# An Efficient, Scalable Hardware Engine for Boolean SATisfiability

Mandar Waghmode† Kanupriya Gulati‡ Sunil P Khatri‡ Weiping Shi‡
† Magma Design Automation, Inc. Santa Clara, CA 95054.
‡Department of EE, Texas A&M University, College Station TX 77843.

## Abstract

Boolean Satisfiability (SAT) is a core NP-complete problem in logic synthesis. Several heuristic software and hardware approaches have been proposed to solve this problem. In this paper, we present a hardware solution to the SAT problem. *We propose a custom IC to implement our approach, in which the traversal of the implication graph as well conflict clause generation are performed in hardware, in parallel*. In our approach, clause literals are stored in specially designed cells. Clauses are implemented in banks, in a manner that allows clauses of variable width to be accommodated in these banks. To maximize the utilization of these banks, we initially partition the SAT problem. Our design is flexible in that it can implement various Boolean Constraint Propagation (BCP) engines on the same die, at the same time, allowing the user to switch BCP engines dynamically. *Our 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, effectively increasing capacity further*. Our area and performance figures are derived from layout and SPICE (using extracted parasitics) estimates. Additionally, the approach presented in this paper have been functionally validated in Verilog. Preliminary results demonstrate that our approach can accommodate instances with approximately 63K clauses on a single IC of size 1.5cm×1.5cm. *The approach results in over 4 orders of magnitude speed improvement over BCP based software SAT approaches (2-3 orders of magnitude over other hardware SAT approaches). The capacity of our approach is significantly higher than most hardware based approaches.*

## 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 $C$.

Given the broad applicability of the problem to several diverse application domains such as logic synthesis, circuit testing, pattern recognition and others [2], there has been much effort devoted to devising efficient heuristics to solve SAT. Some of the more well-known software approaches include [3, 4, 5, 6]. Again, given the general 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 [7].

In this paper, we propose an approach that utilizes a custom IC to accelerate the SAT solution process, with the **goal of speedily solving large instances in a *scalable* fashion**. By scalable, we mean that multiple SAT ICs implemented in our approach can be easily made to work in tandem on larger SAT instances. The hardware implements the GRASP [3] strategy of non-chronological backtracking. 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 maximize the hardware utilization. Experimental results are obtained using area and performance figures derived from layout and 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 heuristics in this paper.** Note that although we used the BCP engine of GRASP [3] 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 HDL and directly synthesized in our approach.

## 2 Previous Work

There have been several hardware based SAT solvers reported in the literature, which are summarized and compared in [7]. Among these approaches, [8, 9] utilize configurable processors to accelerate SAT, demonstrating a maximum speedup of 60× using a board with 121 configurable processors. The largest example mapped to this structure had 24,700 clauses. In [10, 11], the authors describes 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 [12] handles has about 1300 clauses, with an average speedup of 10×. This paper states that the hardware approaches reported in [13, 14, 15] do not handle large SAT problems.

In [16, 17], 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 [3] is between 1-2 orders of magnitude for the accelerated fraction of the SAT problem. The largest problem tackled has 214,304 clauses [17] (after conversion to 3-SAT, which can double the number of clauses [16]). In contrast, our approach performs all tasks in hardware, with a corresponding speedup of 2-3 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 utilizing 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.*

Further, the architecture of our design is designed with scalability in mind, allowing the approach to scale seamlessly to handle larger problems. The rest of this paper is organized as follows. Section 3 describes the hardware architecture employed in our approach. The generation of implications and conflicts (which is done in parallel) is explained, along with the hardware partitioning utilized, the communication protocol that banks implement, and the generation of conflict induced clauses. Section 4 describes the up-front clause partitioning methodology, which targets maximum utilization of the hardware. Section 5 reports the experimental results we have obtained, while Section 6 concludes with some directions for future work in this area.

## 3 Hardware Architecture

### 3.1 Abstract Overview

Figure 1 shows an abstracted view of our approach, in order to illustrate the main concept, and to explain how Boolean Constraint Propa-

gation (BCP) [3] is carried out. Note that the physical implementation we use is different from this abstracted view, as subsequent sections will describe. In Figure 1, the clause bank stores all clauses (a maximum of *n* clauses on *m* variables). 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. The clause bank, in turn, generates implications and possible conflicts due to this assignment. *This is done in parallel, at hardware speeds*. The decision engine accordingly assigns next variable or in case of a conflict, generates a conflict induced clause and backtracks non-chronologically [3].

