Published in IET Computers & Digital Techniques
Received on 1st December 2006
Revised on 20th August 2007
doi: 10.1049/iet-cdt:20060221

ISSN 1751-8601

Efficient, scalable hardware engine for Boolean satisfiability and unsatisfiable core extraction

K. Gulati[1] *M. Waghmode*[2] *S.P. Khatri*[1] *W. Shi*[1]

[1]*Department of ECE, Texas A and M University, College Station, TX 77843, USA*
[2]*Magma Design Automation, Inc., Santa Clara, CA 95054, USA*
E-mail: sunilkhatri@tamu.edu

Abstract: Boolean satisfiability (SAT) is a core non polynomial (NP)-complete problem. Several heuristic software and hardware approaches have been proposed to solve this problem. The authors present a hardware solution to the SAT problem. They propose a custom integrated circuit (IC) to implement their approach, in which the traversal of the implication graph as well as conflict clause generation are performed in hardware, in parallel. Further, extracting the minimum unsatisfiable core (i.e. the formula consisting of the smallest set of clauses of the initial formula which is unsatisfiable) is also a computationally hard problem. The proposed hardware approach, in addition to solving SAT, efficiently extracts the minimum unsatisfiable core for any unsatisfiable formula. To the best of the authors' knowledge, this is the first hardware-based solution proposed for extracting the unsatisfiable core. In this approach, clause literals are stored in specially designed clause cells. Clauses are implemented in banks, in a manner that allows clauses of variable width to be accommodated in these banks. To maximise the utilisation of these banks, the authors initially partition the SAT problem. Their solution has significantly larger capacity than existing hardware SAT solvers, and is scalable in the sense that several ICs can be used to simultaneously operate on the same SAT instance. The area, power and performance figures are derived from layout and SPICE (using extracted parasitics) estimates. The approach presented has been functionally validated in Verilog. Preliminary results demonstrate that the approach can accommodate instances with approximately 63 K clauses on a single IC of size 1.5 cm $\times$ 1.5 cm. This hardware based-SAT solving approach results in over three orders of magnitude speed improvement over Boolean constraint propogation-based software SAT approaches (one to two orders of magnitude over other hardware SAT approaches). The capacity of this approach is significantly higher than most hardware-based approaches. Further, the worst case power consumption was found to be $\leq$1 mW for the implementation.

1 Introduction

Boolean satisfiability (SAT) [1] is a classic NP-complete problem, which has been widely studied in the past. Given a set V of variables and a collection C of conjunctive normal form (CNF) clauses over V, the SAT problem consists of determining if there is a satisfying truth assignment for C. If no such assignment exists, C is called an unsatisfiable instance. A subset of C, such that this subset is also an unsatisfiable instance and is called an unsatisfiable core. Formally, given a formula ψ, the formula ψ_C is an unsatisfiable core for ψ iff ψ_C is unsatisfiable and $\psi_C \subseteq \psi$. Computing or extracting the minimum unsatisfiable core of a given unsatisfiable instance, is also reported to be a computationally hard problem [2, 3].

Given the broad applicability of the SAT and the unsatisfiable core extraction problems to several diverse application domains such as logic synthesis, circuit testing, verification, pattern recognition and others [4], there has been much effort devoted to

devising efficient heuristics to solve them. Some of the more well-known software approaches for SAT include [5–8]. Most approaches for extracting the unsatisfiable core are broadly based on the conflict analysis procedure described in [5].

Again, given the broad applicability of the SAT problem, there has been much interest in the hardware implementation of SAT solvers as well. An excellent survey of existing hardware approaches to solve the SAT problem is found in [9]. Although several hardware implementations of SAT solvers have been proposed, there is, to the best of our knowledge, no hardware approach for extracting the unsatisfiable core. We, therefore claim this paper to be the first paper approaching this problem with a hardware-based solution.

Numerous applications can benefit from the ability to speedily obtain a small unsatisfiable core from an unsatisfiable Boolean formula. Applications like planning an assignment [10], can be cast as a SAT instance (equivalently referred to as a CNF instance in the sequel). The satisfiability of this instance implies that there exists a viable scheduling solution. On the other hand, if a planning is proven infeasible as a result of the SAT instance being unsatisfiable, a small unsatisfiable core can help in locating the reason for infeasibility. Similarly, an unsatisfiable instance in FPGA routing [11] implies that the channel is unroutable. A smaller unsatisfiable core in this case would be a geometrically smaller region, with potentially fewer routes, such that the routing is infeasible. Quickly identifying the reason for unroutability is of importance in routing. Further, SAT-based unbounded model checking [12] also requires the efficient extraction of small unsatisfiable cores.

The key motivation for using a hardware approach for SAT or unsatisfiable core extraction is speed. Our hardware-based SAT solver and unsatisfiable core extractor would be well suited for applications wherein the same instance or a slight modification of the instance is solved repeatedly. This property is found in applications like routing, planning or SAT-based unbounded model checking, logic synthesis, VLSI testing, verification and so on. The cost of initial CNF partitioning and of loading the CNF instance onto the hardware is incurred only once, and the speedup obtained with repeated SAT solving would amply recover this cost. Even a modest speed-up of such SAT-based algorithms is of great interest to the VLSI design automation community, since the fraction of the time spent performing SAT checks in these algorithms is very high.

Key requirements for a hardware approach for Boolean satisfiability or unsatisfiable core extraction are capacity and scalability. By capacity of a hardware SAT approach, we mean the largest size of a SAT instance (in terms of number of clauses) that can fit in the hardware. Our proposed solution has significantly larger capacity than existing hardware-based solutions. In our approach, a single IC of size 1.5 cm × 1.5 cm can accommodate CNF instances containing ~63000 clauses (along with the logic required for solving the instance). This is significantly larger than the capacity of previous hardware approaches for Boolean satisfiability. By scalability of a hardware SAT approach, we mean that multiple hardware SAT units can be easily made to operate in tandem, to tackle larger SAT instances.

In this paper, we propose an approach that utilises a custom IC to accelerate the SAT solution and the unsatisfiable core extraction processes, with the goal of speedily solving large instances in a scalable fashion. The hardware implements a variant of general responsibility assignment software patterns (GRASP) [5], that is, slightly modified strategy of conflict driven learning and non-chronological backtracking. For the extraction of the unsatisfiable core, the hardware approach is augmented to implement the approach described in [3]. In this IC, literals and their complement are implemented as custom cells. Clauses of variable width are implemented in banks. Any row of a bank can potentially accommodate more than one clause. The SAT problem is mapped to this architecture in an initial partitioning step, which helps maximise the hardware utilisation. Experimental results are obtained using area, power and performance figures derived from layout and simulation program with integrated circuit emphasis (SPICE) (using extracted layout-level parasitics) estimates. Our hardware approach performs, in parallel, both the tasks of implicit traversal of the implication graph, as well as conflict clause generation. The contribution of this work is to come up with a high capacity, fast, scalable hardware SAT approach. We do not claim to propose any new SAT solution or unsatisfiable core extraction heuristics in this paper. Note that although we used a variant of the Boolean constraint propagation (BCP) engine of GRASP [5] in our hardware SAT solver, the hardware approach can be modified to implement other BCP engines as well. The BCP logic of any BCP-based SAT solver can be ported to an hardware description language (HDL) and directly synthesised in our approach.

2 Previous work

There have been several hardware-based SAT solvers reported in the literature, which are summarised and compared in [9]. Among these approaches, Zhao *et al.* [13, 14] utilise configurable processors to accelerate SAT, demonstrating a maximum speedup of 60× using a board with 121 configurable processors.

doi: 10.1049/iet-cdt:20060221

The largest example mapped to this structure had 24 700 clauses. In [15, 16], the authors describe an FPGA-based SAT accelerator. The speedup obtained was 30 ×, with 64 FPGA boards required to handle an example containing 1280 clauses. The largest example that the approach of [17] handles has about 1300 clauses, with an average speedup of 10 ×. This paper states that the hardware approaches reported in [18–20] do not handle large SAT problems.

In [21, 22], the authors present a software plus configurable hardware (configware)-based approach to accelerate SAT. Software is used to do conflict diagnosis, backtrack and clause management. Configware is used to do implication computation and next decision variable assignment. The speedup over GRASP [5] is between one to two orders of magnitude for the accelerated fraction of the SAT problem. The largest problem tackled has 2 14 304 clauses [22] (after conversion to 3-SAT, which can double the number of clauses [21]). In contrast, our approach performs all tasks in hardware, with a corresponding speedup of one to two orders of magnitude over the existing hardware approaches, as shown in the sequel. In most of the above approaches, the capacity of the proposed approaches is clearly limited and scalability is a significant problem. The approach in this paper is inspired by the requirement of handling significantly larger problems on a single die, and also with the need to allow the design to scale more elegantly. By utilising a custom IC approach, each die can accommodate significantly larger SAT instances than most of what the above approaches report. Our approach is not FPGA-based and can accommodate 63 000 clauses on a single die.

