, according to Amdhal's law.

The above results have been obtained with the following system configuration: variable bus width is $B = 64$; each variable is encoded with $k = 8$ bits (4 for the 0-literals and 4 for the 1-literals) — thus, $N = 7$ variables are processed in parallel in each stage after excluding the 4 control bits; pipeline depth per page is $D = 391$; each stage has width $M = 10$ clauses; PLD capacity is assumed $S_D = 8$ million gates. For this configuration and for formulas of about 10,000 variables we need about 124.1 Kbytes to store each

Name	Size_b	Size_a	N_P	Tgrasp	Tcomp	Tcw/sw	Frac	SU_frac	SU
2dlx_cc_bug44	46933	108427	13	9.77	2.88	1.52	0.93	10.38	2.21
2dlx_cc_bug56	47015	108509	13	5.45	2.89	1.19	0.90	7.38	1.33
2dlx_cc_bug80	47008	108490	13	65.22	2.94	17.37	0.86	6.76	3.21
c1908	11518	13929	2	2395.50	0.36	870.42	0.64	190.52	2.75
ssa2670-141	5238	6774	1	3.00	0.20	0.60	0.81	53.44	3.73
bf0432-007	9574	13588	2	24.48	0.32	6.94	0.73	50.79	3.36
hole9	900	1260	1	31.05	0.04	10.34	0.66	228.40	2.99
hole10	1210	1672	1	43.85	0.05	14.89	0.66	270.12	2.93
par16-1	8788	8788	1	95.84	0.24	31.62	0.73	11.80	3.00
par16-2	9044	9044	1	66.20	0.24	22.34	0.70	15.91	2.93

Table 2: Comparison to software

hardware page and about 19.5 Kbytes to store the variables. Thus, with about 1 Mbyte of total memory capacity we can store about 10 hardware pages. We estimate a clock frequency of about $f = 80$MHz. We also assume that the PLD is a context switching device, that can swap pages in one cycle ($T_c = 1/f$).

The results show several interesting things. First, we can see that the conversion to 3-SAT introduces a certain overhead in the size of the formula in most cases small. The largest overhead (slightly over 2) is for the `2dlx_cc_bugXX` verification instances. These instances also produce a high number of hardware pages (13) and the highest compile time (2.9 secs). As we can see, the compiler is fast even for these large problems. For all instances, the fraction of time spent in decisions and implications varies from 66 to 93% and provides an opportunity for acceleration. In fact, this fraction can be accelerated up to 2 orders of magnitude. The overall speed-up `SU` is always below 4, but this is already interesting for CAD instances. To achieve more speed-up we need to map parts of the diagnosis engine also in hardware.

5 Conclusions

In this paper we have introduced the first configware/software solution for the SAT problem, where the two most time-consuming tasks - computing implications and selecting the next decision variable - are implemented in configware, while conflict diagnosis, backtrack control, and clause database management are performed in software. In this way we can take advantage of the best features in both domains - the massive parallelism of hardware and the algorithmic sophistication of software. Our approach avoids instance-specific place and route, and features a virtual hardware scheme with context switching which allows solving problems that are larger than the configware capacity. The preliminary results show the feasibility of the approach. Future research targets further speed-ups to be achieved by moving some of the diagnosis engine tasks to configware, and by a better page partitioning algorithm.

Acknowledgements

The authors acknowledge support from the Portuguese Foundation of Science and Technology (FCT) in the scope of the AXEL and HIPERSAT research projects.

References

[1] J. Gu. "Satisfiability Problems in VLSI Engineering". In *DIMACS Workshop on Satisfiability Problems*, 1996.

[2] J.P.M. da Silva and K.A. Sakallah. "Boolean Satisfiability in Electronic Design Automation". In *Proceedings of the 37th Conference on Design Automation (DAC)*, pages 675–680, June 2000.

[3] S. Devadas, K. Keutzer, S. Malik, and A. Wang. "Certified Timing Verification and the Transition Delay of a Logic Circuit". In *Proceedings of the 29th Conference on Design Automation*, pages 549–555, June 1992.

[4] P. C. McGeer, A. Saldanha, P. R. Stephan, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. "Timing Analysis and Delay-Fault Test Generation Using Path-Recursive Functions". In *Proceedings of the IEEE International Conference on Computer-Aided Design*, pages 180–183, November 1991.

[5] S. Devadas. "Optimal Layout Via Boolean Satisfiability". In *Proc. Intn'l. Conf. on CAD*, pages 294–297, November 1989.

[6] Gi-Joon Nam, Karem A. Sakallah, and Rob A. Rutenbar. "Satisfiability-Based Layout Revisited: Detailed Routing of Complex FPGAs Via Search-Based

Boolean SAT". In *International Symposium on Field Programmable Gate Arrays (FPGA)*, pages 167–175. ACM Press, February 1999.

[7] T. Larrabee. "Test Pattern Generation using Boolean Satisfiability". *IEEE Trans. on CAD*, 11(1):4–15, January 1992.