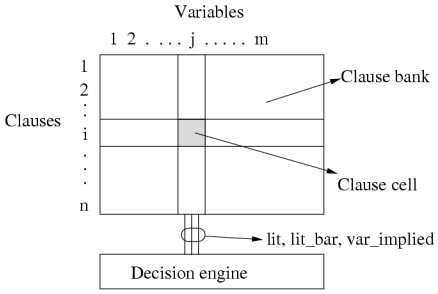



Figure 1: Abstracted view of the proposed idea.

As seen in Figure 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.

## 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 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. In practice, we do not have a single clause bank as shown in Figure 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 3.4. 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 cells, arranged in a tree fashion. The details of this inter-bank communication are provided in Section 3.5. Figure 2 describes the banks, and the inter-bank communication cells. It also shows the centrally located BCP() engine, as well as the banks for storing conflict induced clauses. The clause banks and strips will be illustrated in the sequel.

## 3.3 Clause Cell and Conflict Clause Generation

### 3.3.1 Clause Cell

Figure 3 shows the signal interface of a clause cell. Figure 4 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 *free_lit_cnt* signals serve as an indicator of number of free literals in the clause. The *imp_drv* and *cclause_drive* 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_drive* signals. The next section describes the encoding of these signals and how they are employed to perform BCP.

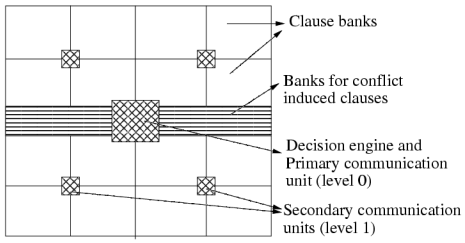



Figure 2: Generic floorplan.

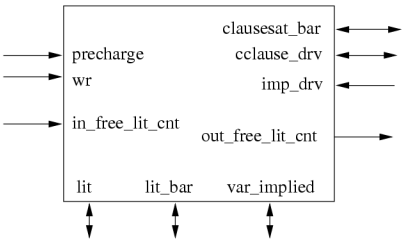



Figure 3: Signal interface of the clause cell.

### 3.3.2 Generating Implications and Conflicts

Note that the signals *lit*, *lit_bar*, *var_implied* and *cclause_drive* are predischarged and *clausesat_bar* is a precharged signal. Also, each clause cell has two registers bits 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 3.3.2. The *iamfree* signal for a variable indicates that the variable has not been assigned a value yet, nor has it been implied.

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

| Encoding | Meaning |
|---|---|
| 00 | variable does not participate in clause. |
| 01 | variable participates as a positive literal. |
| 10 | variable participates as a negative literal. |
| 11 | Illegal. |

The assignments and failure-driven assertions [3] are driven on *lit*, *lit_bar* and *var_implied* signals by the decision engine whereas implications are driven by the clause cells. 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 *lit* and *lit_bar* signals for $V_i$, then $V_i$ is said to contribute a *free literal* to clause $C_j$. 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 $c_{ij}$. In such a case, the precharged signal *clausesat_bar* for $C_j$ is pulled down by $c_{ij}$.

If clause $C_j$ has only one free literal and $C_j$ is unsatisfied, then $C_j$ is called a *unit clause* [3]. When $C_j$ becomes a unit clause with $c_{ij}$ 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 $c_{ij}$ (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