The previous approaches for the extraction of an unsatisfiable core have been software-based techniques. The complexity of this problem has been well studied and algorithms have been reported in [2, 23–25]. Some of the proposed solutions with experimental data to support their algorithms include that given in [26], in which an adaptive search is conducted, guided by clauses' hardness. Goldberg and Novikov, Oh *et al.* and Zhang and Malik [27–29] report resolution-based techniques for generating the empty clause. The unsatisfiable core reported in these cases is the set of clauses involved in the derivation of the empty clause. The minimum unsatisfiability prover from [30] improves upon the existing approaches by removing unnecessary clauses from unsatisfiable sub-formulas to make them minimal.

The approach in [3] attempts to find the minimum unsatisfiable core for a given formula. The augmentation of our hardware architecture for extracting the unsatisfiable core is in accordance with this approach. Broadly speaking, Lynce and Silva [3] employ a SAT solver to search for the minimum unsatisfiable core. This allows a natural match to our hardware-based SAT engine. Resolution-based techniques for unsatisfiable core extraction are not a natural fit to our approach, since resolution is inherently a serial process.

An extended abstract of this paper can be found in [31]. However, Waghmode *et al.* [31] does not include the hardware approach for computing the minimum unsatisfiable core. Also, this paper includes more details about the hardware architecture than that presented in [31], including hardware details for implementing non-chronological backtracking and conflict clause generation. Moreover, the experimental results in this paper include the computation of the worst case power consumption for our approach, which is not present in [31].

The rest of this paper is organised as follows. Section 3 describes the hardware architecture employed in our approach. It includes a discussion on the generation of implications and conflicts (which is done in parallel), along with the hardware partitioning utilised, the communication protocol that banks implement and the generation of conflict induced clauses. An example of conflict clause generation is described in Section 4. Section 5 describes the up-front clause partitioning methodology, which targets maximum utilisation of the hardware. Section 6 describes our approach to finding the unsatisfiable core. The experimental results we have obtained are reported in Section 7. Section 8 concludes with some directions for future work in this area.

3 Hardware architecture

3.1 Abstract overview

Fig. 1 shows an abstracted view of our approach, in order to illustrate the main concept and to explain how BCP [5] is carried out. Note that the physical implementation we use is different from this abstracted view, as subsequent sections will describe. In Fig. 1, the clause bank stores all clauses (a maximum of n clauses on m variables). In the

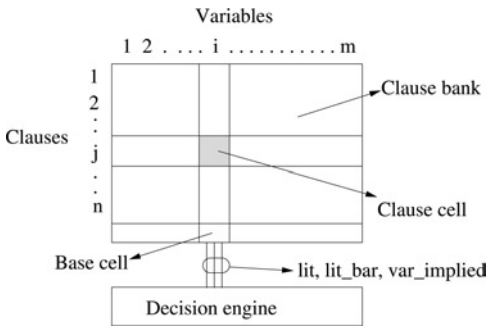


Figure 1 *Abstracted view of the proposed idea*

doi: 10.1049/iet-cdt:20060221

hardware there are $n \cdot m$ clause cells, each of which stores a single literal of the SAT instance. The bank architecture is capable of implicitly storing the implication graph and consequently generating implications and conflicts. A variable is assigned by the *decision engine* and the assignment is communicated to the clause bank via the base cells. The clause bank, in turn, generates implications and possible conflicts because of this assignment. This is done in parallel, at hardware speeds. The base cells sense these implications and conflicts and in turn communicate them back to the decision engine. The decision engine accordingly assigns the next variable or, in case of a conflict, generates a conflict-induced clause and backtracks non-chronologically [5].

As seen in Fig. 1, a column in the bank corresponds to a variable, a row corresponds to a clause and a clause cell corresponds to a literal (which can be positive, negative or absent) in the clause. The clause cell is central to our idea and provides the parallelism obtainable by solving the satisfiability problem in hardware.

The overall flow for solving any SAT instance S consists of first loading S into the clause bank. The hardware then solves S, after which a new SAT instance may be loaded and solved.

3.2 Hardware overview

The actual hardware architecture of our SAT IC differs from the abstracted view of the previous section. The differences are not functional, rather they are caused by circuit partitioning and speed constraints. The different components of the hardware SAT IC are briefly described next.

The hardware details are presented in the following order. The finite state machine for the decision engine is explained in Section 3.3.1. The core circuit structure of our implementation, the clause cell, is capable of computing the implication graph implicitly, and also helps in generating implications and conflicts, all in parallel. This is explained in Section 3.3.2. The implications and conflicts are sensed and forwarded to the decision engine by the base cells. The base cell and its interaction with the decision engine are explained in Section 3.3.3. In practice, we do not have a single clause bank as shown in Fig. 1. Rather, clauses are arranged in several banks, with a limited number of rows (clauses) and columns (variables). Each bank has several strips, which partition the columns of the bank into smaller groups. Between strips, we have special cells which allow us to implement arbitrarily long rows (clauses). The bank and strip structures are explained in Section 3.3.4. Because we partition the hardware into many banks, it is possible that a particular variable occurs in several banks. Therefore implications or assignments on such variables, generated in a bank b_1, must be communicated to other banks b_i where the same variable occurs. This communication is performed by a hierarchical arrangement of communication units, arranged in a tree fashion. The details of this inter-bank communication are provided in Section 3.3.5. Fig. 2 describes the banks and the inter-bank communication units. It also shows the centrally located BCP() engine, as well as the banks for storing conflict induced clauses.

3.3 Hardware details

3.3.1 Decision engine: Fig. 3 shows the state machine of the decision engine. To begin with, the CNF instance is loaded onto the hardware. Our hardware uses dynamic circuits so all signals are initialised into their precharged or predischarged states (in the `refresh` state). The decision engine assigns the variables in the order of their `identification tag`, which is a numerical ID for each variable, statically assigned such that most commonly occurring variables are assigned a lower tag. The decision engine assigns a variable (in `assign_next_variable` state) and this assignment is forwarded to the banks via the base cells. The decision engine then waits for the banks to compute all the implications during

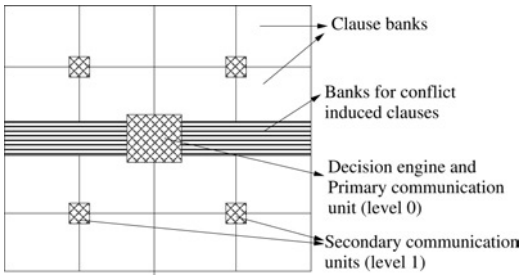


Figure 2 *Generic floorplan*

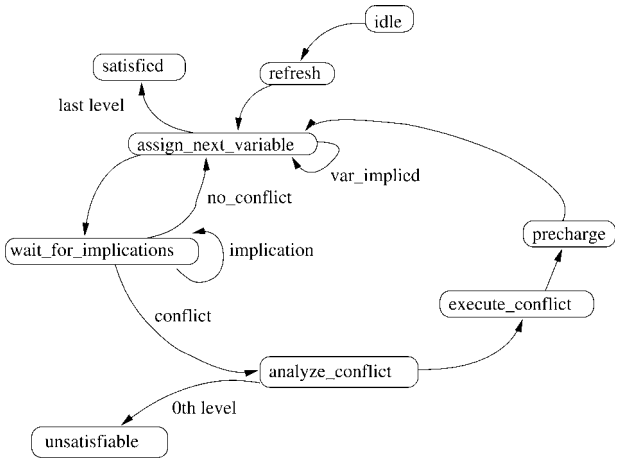


Figure 3 *State diagram of the decision engine*

wait_for_implications state. If no conflict is generated as a result of the assignment, the decision engine assigns the next variable. If there is a conflict, all the variables participating in the conflict clause are communicated by the banks to the decision engine via the base cell. Based on this information, during the analyse_conflict state, the base cell generates the conflict induced clause and then stores it in the clause bank. Also, it non-chronologically backtracks according to the GRASP [5] algorithm. Each variable in a bank retains the decision level of the current assignment/implication. When the backtrack level is lower than this stored decision level, then the stored decision level is cleared before further action by the decision engine during the execute_conflict state. After a conflict is analysed, the banks are again precharged (in the precharge state) and the backtracked decision is applied to the banks. If all the variables have either been assigned or implied with no conflicts, (this is detected from the assignment on the last level) the CNF instance is reported to be 'Satisfiable' (in the satisfied state of the decision engine finite state machine). On the other hand, if the decision engine has already backtracked on the variable at the 0th level and a conflict still exists, the CNF instance is reported to be 'Unsatisfiable' (in the unsatisfiable state).

3.3.2 Clause cell: Fig. 4 shows the signal interface of a clause cell. Fig. 5 provides details of the clause cell structure. Each column (variable) in the bank has three signals, lit, lit_bar and var_implied, which are used to communicate assignments, implications and conflicts on that variable. Each row (clause) in the bank has a signal clausesat_bar to indicate if the clause is satisfied. The 2-bit free_lit_cnt signals serve as an indicator of number of free literals in the clause. If the literal in the clause cell is free (indicated by iamfree) then out_free_lit_cnt is one more than in_free_lit_cnt. The imp_drv and cclause_drv signals facilitate generation of implications and conflict clauses, respectively. Also, each row has a termination cell at its end (which we assume is at the right side of the row) which drives the imp_drv and cclause_drv

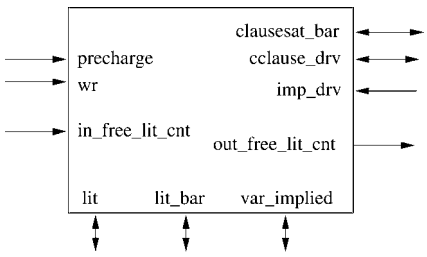


Figure 4 *Signal interface of the clause cell*

signals. We next describe the encoding of these signals and how they are employed to perform BCP.