[8] P.R. Stephan, R.K. Brayton, and A. Sangiovanni-Vincentelli. "Combinational Test Generation using Satisfiability". *IEEE Trans. on CAD*, 15(9):1167–1176, September 1996.

[9] L.G. Silva, J.P. Marques-Silva, L.M. Silveira, and K.A. Sakallah. "Realistic Delay Modeling in Satisfiability-Based Timing Analysis". In *Proc. IEEE Int. Symp. on Circ. and Syst. (ISCAS)*, 1998.

[10] R. Brayton, G. Hatchel, C. Mullen, and A. Sangiovani-Vincentelli. *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer Academic Publishers, 1984.

[11] J. Gu and R. Puri. "Asynchronous Circuit Synthesis with Boolean Satisfiability". *IEEE Trans. on CAD*, 14(8):961–973, August 1995.

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

[13] J. Marques-Silva and Sakallah. "GRASP: A Search Algorithm for Propositional Satisfiability". *IEEE Transactions on Computers*, 48, 1999.

[14] DIMACS. "Challenge Benchmarks". ftp://dimacs.rutgers.edu/pub/challenge/sat.

[15] T. Suyama, M. Yokoo, and H. Sawada. "Solving Satisfiability Problems on FPGAs". In Reiner W. Hartenstein and Manfred Glesner, editors, *Field-Programmable Logic: Smart Applications, New Paradigms and Compilers*, pages 136–145. Springer-Verlag, September 1996. Proceedings of the 6th International Workshop on Field-Programmable Logic and Applications, FPL 1996. Lecture Notes in Computer Science 1142.

[16] M. Abramovici and D. Saab. "Satisfiability on Reconfigurable Hardware". In W. Luk, P. Cheung, and M. Glesner, editors, *Field-Programmable Logic and Applications*, pages 448–456. Springer-Verlag, 1997. Proceedings of the 7th International Workshop on Field-Programmable Logic and Applications, FPL 1997. Lecture Notes in Computer Science 1304.

[17] M. Platzner and G. De Micheli. "Acceleration of Satisfiability Algorithms by Reconfigurable Hardware". In *Field-Programmable Logic: From FPGAs to Computing Paradigm*, pages 69–78, August 1998. Proceedings of the 8th International Workshop on Field-Programmable Logic and Applications, FPL 1998. Lecture Notes in Computer Science 1482.

[18] Peixin Zhong, Margaret Martonosi, Pranav Ashar, and Sharad Malik. "Accelerating Boolean Satisfiability with Configurable Hardware". In Kenneth L. Pocek and Jeffrey Arnold, editors, *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 186–195, April 1998.

[19] Peixin Zhong, Margaret Martonosi, Pranav Ashar, and Sharad Malik. "Solving Boolean Satisfiability with Dynamic Hardware Configurations". In Reiner W. Hartenstein and Andres Keevallik, editors, *Field-Programmable Logic: From FPGAs to Computing Paradigm*, pages 326–235. Springer-Verlag, August 1998. Proceedings of the 8th International Workshop on Field-Programmable Logic and Applications, FPL 1998. Lecture Notes in Computer Science 1482.

[20] Miron Abramovici, Jose T. de Sousa, and Daniel D. Saab. "A Massively-Parallel Easily-Scalable Satisfiability Solver Using Reconfigurable Hardware". In *Proc. Design Automation Conference (DAC)*, pages 684–690, 1999.

[21] M. Davis and H. Putnam. "A Computing Procedure for Quantication Theory". *ACM*, 7:201–215, 1960.

[22] P. Goel. "An Implicit Enumeration Algorithm to Generate Tests for Combinational Logic Circuits". *IEEE Trans. Comp.*, C-30:215–222, Mar 1981.

[23] A. Rashid, J. Leonard, and W. H. Mangione-Smith. "Dynamic Circuit Generation for Solving Specific Problem Instances of Boolean Satisfiability". In K. L. Pocek and J. M. Arnold, editors, *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, pages 196–204, April 1998.

[24] M. Boyd and T. Larrabee. "ELVIS - A Scalable, Loadable Custom Programmable Logic Device for Solving Boolean Satisfiability Problems". In *IEEE Symposium on FPGAs for Custom Computing Machines*, April 2000.

[25] M. Abramovici and J.T. de Sousa. "A SAT Solver Using Reconfigurable Hardware and Virtual Logic". *Journal of Automated Reasoning*, 24(1/2), February 2000.

[26] Roberto J. Bayardo, Jr. and Robert C. Schrag. "Using CSP Look-Back Techniques to Solve Real-World SAT Instances". In *Proceedings of the 14th National Conference on Artificial Intelligence and 9th Innovative Applications of Artificial Intelligence Conference (AAAI/IAAI)*, pages 203–208, July 1997.

[27] H. Zhang. "SATO: An Efficient Propositional Prover". In William McCune, editor, *Proceedings of the 14th International Conference on Automated deduction*, volume 1249 of *LNAI*, pages 272–275. Springer-Verlag, July 1997.