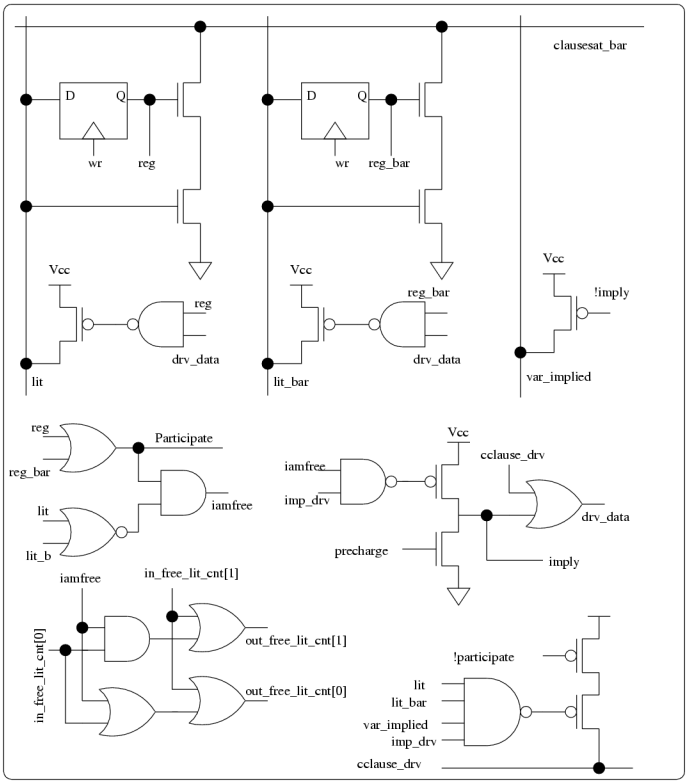



Figure 4: Schematic of the clause cell.

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 |

no free literal in the clause and the clause is unsatisfied (indicated by *clausesat_bar* staying precharged). 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. Thus, if two clauses cause different implications on a variable, both the 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.

Figure 6 shows an example CNF instance, its implication graph and how it is implicitly traversed in this scheme. $c_1 \ldots c_6$ are the clauses as shown in Figure 6(b). Let us call the *lit*, *lit_bar* and *var_implied* signals for a variable as a *signal triplet*. Initially the signals of all signal triplets are predischarged and held at high impedance. The implication graph in Figure 6(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. Figure 6(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 $e$ due to 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 i.e. $(a+b+c+\bar{p})$. 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. This makes the generation of implications and conflict clauses implicit and parallel and hence fast.

### 3.3.3 Decision Engine and Conflict Induced Clauses

Figure 5 shows the state machine of the decision engine. To begin with, the hardware is programmed with the CNF instance and all the banks' signals are precharged or predischarged as mentioned earlier. 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 and waits for the banks to compute all the implications. If no conflict is generated due to 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. Based on this information, the decision engine generates and stores the conflict induced clause. Also it non-chronologically backtracks according to the GRASP [3] algorithm. After a conflict is analyzed, the banks are again precharged and the backtracked decision is applied to the banks. 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 computations of the bank.

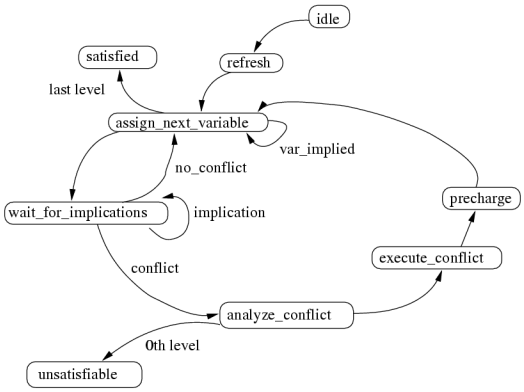
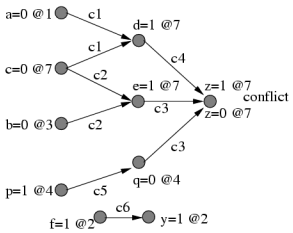



Figure 5: State diagram of the decision engine.

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 utilize 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.

The above functionality of the clause cell, along with implication, conflict generation, BCP, backtracking and decision generation has been implemented and verified in Verilog.




(a) Implication Graph

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

(b) CNF instance

| | 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) Implicit, Parallel Generation of Conflict Induced Clause

Figure 6: Example of implicit traversal of implication graph.

## 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 (Figure 1) results in a lot of non-participating clause cells. For the DIMACS [18] 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. Though a clause is fully contained in one bank, note that a variable may appear in more than one banks.