Note that the signals lit, lit_bar, var_implied and cclause_drv are predischarged and clausesat_bar is a precharged signal. Also, each clause cell has two single bit registers namely reg and reg_bar to store the literal of the clause. The data in these registers can be driven in or driven out on the lit and lit_bar signals.

A variable is said to participate in a clause if it appears as a positive or negative literal in the clause. The encoding of the reg and reg_bar bits is as shown in Table 1. The iamfree signal for a variable indicates that the variable has not been assigned a value yet, nor has it been implied.

The assignments and failure-driven assertions [5] are driven on lit, lit_bar and var_implied signals by the decision engine whereas implications are driven by the clause cells. Communication in both directions (i.e. from clause cell to the decision engine and vice-versa) is performed via the base cells using the above signals. There exists a base cell for each variable. Table 2 lists the encoding of the lit, lit_bar and var_implied signals.

If a variable V_i participates in clause C_j and no value has been assigned or implied on the lit and lit_bar signals for V_i, then V_i is said to contribute a free literal to clause C_j. This is indicated by the assertion of the signal iamfree for the (j, i)th clause cell. Also, a clause is satisfied when variable V_i participates in clause C_j and the value on the lit and lit_bar signals for V_i matches the register bits in clause cell c_{ji}. In such a case, the precharged signal clausesat_bar for C_j is pulled down by c_{ji}.

If clause C_j has only one free literal and C_j is unsatisfied, then C_j is called a unit clause [5]. When C_j becomes a unit clause with c_{ji} as the only free literal, its termination cell senses this condition by monitoring the value of free_lit_cnt and testing if its value is 1. If free_lit_cnt is found to be 1, the termination cell asserts the imp_drv signal. When c_{ji} (which is the free literal cell) senses the assertion of imp_drv, then it drives out its reg and reg_bar values on the lit and lit_bar wires and also asserts its var_implied signal, indicating an implication on variable V_i.

A conflict is indicated by the assertion of the cclause_drv signal. It can be asserted by the termination cell or a clause cell. The termination cell asserts cclause_drv when free_lit_cnt indicates that there is no free literal in the clause and the

doi: 10.1049/iet-cdt:20060221

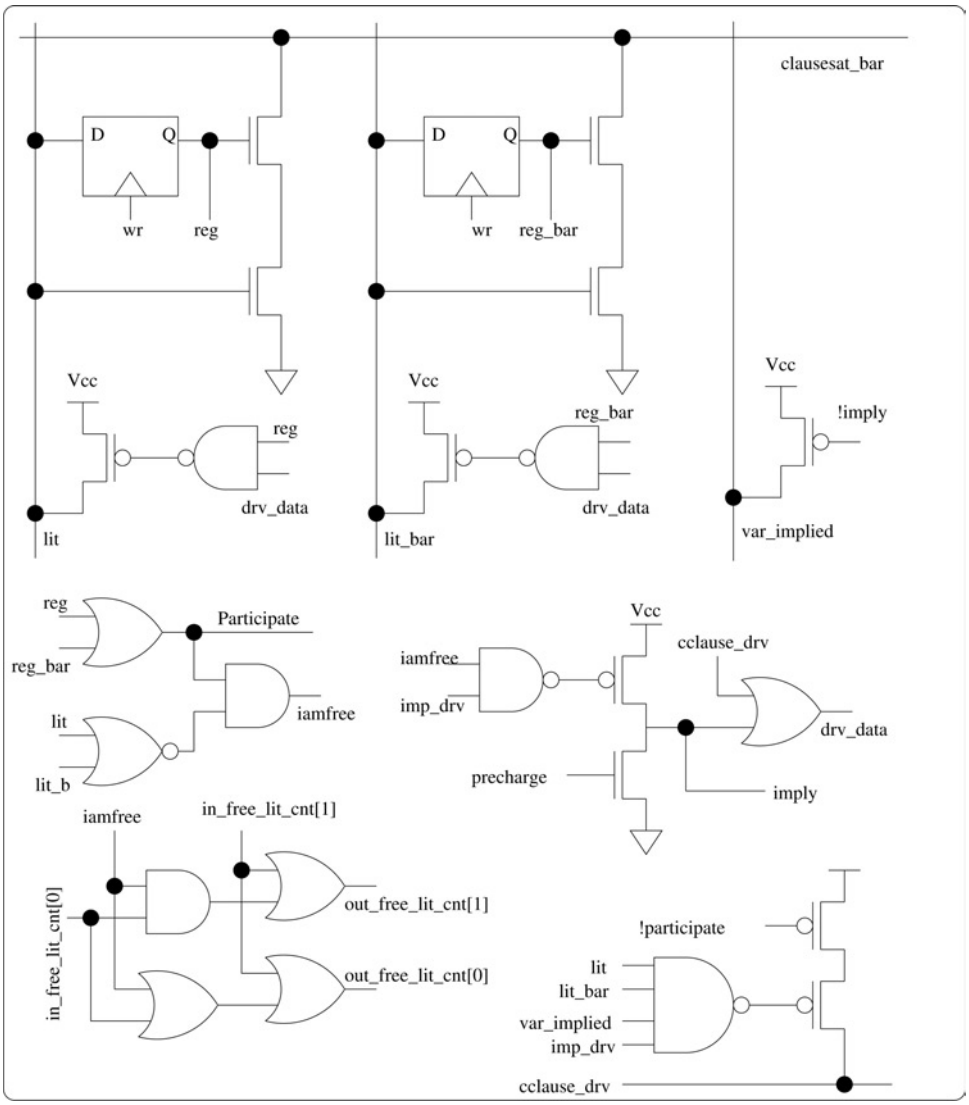


Figure 5 *Schematic of the clause cell*

clause is unsatisfied (indicated by `clausesat_bar` staying precharged). A participating clause cell c_{ji} asserts `cclause_drv` for clause C_j when it detects a conflict on variable V_i, and senses `imp_drv`. When `cclause_drv` is asserted for clause C_j, all the clause cells in C_j drive out their respective `reg` and `reg_bar` values on the respective `lit` and `lit_bar` wires. In other words the `drv_data` signal for the (j, i)th clause cell is asserted (or `reg` and `reg_bar` are driven out on `lit` and `lit_bar`) when either of (i) `cclause_drv` is asserted or (ii) `imp_drv` is asserted, and the current clause cell has its `iamfree` signal asserted. Thus, if two clauses cause different implications on a variable, both clauses will drive out all their literals (which will both be high, since `lit` and `lit_bar` are predischarged signals). This indicates a conflict to the decision engine, which monitors the state of `lit`, `lit_bar` and `var_implied` for each variable. This can trigger a chain of `cclause_drv` assertions leading to back-tracing of the implication graph in parallel, which causes all the variables taking part in the conflict clause to be identified.

Table 1 Encoding of {*reg, reg_bar*} bits

Encoding	Meaning
00	variable does not participate in clause
10	variable participates as a positive literal
01	variable participates as a negative literal
11	illegal

Fig. 6 shows the layout view of our clause cell. The layout, generated in a full-custom manner, had a size of 12 μm by 9 μm, and was implemented in a 0.1 μm technology.

3.3.3 Base cell: There is one base cell for each variable in a bank. The base cell performs several functions. It

doi: 10.1049/iet-cdt:20060221

Table 2 Encoding of {*lit,lit_bar*} and *var_implied* signals

Encoding		Meaning
00	0	variable is neither assigned nor implied
01	0	value 0 is assigned to the variable
10	0	value 1 is assigned to the variable
01	1	value 0 is implied on the variable
10	1	value 1 is implied on the variable
11	1	0 as well as 1 implied, i.e. conflict
11	0	variable participates in conflict induced clause
00	1	illegal

stores information about its variable (its `identification tag`, value, decision level and assigned/implied state). It also detects an implication on the variable, participates in generating the conflict induced clause, and helps in performing non-chronological backtrack. These aspects of the base cell functionality are discussed next, after an explanation of its signal interface.

- *Signal interface:* Fig. 7 shows the signal interface of the base cell. The signals `lit`, `lit_bar` and `var_implied` in the base cell are bidirectional and are the means of communication between the decision engine and the clause bank. This communication is directed by the base cell. The signal `curr_lvl` stores the value of the current decision level. The base cell of each variable keeps track of any decision or implication on its variable through the signals `assign_val` and `imply_val`, respectively. The signal `identify_cclause` is used during conflict analysis as described later. The `bck_lvl` signal indicates the level that the engine backtracks to, in case of a conflict. The

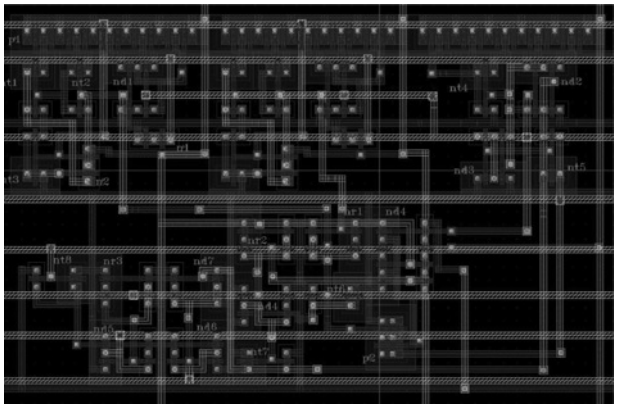


Figure 6 *Layout of the clause cell*

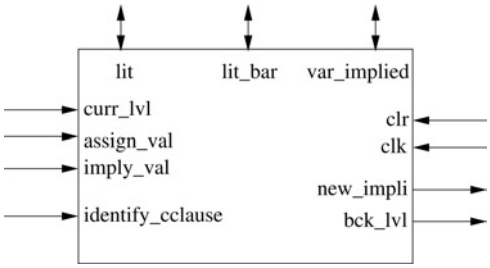


Figure 7 *Signal interface of the base cell*

`new_impli` signal is driven when an implication is detected.

- *Detecting implications:* Fig. 8 shows the circuitry in the base cell to generate the `new_impli` signal, which is high for one clock cycle when an implication occurs (this constraint is required for the decision engine to remain in the state `wait_for_implications` while there are any new implications (indicated by `new_impli`)). This is done as follows. Initially both the flip-flop outputs are low. When the `var_implied` signal is high during the positive edge of a clock pulse, the flip-flop labelled *A* has its output driven high. This causes the output of the AND gate feeding the wired-OR to be driven high. In the next clock pulse, the flip-flop labeled *B* has its output driven high. This signal pulls the output of the AND gate (feeding the wired-OR) low. Thus, due to a `var_implied` signal, the `new_impli` is high for exactly one clock pulse. The flip-flops are cleared using the `clr` signal, which is controlled by the decision engine. The `clr` is asserted during the `refresh` state for all base cells, and during the `execute_conflict` state (for base cells having a decision level higher than the current backtrack level `bck_lvl`).

- *Conflict clause generation:* The base cell also has the logic to identify a conflict clause literal and appropriately communicate it to the clause banks (for the purpose of creating a new conflict clause). During the analyse_conflict state, the decision engine sets the `identify_cclause` signal high. The base cell then records the current values of `lit`, `lit_bar` and `var_implied`. If the tuple is equal to 110, the base celldrives the complement of this variable to

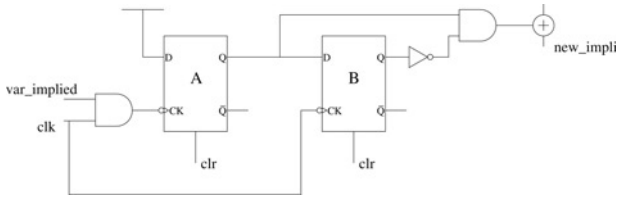


Figure 8 *Indicating a new implication*

doi: 10.1049/iet-cdt:20060221

the clause bank and asserts the clause write signal (wr) for the next available clause. This ensures that the conflict clause is written into the clause bank. Thus, any variable participating in the current conflict and having its lit, lit_bar and var_implied as 110 is recorded and hence, the conflict induced clause is generated.

As the conflict induced clauses are generated dynamically, the width of the conflict clause banks can not be fixed while programming the CNF instance in the hardware. Therefore the width of conflict induced clause banks is kept equal to the number of variables in the given CNF instance. The decision engine can still pack more than one conflict induced clause in one row of the conflict clause banks. To be able to use the space in the conflict induced clause banks effectively, we propose to store only the clauses having fewer literals than a pre-determined limit, updated in a first-in-first-out manner (such that old clauses are replaced by newly generated clauses). Further, we can utilise the clause banks for regular or conflict clauses, allowing our approach to devote a variable number of banks for conflict clauses, depending on the SAT instance.

- *Non-chronological backtrack:* The decision level to which the SAT solver backtracks, in case of a conflict, is determined by the base cell. The schematic for this logic is described next. Fig. 9 shows the circuitry in the base cell to determine the backtrack level [5]. The signal my_lvl is the decision level associated with the variable. The signal bck_lvl (backtrack level) is a wired-OR signal. The variable that has the highest decision level among all the variables participating in a conflict sets the value of bck_lvl to its my_lvl. This is done as follows. Let the set of variables participating in the conflict be called C. Let v^{max} be the variable with the highest decision level among all variables $v \in C$. Each bit of every variable v's decision level is XNORed with the corresponding bit of the current value of bck_lvl. If the most significant bits my_lvl[k] and bck_lvl[k] are equal (which makes the output of the corresponding XNOR high) then the output of the XNOR of the next most significant bits are checked and so on. If for a certain bit i, my_lvl[i] is low and bck_lvl[i] is high, then the value of bck_lvl is higher than this variable's my_lvl. The output of the XNOR of the rest of the lesser significant bits ($j < i$) for this variable are ignored. This is done by ANDing the output of the ith bit's XNOR with the my_lvl[$i-1$] bit, to obtain a '0' result which is wire-ORed into bck_lvl[$i-1$]. This in turn gets trickled down to the my_lvl of the least significant bit. On the other hand, in case my_lvl[i] is high and bck_lvl[i] is low, then the AND gate feeding the wired-OR for the ith bit, would drive a high value to the wired-OR and hence update bck_lvl[i] to high. The above continues until all the bits of bck_lvl are equal to the corresponding bits of v^{max}'s decision level.

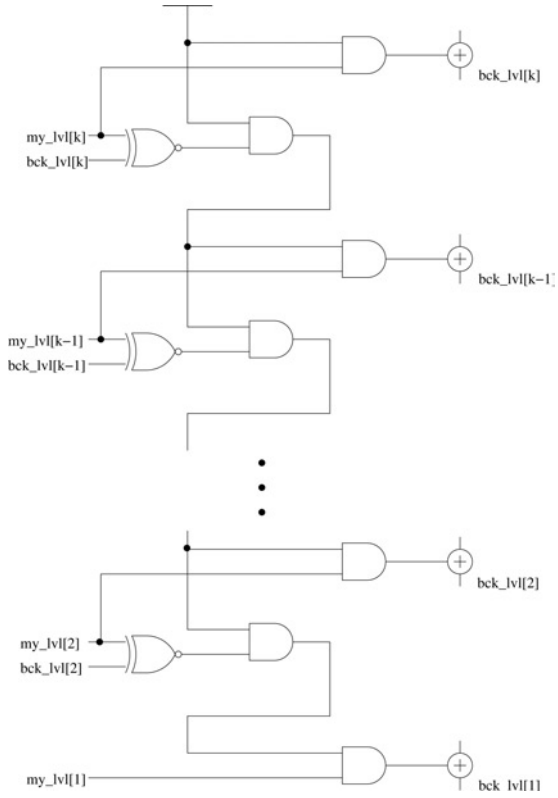


Figure 9 *Computing backtrack level*

Our hardware SAT solver, consisting of clause banks, clause cells, base cells, decision engine, conflict generation, BCP and non-chronological backtracking, has been implemented in Verilog, and has been simulated and verified for correctness.

3.3.4 Partitioning the hardware: In a CNF instance, a very small subset of variables participate in a single clause. Thus, putting all the clauses in one monolithic bank, as shown in the abstracted view of the hardware (Fig. 1) results in a lot of non-participating clause cells. For the center for Discrete Mathematics and Theoretical Computer Science (DIMACS) [32] examples, on average, more than 99% of the clause cells do not participate in the clauses if we arrange the clauses in one bank. Therefore we partition the given CNF instance into disjoint subsets of clauses and put each subset in a separate clause bank. Although a clause is fully contained in one bank, note that a variable may appear in more than one banks.