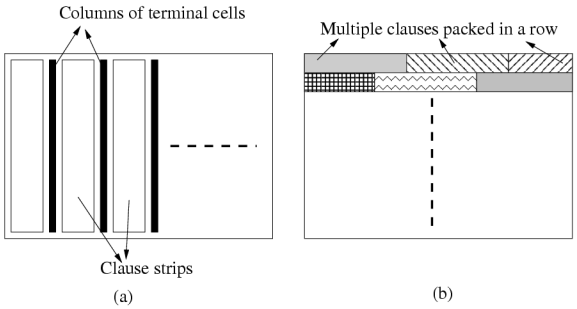



Figure 7: (a)Internal structure of a bank. (b)Multiple clauses packed in one bank-row.

Figure 7 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 minimized). We try to fit more than one clause per row with the help of strips. This is achieved by inserting a column of *terminal cells* between the strips. Please refer to Figure 8 for a 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 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 schematic view of the terminal cell is shown in Figure 8.

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*.

Figures 4 shows the schematic of our clause cell. The layout, generated in a full-custom manner, had a size of 12$\mu$m by 9$\mu$m.

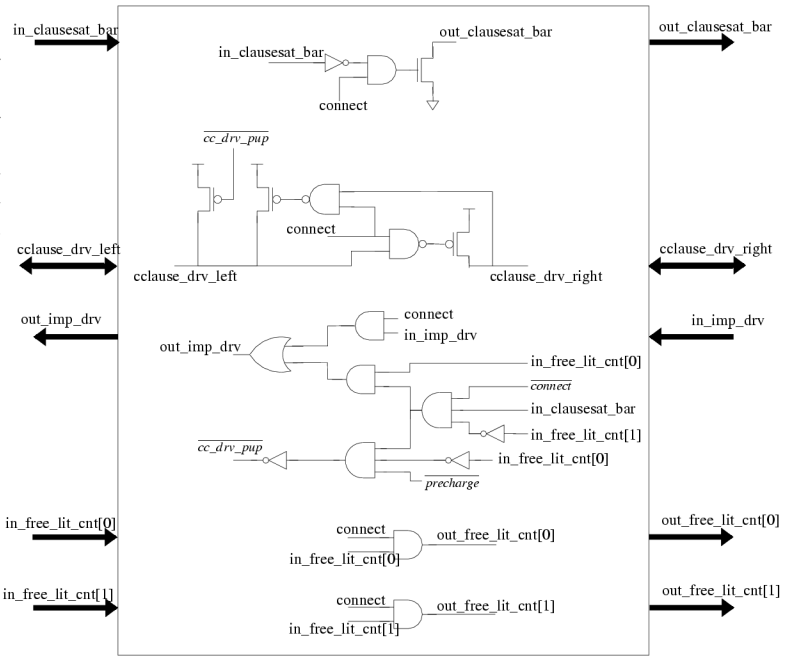



Figure 8: Schematic of a terminal cell.

The algorithm for partitioning the problem into banks, and for packing the clauses of any bank into its strips (to minimize the number of non-participating cells) is described in Section 4. Also, experimental results and optimal dimensions of the banks and strips are presented in Section 5.

## 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 Figure 9. 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 cells. As seen in Figure 9, the communication cells and base cells form a tree structure. The communication cell directly interacting with the decision engine is said to be at $0^{th}$ *level* of hierarchy and base cells are said to be at the highest level of hierarchy.

Each variable is associated with an *identification tag* in this scheme.

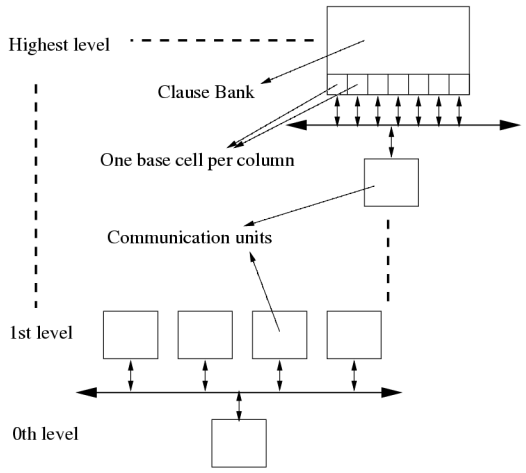



Figure 9: Hierarchical structure for inter-bank communication.

Every base cell has a 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 cell 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 cell $C$ at the next lower level of hierarchy. The communication cell $C$ communicates these data to other communication cells 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 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.