Fig. 10 depicts an individual bank. Each bank is further divided into strips to facilitate a dense packing of clauses (such that the non-participating clause cells are minimised). We try to fit more than one clause per row with the help of strips. This is achieved by inserting a

doi: 10.1049/iet-cdt:20060221

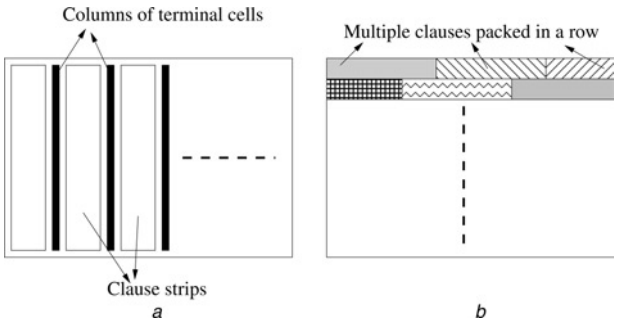


Figure 10 *Schmatic diagram of an individual bank*

a Internal structure of a bank
b Multiple clauses packed in one bank-row

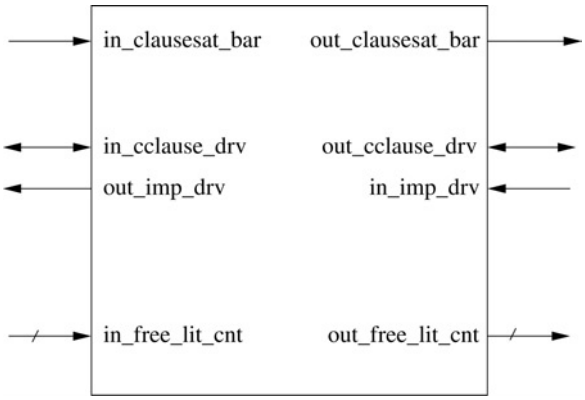


Figure 11 *Signal interface of the terminal cell*

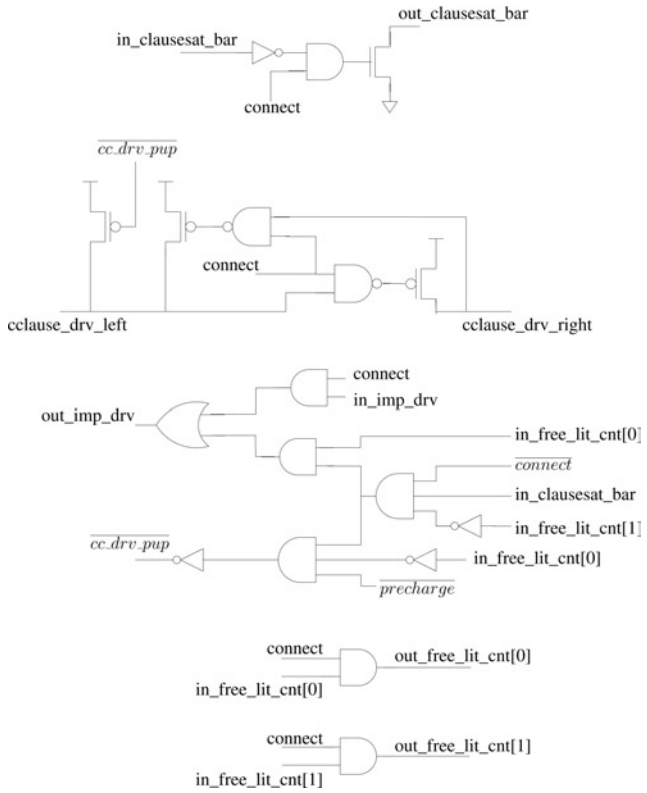


Figure 12 *Schematic of a terminal cell*

column of `terminal cells` between the strips. Fig. 11 describes the signal interface of the terminal cell, whereas Fig. 12 shows the detailed schematic of the terminal cell. Each terminal cell has a programmable register bit indicating if the cell should act as a mere connection between the strips or act as a clause termination cell. While acting as a connection, the terminal cell repeats the `clausesat_bar`, `cclause_drv`, `imp_drv` and `free_lit_cnt` signals across the strips, thereby expanding a clause over multiple strips. However, while acting as a clause termination cell, it generates `imp_drv` and `cclause_drv` signals for the clause being terminated. A new clause can start from the next strip (the strip to the right of the terminal cell).

The number of clause cell columns in a bank (or a strip) is called the width of a bank (or a strip) and number of rows in a bank is called height of a bank. On the basis of extensive experimentation, we settled on 25 rows and 6 columns in a strip. With the help of terminal cells, we can connect as many strips as needed in a bank. Consequently, a bank will have 25 rows but its width is variable since the bank can have any number of strips connected to each other through the terminal cells.

The algorithm for partitioning the problem into banks, and for packing the clauses of any bank into its strips (to minimise the number of non-participating cells) is described in Section 5. Also, experimental results and optimal dimensions of the banks and strips are presented in Section 7.

3.3.5 Inter-bank communication: Since a variable may appear in multiple banks (we refer to such variables as repeated variables), implications on such variables need to be communicated between the banks. Also, the assignments done by the decision engine need to be communicated to the banks and the implications or conflict clauses generated in the bank need to be communicated back to the decision engine.

In our design, we employ a hierarchical arrangement of communication units to perform this communication between the banks and the decision engine, as depicted in Fig. 13. Each column in the bank has a base cell that actually drives and senses the `lit`, `lit_bar` and `var_implied` signals for that variable, and communicates with the decision engine through a hierarchy of communication units. As seen in Fig. 13, the communication units and base cells form a tree structure. The communication unit directly interacting with the decision engine is said to be at 0th level of hierarchy and base cells are said to be at the highest level of hierarchy.

Each variable is associated with an identification tag as explained in Section 3.3.1. Every base cell has a

doi: 10.1049/iet-cdt:20060221

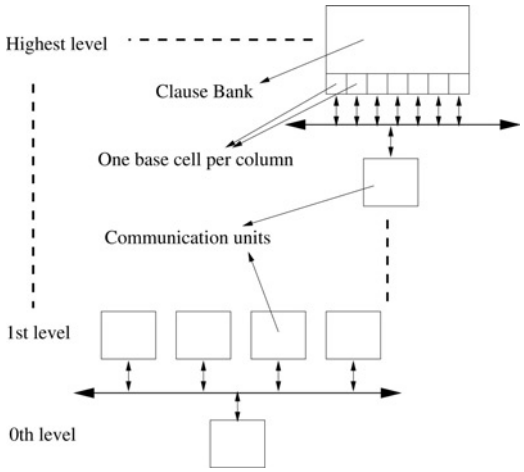


Figure 13 *Hierarchical structure for inter-bank communication*

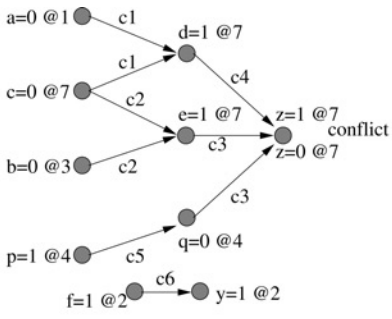
register to store the identification tag of the variable it represents. The base cells and the decision engine use the identification tags to communicate assignments, implications, conflict clause variables and backtrack level. A base cell also has a programmable register bit named repeat bit and a register named repeat level. The repeat bit indicates if the variable represented by the base cell is a repeated variable. The repeat level register for any variable v is pre-programmed with the hierarchy level of the communication unit that forms the root of the subtree containing all the base cells containing that repeated variable v. If the repeat bit for variable v is set, and an implication has occurred on v, the base cell of the variable v communicates the implied value, its identification tag and its repeat level to the communication unit C at the next lower level of hierarchy. The communication unit C communicates these data to other communication units at lower levels, if the repeat level of the implied variable v is lower than its own hierarchy level. In this way, the inter-bank implication communication is completed using the smallest possible communication subtree, allowing for maximal parallelism during inter-bank communication.

The assignments made by the decision engine are broadcast to all levels. The variables participating in the conflict induced clause are also communicated to the decision engine via this hierarchy.

Fig. 2 shows the proposed floorplan. The decision engine is at the centre of the chip surrounded by the clause banks. Additional banks required to store the conflict induced clauses are also near the centre of the chip. Each communication unit resides at the centre of the chip area occupied by the banks in its communication subtree, as shown in Fig. 2.

4 An example of conflict clause generation

Fig. 14 shows an example CNF instance, its implication graph and how it is implicitly traversed in this scheme.



a

c_1	$(a + c + d)$
c_2	$(b + c + e)$
c_3	$(\bar{z} + \bar{e} + q)$
c_4	$(\bar{d} + z)$
c_5	$(\bar{p} + \bar{q})$
c_6	$(\bar{f} + y)$

b

	a	b	c	d	e	f	p	q	y	z
Initial(predischarge)	000	000	000	000	000	000	000	000	000	000
Assignments till @7	010	010				100	100			
Implications till @7								011	101	
Assignment @7			010							
Implications @7				101	101					
Conflict at @7										111
Backtracing				111	111			111		111
Conflict clause variables	110	110	110				110			

c

Figure 14 *Example of implicit traversal of implication graph*

a Implication graph
b CNF instance
c Implicit, parallel generation of conflict induced clause

doi: 10.1049/iet-cdt:20060221

$c_1, \ldots, c_6$ are the clauses as shown in Fig. 14*b*. Let us call the `lit`, `lit_bar` and `var_implied` signals for a variable as a signal triplet. Initially all signal triplets are predischarged and held at high impedance. The implication graph in Fig. 14*a* shows a conflict occurring at decision level 7. $a = 0$, $b = 0$, $p = 1$ and $f = 1$ are the assignments made before level 7 and $q = 0$ and $y = 1$ are the implications caused by them. Fig. 14*c* shows the transitions occurring on the signal triplet of each variable. Decisions are reflected as logic low and implication as logic high on the `var_implied` signal. The decision $c = 0$ at level 7 causes implications on d and e because of clauses c_1 and c_2, respectively. It results in c_3 and c_4 imposing conflicting requirements on the value of z. Therefore c_3 drives 011 and c_4 drives 101 on the signal triplet of z and the resultant status on z becomes 111. Note that triplet signals that are 0 are initially predischarged, so that they can be driven to 1 during the implication graph analysis. After the occurrence of a conflict, an implicit process of back-traversal of the graph starts in hardware. The conflict on z causes the assertion of the `cclause_drv` signal in c_3 and c_4 which in turn causes the data in their registers to be driven on the `lit` and `lit_bar` signals. Thus, 111 gets driven on the signal triplets of d due to c_4, and e and q due to c_3 (as they are implied variables). The 111 on d causes the assertion of `cclause_drv` in c_1, resulting in 110 on a and c as they are decision variables. Similarly 110 is driven on b and c due to c_2 and on p due to c_5. And thus, the variables taking part in the conflict clause are a, b, c and p and the conflict clause is formed by inverting their assigned values, that is, $(a + b + c + \bar{p})$. Also, it can be seen that the status on f and y does not change as they are not a part of the conflict graph. Thus implications and conflict clauses are implicitly generated and in parallel, and hence the process is quite fast.