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

# 4 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 [19] to solve them.

To partition the CNF instance in 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 minimized.

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 minimize the number of rows required to fit the clauses in the bank. For this purpose, we use a 2-dimensional graph bandwidth minimization algorithm and then use a greedy bin-packing approach to pack the clauses in the rows. Figure 7 depicts this packing of multiple clauses in one row. The details of the diagonalization and greedy bin-packing algorithm are omitted from this description due to space constraints.

# 5 Experimental Results

To validate our ideas, we tested several examples from the DIMACS [18] test suite and from the SAT-2004 [20] 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 6 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 [21], using layout-extracted parasitics. The hardware SAT IC was implemented in a 0.1µm process, with a VDD of 1.2V.*

For all the examples listed in Table 3, we performed partitioning (into banks) and binning (into strips) as described in Section 4. 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.

$$\text{PF} = \frac{\text{Total \# of Cells}}{\text{\# of Participating Cells}}$$

The PF before partitioning and binning is shown in Column 4. This corresponds to the PF of a monolithic 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 10 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, as opposed to about 63,000 clauses on a single die for our approach. Further, the total run-time for the partitioning (using hMetis [19]), diagonalization and greedy bin-packing for the examples listed in Table 3 ranged from 8 to 200 seconds on a 3.6GHz, 3GB 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.*

Table 3: Partitioning and Binning Results

| Instance | #Clauses | #Vars | PF (initial) | PF (opt.) | #strips | avg #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 |
| x1mul | 55571 | 8755 | 2592 | 7.50 | 8054 | 2.15 |
| frg2mul | 62943 | 10313 | 3063 | 8.68 | 10514 | 2.41 |

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 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 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 $\Delta_i$. Then the total communication delay is estimated as

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

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

Our Runtime = Number of Decisions $\cdot \Delta_C$

Our runtime is compared, in Table 4, against MiniSAT[22], a state-of-the-art BCP based software SAT solver. The number of decisions made by MiniSAT was used in computing our runtime. The MiniSAT runtimes for these instances were obtained on a 3.6 GHz, 3GB machine running Linux. The average speed up over MiniSAT obtained is $2.29 \times 10^4$.

*In other words, our approach yields over four orders of magnitude improvement in time, for the accelerated fraction of the SAT problem, over an advanced BCP based software SAT solver, (2-3 orders of magnitude over other hardware SAT approaches), which is significantly better than the results reported for other hardware SAT engines.* Other hardware SAT approaches have significant capacity problems, making them impractical for large instances. With our high capacity and scalability, our approach is ideally suited for large SAT instances.

Table 4: Comparing Runtimes against MiniSAT BCP-based Software SAT

| Instance | MiniSAT runtime(s) | Our Runtime(s) | Speed Up |
|---|---|---|---|
| par16-3 | 0.568 | 0.169e-3 | 3362.16 |
| ii8b4 | 0.006 | 0.015e-3 | 389.86 |
| am | 12631.400 | 0.646 | 19561.99 |
| par32-5 | 5355.150 | 0.183 | 29261.85 |
| ii16a1 | 0.013 | 0.024e-3 | 530.85 |
| ii32c4 | 0.019 | 0.001e-3 | 15637.86 |
| dekker | 535.778 | 0.007 | 72634.34 |
| x1mul | 44265.500 | 0.768 | 57659.68 |
| frg2mul | 621.059 | 0.088 | 7081.45 |
| AVG | | | 22902.24 |

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 center of the chip. With this limit, we noted a negligible difference in the SAT runtimes compared to the case when there was no limit (for a timeout of 1 hour). Since our clause banks can be interchangeably used for conflict clause storage as well as regular clause storage, we can trade off the size of the SAT instance stored in the IC with the size of the conflict clause banks.

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.

# 6 Conclusion and Future Work

In this paper, we have presented a custom IC implementation of a hardware SAT solver. The speed and capacity 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 utilization of clause cells. Note that although we used the BCP engine of GRASP [3] in our hardware SAT solver, the hardware approach can be modified to implement other BCP engines as well. In the future, we plan to fabricate this design to validate its performance in a real-life setting.

# References


[1] S. Cook, "The complexity of theorem-proving procedures," in *Proceedings, Third ACM Symp. Theory of Computing*, pp. 151–158, 1971.