In case of multiple conflicts, our approach would create a single conflict clause which is the disjunction of all the new conflict clauses. This leads to lesser pruning of the search space as compared with storing the new conflict clauses individually.

In the current form, our hardware SAT solver only records the last row of the table (only the variables with decisions) in the conflict clause. A possible extension of our approach for generating smaller clauses (with fewer literals) is to store a row which is below the row corresponding to the conflict (i.e. row 7 of Fig. 14*c*), and has the smallest number of entries (excluding the entry for the variable on which the conflict is detected). For example, the literals of row 8 of Fig. 14*c* would yield a conflict clause $(\bar{d} + \bar{e} + q)$. Variable z would not be added in this conflict clause since it is the variable on which the conflict is detected. Adding this variable would not help in pruning the search space efficiently.

5 Partitioning the CNF instance

This section describes the algorithms used to partition the given CNF instance into banks and strips. We cast these problems as hypergraph-partitioning problems, and use hMetis [33] to solve them.

To partition the CNF instance into multiple banks, we represent the clauses as vertices in the hypergraph and variables as hyperedges. Let $C = c_1, c_2, \ldots, c_n$ be the set of all clauses and $V = v_1, v_2, \ldots, v_m$ be the set of all variables in the given CNF instance. Then the resultant hypergraph is $G = (U, E)$, where $U = u_1, u_2, \ldots, u_n$ is a set of n vertices each corresponding to a clause in C and $E = e_1, e_2, \ldots, e_m$ is a set of m hyperedges each corresponding to a variable in V. Edge e_i connects vertex u_j if and only if variable v_i participates in clause c_j. This hypergraph is partitioned with hMetis such that each balanced partition contains k vertices and the number of hyperedges cut due to partitioning is minimised.

To partition a bank into strips, we represent the clauses as hyperedges and variables as vertices in the hypergraph. Similar to the above construction, let $C_i = c_1, c_2, \ldots, c_k$ be the set of clauses and $V_i = v_1, v_2, \ldots, v_l$ be the set of variables in bank B_i. Then the resultant hypergraph is $G_i = (U_i, E_i)$, where $U_i = u_1, u_2, \ldots, u_l$ is a set of l vertices each corresponding to a variable in V_i and $E_i = e_1, e_2, \ldots, e_k$ is a set of k hyperedges each corresponding to a clause in C_i. Edge $e_p \in E_i$ connects vertex $u_q \in U_i$ if and only if variable v_q participates in clause c_p.

After each bank is partitioned into strips, we need to order the strips so as to minimise the number of rows required to fit the clauses in the bank. For this purpose, we use a two-dimensional graph bandwidth minimisation algorithm and then use a greedy bin-packing approach to pack the clauses in the rows. Fig. 10 depicts this packing of multiple clauses in one row. The details of the diagonalisation and greedy bin-packing algorithm are omitted from this description because of space constraints.

6 Extraction of the unsatisfiable core

The work in [3] proposes a SAT-based algorithm for computing the minimum unsatisfiable core. The approach given in [3] in brief is as follows: given a Boolean formula ψ defined over n variables, $X = x_1, \ldots, x_n$, such that ψ has m clauses, $\Omega = \omega_1, \ldots, \omega_m$, the approach begins with the definition of a set S of m new variables $S = s_1, \ldots, s_m$, and the creation of a

doi: 10.1049/iet-cdt:20060221

new formula ψ' defined on $n + m$ variables, $X \cup S$, with m clauses $\Omega' = \omega_1', \ldots, \omega_m'$. Each clause $\omega_i' \in \psi'$ is derived from a corresponding clause $\omega_i \in \psi$ as $\omega_i' = \neg s_i + \omega_i$. For a certain assignment to the variables in S, ψ' can be satisfiable or unsatisfiable. The minimum unsatisfiable core is obtained from the unsatisfiable sub-formula with the least number of S variables assigned to value 1.

The model given in [3] can be seamlessly implemented in our hardware architecture. This is because this model simply extends the SAT problem. Since our approach exploits the parallelism that is inherent in any SAT problem, the two approaches can be naturally integrated. The experimental results reported in [3] are strongly limited by the number of variables and clauses in the problem instances. Although they compute the minimum unsatisfiable core, which was not reported by earlier approaches, the complexity of the model is significant for a software-based SAT solver. Testing on bigger instances was limited due to the inability of software SAT solvers to handle such instances. This is where our hardware-based SAT solver fits in. It elegantly complements their approach by providing a fast and scalable SAT solver to find the unsatisfiable core. Pseudocode for this algorithm is shown in Fig. 15.

The following changes are made to our architecture for implementing the above approach. In order to introduce the set S of m new variables (m is the initial number of clauses), the number of base cells are increased by m. The identification tag of any new variables (which is also the decision level of the new variables), is set to be lower than all the variables in the original SAT instance. Also since we add a new variable to each clause, we have to add a new clause cell in each of the m clauses. Since we use efficient SAT instance partitioning, clause bank partitioning and clause packing techniques, the overhead in terms of new clause cells required is $\leq m^2$. The extraction procedure (`min_clause_solve`(ψ')) for the unsatisfiable core proceeds as follows. We perform repeated invocations of the hardware SAT solver with a different set of variables $S' \subseteq S$ being assigned to 1. For a certain run, prior to the first assignment made by the decision engine, the signals `lit`, `lit_bar` and `var_implied` for all the variables in S' are driven to 100 (i.e. forcing a decision of 1 on all variables $s_i \in S'$). If the SAT solver reports the SAT instance as unsatisfiable, the clauses containing $s_i \in S'$ are recorded. The corresponding clauses of the original SAT instance together make one unsatisfiable core. Next, a new clause consisting of all the variables in S' is added to the clause bank in a manner similar to adding a conflict induced clause. In other words, we add a clause $\sum(\neg s_i)$, where $s_i \in S'$, to the instance. This new clause avoids generating the same unsatisfiable core in future runs. Among all the unsatisfiable cores, the core with the smallest number of clauses is the minimum unsatisfiable core and is finally reported.

```
min_unsat_core(ψ(X, Ω)){
S ← add_new_variables(|Ω|) // add variables s1, s2, ... ,sm
ψ' ← Φ
for i = 1; i ≤ |Ω|; i++ do
   ωi' ← ¬si + ωi
   ψ' ← ψ' ∪ ωi'
end for
min_clause_solve(ψ') // explained in text
}
```

Figure 15 *Pseudocode for extracting the minimum unsatisfiable core*

Other existing optimisation techniques that are discussed in [3] can also be easily grafted in the modified hardware SAT solver. For example, any conflict induced clause containing only variables $s_i \in S$ also generates an unsatisfiable core. This is because the clauses of the original SAT instance, corresponding to the clauses which contain s_i, represent an unsatisfiable core and can be recorded.

7 Experimental results

To validate our ideas, we tested several examples from the DIMACS [32] test suite and from the SAT-2004 [34] competition benchmark suite. The examples we used are listed in Table 3, along with the number of clauses and variables (Columns 1 through 3). For a IC of size 1.5 cm on a side, we can accommodate 1.875 million clause cells. The total number of strips in the IC is therefore 12 500. The IC implements a total of six hierarchical levels in the inter-bank communication methodology.

We tested the functionality of the clause and termination cells, the implication generation and conflict clause generation logic in Verilog. The chip level performance estimates were obtained by running SPICE [35], using layout-extracted parasitics. The hardware SAT IC was implemented in a 0.1 μm process, with a voltage supply (VDD) of 1.2 V.

For all the examples listed in Table 3, we performed partitioning (into banks) and binning (into strips) as described in Section 5. The initial partitioning was performed to create banks with 200 clauses. We define the packing factor (PF) as a figure of merit for the partitioning and binning procedure.

$$\text{PF} = \frac{\text{Total no. of cells}}{\text{No. of participating cells}}$$

The PF before partitioning and binning is shown in Column 4. This corresponds to the PF of a monolithic

doi: 10.1049/iet-cdt:20060221

Table 3 Partitioning and binning results

Instance	No. of Clauses	No. of Vars	PF (initial)	PF (opt.)	No. of strips	Avg no. of strips per cl.
par16-3	3344	1014	379	9.53	486	1.93
ii8b4	8214	1067	474	14.68	1548	2.19
am	7814	2268	835	8.42	1021	2.04
par32-5	10325	3175	1183	9.01	1426	1.76
ii16a1	19368	1649	719	25.71	10514	2.87
ii32c4	20862	758	137	12.45	8178	4.57
dekker	58308	19472	8346	10.40	8084	1.78
frg2mul	62943	10313	3063	8.68	10514	2.41

implementation. Note that this can be as high as a few 1000. The PF after partitioning and binning is shown in Column 5, and it is about ten on average. Attempting to lower the PF beyond this value results in several variables appearing in multiple banks. The total number of strips for all the examples are shown in Column 6. Note that all examples require less than 12 500 strips, indicating that they would fit on our IC. This is a dramatic improvement in capacity over existing monolithic hardware-based SAT approaches, which can handle between 1280 and 24 700 clauses with 64 FPGA boards or 121 configurable processors, respectively, as opposed to about 63 000 clauses on a single die for our approach. Further, the total run-time for the partitioning (using hMetis [33]), diagonalisation and greedy bin-packing for the examples listed in Table 3 ranged from 8 to 200 seconds on a 3.6 GHz, 3 GB machine running Linux. These runtimes are significantly lower than the BCP-based software SAT runtimes for these examples. Even if the partitioning runtimes were higher, the time spent in partitioning is amply recovered when multiple SAT calls need to be made for the same instance.

The delay of each bank (the difference between the time a new decision variable is driven to the time the last implication is driven out by the bank) was computed via SPICE simulations to be $\Delta_B = 3$ ns (for a bank with 3 strips, which is approximately the average number of strips per clause as indicated in Column 7 of Table 3). We also estimated the delay due to the inter-bank communication via SPICE simulations. To do this, we first found the average number of implications caused by any decision, over all the examples under consideration. The average number of implications per decision was found to be about 21. For the computation of delay due to inter-bank communication, we conservatively assumed that the average number of implications per decision was 25. We assumed the worst-case situation (where each of these 25 implications are on variables that repeat across banks, with a repeat level of 0). This results in the slowest inter-bank communication scenario. Using SPICE delay values (computed using layout-extracted wiring parasitics), we obtained the values of the delay between communication units at level i and $i+1$. Let this delay be denoted by Δ_i. Then the total delay is estimated as

$$\Delta_C = 2 \cdot 25 \cdot \sum_{i=0}^{5} (\Delta_i) + \Delta_B$$

Note that long wires (between communication units at different repeat levels) are optimally buffered for minimal delay. Using the values of Δ_i that we obtained, Δ_C is computed to be 27 ns. Using this estimate, we compute the time for the accelerated fraction of the SAT problem in our hardware SAT engine as

$$\text{Our runtime} = \text{Number of decisions} \cdot \Delta_C$$

The worst case time to generate and communicate implications (Δ_C) dominates the conflict analysis time, and hence our runtime estimates are based on Δ_C alone. Our runtime is compared, in Table 4, against MiniSAT [36], a state-of-the-art BCP-based software SAT solver. We modified MiniSAT in two ways, in order to estimate the runtime of our hardware approach. First, we modified MiniSAT to implement a static decision strategy which is the same as the decision strategy used in our hardware engine. MiniSAT performs a smart conflict clause simplification by applying subsumption resolution [37] and caching intermediate results. So, in our second modification of MiniSAT, we disabled any simplification of the conflict clauses. This variant of MiniSAT (modified in the above two ways) is referred to as MiniSAT* in the sequel. The number of decisions made by MiniSAT* was used in computing our runtime using the above equation. Columns 2 and 3 of Table 4 list the number of decisions and the number of conflicts reported by MiniSAT. Column 4

doi: 10.1049/iet-cdt:20060221

Table 4 Comparing against MiniSAT (a BCP-based software SAT solver)

Instance	MiniSAT		MiniSAT runtime, s	MiniSAT*		Our runtime, s	Speed up
	no. of decisions	no. of conflicts		no. of decisions	no. of conflicts		
par16-3	6.26×10^3	5.98×10^3	5.68×10^{-1}	1.43×10^4	1.15×10^4	3.11×10^{-4}	1.83×10^3
ii8b4	5.70×10^2	0	6.00×10^{-3}	5.01×10^2	0	1.35×10^{-5}	4.44×10^2
am	4.64×10^7	3.95×10^7	1.26×10^4	4.62×10^9	3.64×10^9	1.24×10^2	1.02×10^2
par32-5	6.62×10^7	6.14×10^7	5.36×10^3	5.53×10^8	4.25×10^8	1.49×10^1	3.60×10^2
ii16a1	9.07×10^2	7	1.30×10^{-2}	9.70×10^2	3	2.03×10^{-5}	6.40×10^2
ii32c4	4.50×10^1	4	1.90×10^{-2}	1.50×10^2	9.90×10^1	3.15×10^{-6}	6.03×10^3
dekker	6.89×10^5	5.87×10^5	5.35×10^2	3.81×10^6	1.83×10^6	1.03×10^{-1}	5.19×10^3
frg2mul	3.24×10^6	6.07×10^5	6.21×10^2	1.57×10^8	2.09×10^7	4.24	1.47×10^2
Avg							1.84×10^3

lists the MiniSAT runtimes. The MiniSAT runtimes for these instances were obtained on a 3.6 GHz, 3 GB machine running Linux. Columns 5 and 6 list the number of decisions and the number of conflicts reported by MiniSAT*. Our estimated runtimes are reported in Column 7. The speed up obtained over MiniSAT is reported in Column 8. The average speed up over MiniSAT obtained is 1.84×10^3.

In other words, our approach yields over three orders of magnitude improvement in runtime, for the accelerated fraction of the SAT problem, over an advanced BCP-based software SAT solver. It achieves one to two orders of magnitude speedup over other hardware SAT approaches as well. Other hardware SAT approaches have significant capacity problems, making them impractical for large instances. Our approach has a large capacity and is highly scalable, and hence is ideally suited for large SAT instances.

In order to estimate the power consumption of our approach, we conducted additional SPICE simulations. These simulations were performed for computing the average power required for a single implication within a bank, and the average power required for communicating this implication to every other bank. The power consumption for the long wires (between communication units at different repeat levels), for the latter experiment was computed using layout-extracted wiring parasitics. The value obtained was $P_{\text{single}}^{\text{comm.}} = \sim 3.69\,\text{nW}$. Again assuming the worst-case situation (where each of the 25 implications/decision are on variables that repeat across banks, with a repeat level of 0), the total power required for all communications per decision (per clock cycle) is

$$P_{\text{comm.}} = P_{\text{single}}^{\text{comm}} \cdot 25 = 92.25\,\text{nW}.$$

The average power consumed by the clause bank for generating an implication, $P_{\text{single}}^{\text{imp}}$, was obtained to be about 0.363 μW. The total number of banks per IC would be at most 64 (since only six levels of hierarchy are present in the IC). In the worst case, assume that the partitions obtained from hMetis repeat a single variable v over all the 64 banks. Now suppose that there is an implication on v in every bank. For driving an implication, as explained in the previous sections, only one of the `lit` or `lit_bar` signal along with the `var_implied` signal is driven. For a conflict, on the other hand, all three signals are driven. Therefore the average power consumption for driving a single conflict literal ($P_{\text{single}}^{\text{conf}}$) is $(3/2) \cdot P_{\text{single}}^{\text{imp}}$. Since there are on average 25 implications per decision, and assuming each decision leads to a conflict involving each of the 25 implications, there are in the worst case 25 implied variables that can participate in analysing the conflict. Hence the average power for the BCP engine (which performs implication/conflict analysis) per clock cycle is

$$P_{\text{BCP}} = P_{\text{single}}^{\text{conf}} \cdot 25 \cdot \text{ Number of Banks } = 871.2\,\mu\text{W}.$$

The worst case power per cycle for our hardware SAT solver is therefore

$$P_{\text{avg}} = P_{\text{BCP}} + P_{\text{comm.}} = 871.3\,\mu\text{W}$$

doi: 10.1049/iet-cdt:20060221

Note that this low power arises from the fact that in practice, there is very little conflict activity whenever any decision is made. The majority of the clause cells do not participate in a conflict, thereby keeping the worst case power consumption low.

For the examples listed in Table 3 we compared the BCP-based software SAT runtimes with or without a limit on the number and width of the conflict clauses. The purpose of this experiment was to determine if limiting the number and width of conflict clauses significantly affects SAT runtimes. The number and width of clauses corresponded to a single row of clause banks in the centre of the chip. With this limit, we noted a negligible difference in the SAT runtimes compared with the case when there was no limit (for a timeout of one hour). Since our clause banks can be interchangeably used for conflict clause storage as well as regular clause storage, we can handle larger SAT instances by storing fewer conflict clauses in the IC.