[2] J. Gu, P. Purdom, J. Franco, and B. Wah, "Algorithms for the satisfiability (SAT) problem : A survey," *DIMACS Series in Discrete Math. and Theoretical Computer Science*, vol. 35, pp. 19–151, 1997.

[3] M. Silva and J. Sakallah, "GRASP-a new search algorithm for satisfiability," in *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*, pp. 220–7, November 1996.

[4] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an efficient SAT solver," in *Proceedings of the Design Automation Conference*, July 2001.

[5] E. Goldberg and Y. Novikov, "BerkMin: A fast and robust SAT solver," in *Proc., Design, Automation and Test in Europe (DATE) Conference*, pp. 142–149, 2002.

[6] H. Jin, M. Awedh, and F. Somenzi, "CirCUs: A satisfiability solver geared towards bounded model checking," in *Computer Aided Verification*, pp. 519–522, 2004.

[7] I. Skliarova and A. Ferrari, "Reconfigurable hardware SAT solvers: A survey of systems," *IEEE Transactions on Computers*, vol. 53, pp. 1449–1461, November 2004.

[8] Y. Zhao, S. Malik, M. Moskewicz, and C. Madigan, "Accelerating Boolean Satisfiability through application specific processing," in *Proceedings, International Symposium on System Synthesis (ISSS)*, pp. 244–249, 2001.

[9] Y. Zhao, S. Malik, A. Wang, M. Moskewicz, and C. Madigan, "Matching architecture to application via configurable processors: A case study with Boolean Satisfiability problem," in *Proceedings, International Conference on Computer Design (ICCD)*, pp. 447–452, Sept 2001.

[10] P. Zhong, P. Ashar, S. Malik, and M. Martonosi, "Using reconfigurable computing techniques to accelerate problems in the CAD domain: A case study with Boolean Satisfiability," in *Proceedings, Design Automation Conference*, pp. 194–199, Jun 1998.

[11] P. Zhong, M. Martonosi, P. Ashar, and S. Malik, "Accelerating Boolean Satisfiability with configurable hardware," in *Proceedings, IEEE Symposium on FPGAs for Custom Computing Machines,*, pp. 186–195, April 1998.

[12] T. Pagarani, F. Kocan, D. Saab, and J. Abraham, "Parallel and scalable architecture for solving Satisfiability on reconfigurable FPGA," in *Proceedings, IEEE Custom Integrated Circuits Conference (CICC)*, pp. 147–150, May 2000.

[13] M. Abramovici and D. Saab, "Satisfiability on reconfigurable hardware," in *Proceedings, International Workshop on Field Programmable Logic and Applications*, pp. 448–456, 1997.

[14] T. Suyama, M. Yokoo, H. Sawada, and A. Nagoya, "Solving satisfiability problems using reconfigurable computing," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 9, pp. 109–116, Feb 2001.

[15] M. Abramovici, J. de Sousa, and D. Saab, "A massively-parallel easily-scalable satisfiability solver using reconfigurable hardware," in *Proceedings, Design Automation Conference (DAC)*, pp. 684–690, June 1999.

[16] J. T. de Souza, M. Abramovici, and J. M. da Silva, "A configware/software approach to sat solving," in *IEEE Symposium on FPGAs for Custom Computing Machines*, May 2001.

[17] N. A. Reis and J. T. de Souza, "On implementing a configware/software sat solver," in *Proceedings, 10 th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, 2002.

[18] "ftp://dimacs.rutgers.edu/pub/challenge/satisfiability/." The DIMACS ftp site.

[19] G. Karypis and V. Kumar, *A Software package for Partitioning Unstructured Graphs, Partitioning Meshes and Computing Fill-Reducing Orderings of Sparse Matrices.* http://www-users.cs.umn.edu/ karypis/metis, September 1998.

[20] "http://www.lri.fr/~simon/contest04/results/." The SAT'04 Competition.

[21] L. Nagel, "Spice: A computer program to simulate computer circuits," in *University of California, Berkeley UCB/ERL Memo M520*, May 1995.

[22] "http://www.cs.chalmers.se/cs/research/formalmethods/minisat/main.html." The MiniSAT Page.