Larger designs can be handled elegantly by our approach, since multiple SAT ICs can be connected to work cooperatively on a single large instance. A pair of such ICs would effectively implement an additional level in the inter-bank communication tree. The only wires that are shared between two such ICs are those implementing inter-bank communication. By implementing these using fast board-level IO, the system of cooperating SAT ICs can be made to operate extremely fast. The decision engine of each IC other than the root IC, behaves as a communication unit, in such a scenario.

8 Conclusion and future work

In this paper, we have presented a custom IC implementation of a hardware SAT solver and also augmented it for extracting the minimum unsatisfiable core. The speed and capacity for our SAT solver obtained are dramatically higher than those reported for existing hardware SAT engines. The speedup comes from performing the tasks of computing implications and determining conflicts in parallel, using a specially designed clause cell. Approaches to partition a SAT instance into banks and bin them into strips have been developed, resulting in a very high utilisation of clause cells. Also, through SPICE simulations we determined that the average power consumed per cycle by our SAT solver is under 1 mW, which further strengthens the practicality of our approach. Note that although we used a variant of the BCP engine of GRASP [5] in our hardware SAT solver, the hardware approach can be modified to implement other BCP engines as well. For extracting the unsatisfiable core, we implemented the approach described in [3] since our architecture naturally complements the technique proposed in [3]. Also the additional optimisations given in [3] can be seamlessly implemented in our architecture. In the future, we hope to fabricate this design to validate its performance in a real-life setting.

9 References

[1] COOK S.: 'The complexity of theorem-proving procedures'. Proc. 3rd ACM Symp. Theory of Computing, 1971, pp. 151–158

[2] PAPADIMITRIOU C.H., WOLFE D.: 'The complexity of facets resolved', *J. Comput. Syst. Sci.*, 1998, **37**, (1), pp. 2–13

[3] LYNCE J., MARQUES-SILVA J.: 'On computing minimum unsatisfiable cores'. 7th Int. Conf., Theory and Applications of Satisfiability Testing, 2004

[4] GU J., PURDOM P., FRANCO J., WAH B.: 'Algorithms for the satisfiability (SAT) problem: a survey' in 'DIMACS Series in Discrete Math. and Theoretical Computer Science', American Mathematical Society, 1997, vol. 35, pp. 19–151

[5] SILVA M., SAKALLAH J.: 'GRASP-a new search algorithm for satisfiability'. Proc. Int. Conf. Computer-Aided Design (ICCAD), November 1996, pp. 220–7

[6] MOSKEWICZ M., MADIGAN C., ZHAO Y., ZHANG L., MALIK S.: 'Chaff: Engineering an efficient SAT solver'. Proc. Design Automation Conf., July 2001

[7] GOLDBERG E., NOVIKOV Y.: 'BerkMin: a fast and robust SAT solver'. Proc., Design, Automation and Test in Europe (DATE) Conf., 2002, pp. 142–149

[8] JIN H., AWEDH M., SOMENZI F.: 'CirCUs: a satisfiability solver geared towards bounded model checking', *Comput. Aided Verif.*, 2004, pp. 519–522

[9] SKLIAROVA I., FERRARI A.: 'Reconfigurable hardware SAT solvers: a survey of systems', *IEEE Trans. Comput.*, 2004, **53**, pp. 1449–1461

[10] KAUTZ H.A., SELMAN B.: 'Planning as satisfiability'. Proc., 10th European Conf. Artificial Intelligence, 1992

[11] NAM G., SAKALLAH K.A., RUTENBAR R.A.: 'Satisfiability-based layout revisited: routing complex FPGAs via search-based Boolean SAT'. Proc. Int. Symp. FPGAs, Feburary 1999

[12] MCMILLAN K.L.: 'Interpolation and SAT-based model checking'. Proc. Computer Aided Verification, May 2003

[13] ZHAO Y., MALIK S., MOSKEWICZ M., MADIGAN C.: 'Accelerating Boolean satisfiability through application specific processing'. Proc. Int. Symp. System Synthesis (ISSS), 2001, pp. 244–249

[14] ZHAO Y., MALIK S., WANG A., MOSKEWICZ M., MADIGAN C.: 'Matching architecture to application via configurable processors: a case study with Boolean satisfiability problem'. Proc. Int. Conf. Computer Design (ICCD), September 2001, pp. 447–452

[15] ZHONG P., ASHAR P., MALIK S., MARTONOSI M.: 'Using reconfigurable computing techniques to accelerate problems in the CAD domain: a case study with Boolean satisfiability'. Proc. Design Automation Conf., June 1998, pp. 194–199

[16] ZHONG P., MARTONOSI M., ASHAR P., MALIK S.: 'Accelerating Boolean satisfiability with configurable hardware'. Proc., IEEE Symp. FPGAs for Custom Computing Machines, April 1998, pp. 186–195

[17] PAGARANI T., KOCAN F., SAAB D., ABRAHAM J.: 'Parallel and scalable architecture for solving satisfiability on reconfigurable FPGA'. Proc. IEEE Custom Integrated Circuits Conf. (CICC), May 2000, pp. 147–150

[18] ABRAMOVICI M., SAAB D.: 'Satisfiability on reconfigurable hardware'. Proc., Int. Workshop on Field Programmable Logic and Applications, 1997, pp. 448–456

[19] SUYAMA T., YOKOO M., SAWADA H., NAGOYA A.: 'Solving satisfiability problems using reconfigurable computing', *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, 2001, **9**, pp. 109–116

[20] ABRAMOVICI M., DE SOUSA J., SAAB D.: 'A massively-parallel easily-scalable satisfiability solver using reconfigurable hardware'. Proc. Design Automation Conf. (DAC), June 1999, pp. 684–690

[21] DE SOUZA J.T., ABRAMOVICI M., DA SILVA J.M.: 'A configware/software approach to SAT solving'. IEEE Symp. FPGAs for Custom Computing Machines, May 2001

[22] REIS N.A., DE SOUZA J.T.: 'On implementing a configware/software SAT solver'. Proc. 10th Annual IEEE Symp. Field-Programmable Custom Computing Machines, 2002

[23] BUNING H.K.: 'On subclasses of minimal unsatisfiable formulas', *Discrete Appl. Math.*, 2000, **107**, (1-3), pp. 83–98

[24] DAVYDOV G., DAVYDOVA I., BUNING H.K.: 'An efficient algorithm for the minimal unsatisfiability problem for a subclass of CNF', *Ann. Math. Artif. Intell.*, 1998, **23**, (3-4), pp. 229–245

[25] FLEISCHNER H., KULLMANN O., SZEIDER S.: 'Polynomial-time recognition of minimal unsatisfiable formulas with fixed clause-variable difference', *Theor. Comput. Sci.*, 2002, **289**, (1), pp. 503–516

[26] BRUNI R., SASSANO A.: 'Restoring satisfiability or maintaining unsatisfiability by finding small unsatisfiable subformulae'. In LICS Workshop Theory and Applications of Satisfiability Testing, June 2001

[27] GOLDBERG E., NOVIKOV Y.: 'Verification of proofs of unsatisfiability for CNF formulas'. Proc., Design and Test in Europe Conf., March 2003, pp. 10886–10891

[28] OH Y., MNEIMNEH M., ANDRAUS Z.S., SAKALLAH K.A., MARKOV I.L.: 'Amuse: a minimally unsatisfiable subformula extractor'. Proc., Design Automation Conf., June 2004

[29] ZHANG L., MALIK S.: 'Validating SAT solvers using an independent resolution-based checker: practical implementations and other applications'. Proc., Design and Test in Europe Conf., March 2003

[30] HUANG J.: 'MUP: a minimal unsatisfiability prover'. Proc. 10th Asia and South Pacific Design Automation Conf., January 2005

[31] WAGHMODE M., GULATI K., KHATRI S., SHI W.: 'An efficient, scalable hardware engine for Boolean satisfiability'. Proc. Int. Conf. Computer Design (ICCD), October 2006. To be Published

[32] DIMACS challenge – satisfiability. Available at: ftp://dimacs.rutgers.edu/pub/challenge/satisfiability/. The DIMACS ftp site

[33] KARYPIS G., KUMAR V.: 'A software package for partitioning unstructured graphs, partitioning meshes and computing fill-reducing orderings of sparse matrices', available at: http://www-users.cs.umn.edu/karypis/metis, September 1998

[34] The SAT'04 Competition, available at: 'http://www.lri.fr/ ∼ simon/contest04/results/'

[35] NAGEL L.: 'Spice: a computer program to simulate computer circuits,' in *University of California, Berkeley UCB/ERL Memo M520*, May 1995

[36] EÉN N., SÖRENSSON N.: The MiniSAT Page. Available at: 'http://www.cs.chalmers.se/cs/research/formalmethods/minisat/main.html'.

[37] ZHENG L., STUCKEY P.J.: 'Improving SAT using 2SAT'. ACSC '02: Proc. 25th Australasian Conf. Computer science, 2002, (Australian Computer Society, Inc. Darlinghurst, Australia), pp. 331–340

doi: 10.1049/iet-cdt:20060221