The Journal of Supercomputing, 26, 109–129, 2003

Instance-Specific Accelerators for Minimum Covering

CHRISTIAN PLESSL plessl@tik.ee.ethz.ch

Computer Engineering and Networks Laboratory, Swiss Federal Institute of Technology (ETH) Zurich, 8092 Zurich, Switzerland

MARCO PLATZNER platzner@tik.ee.ethz.ch

Computer Engineering and Networks Laboratory, Swiss Federal Institute of Technology (ETH) Zurich, 8092 Zurich, Switzerland

Abstract. This paper presents the acceleration of minimum-cost covering problems by instance-specific hardware. First, we formulate the minimum-cost covering problem and discuss a branch & bound algorithm to solve it. Then we describe instance-specific hardware architectures that implement branch & bound in 3-valued logic and use reduction techniques similar to those found in software solvers. We further present prototypical accelerator implementations and a corresponding design tool flow. Our experiments reveal significant raw speedups up to five orders of magnitude for a set of smaller unate covering problems. Provided that hardware compilation times can be reduced, we conclude that instance-specific acceleration of hard minimum-cost covering problems will lead to substantial overall speedups.

Keywords: reconfigurable computing, instance-specific acceleration, minimum covering

1. Introduction

In the last years, several reconfigurable accelerators have been presented that speed up combinatorial problems. These accelerators are instance-specific i.e., they generate circuits for specific instances of these problems on the fly. It has been shown that instance-specific accelerators outperform software solvers for many instances of hard combinatorial problems, provided compilation time is kept short. Given state of the art commercial tools for circuit synthesis and optimization, instance-specific accelerators can be competitive if the runtime of the software solvers is in the order of minutes.

The best-investigated problem so far is the Boolean satisfiability problem (SAT) [9]. Given

- a set of n Boolean variables $x_1, x_2, \ldots, x_n$,
- a set of literals, consisting of variables x_i and their complements $\bar{x}_i$, and
- a set of m clauses $C_1, C_2, \ldots, C_m$, consisting of literals combined by the logical or operator $+$,

Published in: *Journal of Supercomputing*
 DOI: 10.1023/a:1024443416592

SAT quests for an assignment of truth values to the variables that makes the conjunctive normal form (CNF) $C_1 \cdot C_2 \cdot \cdots \cdot C_m$ true, where $\cdot$ denotes the logical and operator.

While most of the work published on instance-specific accelerators targets discrete decision problems such as SAT, this paper concentrates on discrete optimization problems. We present exact solvers for minimum-cost covering problems. Minimum-cost covering problems are important SAT-related optimization problems that must be solved quite frequently in engineering applications such as synthesis and optimization of digital circuits [7]. The minimum-cost covering (or set cover) problem is defined as follows:

Definition 1.1 (set cover [17]). Given a set U of elements and a set S of subsets t_i of U, $i = 1, \ldots, |S|$, find the smallest subset $T \subseteq S$ that contains all elements of U:

$$\bigcup_{t_i \in T} t_i = U. \tag{1}$$

The set cover problem is the generalization of a number of important specialized problems. One such special covering problem is the vertex covering problem: Given a graph $G = (V, E)$ with a set of vertices V and a set of edges E, we seek for the smallest set of vertices such that each edge is connected to at least one vertex in this set. In this case, U is modeled as the set of all edges and t_i as the set of edges incident to vertex v_i. Instead of referring to the sets t_i, we can also refer to the vertices v_i. Figure 1(a) shows an example with four vertices and four edges. The set $\{v_2, v_3\}$ forms a minimum-cost cover (minimum cover for short).

Minimum covering problems can be regarded as minimum-cost SAT problems. Minimum-cost SAT means to find a satisfying solution for a CNF that minimizes a linear cost function over the variables, $\mathbf{c}^T\mathbf{x}$, where $\mathbf{c}$ denotes a cost vector. A minimum covering problem is transformed to a minimum-cost SAT problem by identifying the elements of S with the binary decision variables x_i and the elements of U with the clauses of the CNF. The CNF corresponding to the vertex covering problem of Figure 1 is $(x_1 + x_2) \cdot (x_1 + x_3) \cdot (x_2 + x_3) \cdot (x_3 + x_4)$.

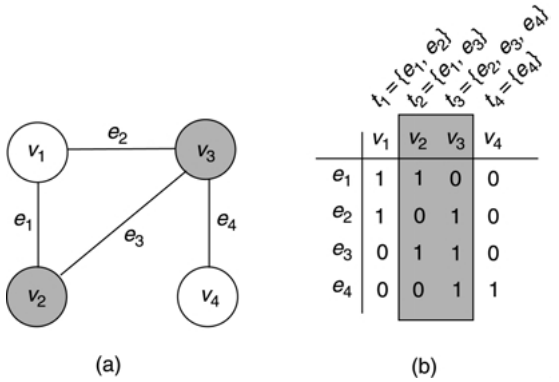

Figure 1. Vertex covering example: (a) graph, (b) matrix representation and minimum-cost cover.

Both problems can be modeled using a matrix representation. A covering problem is described by matrix $A \in \mathscr{B}^{m \times n}$, where the set of rows corresponds to the elements of $U(m = |U|)$ and the set of columns corresponds to the elements of $S(n = |S|)$. A matrix entry in row r and column c equals 1, if element e_r is a member of subset t_c. Otherwise, the matrix entry equals 0. A cover corresponds to a subset of columns having at least one 1-entry in all rows of **A**. In SAT notation, a minimum-cost cover is a selection $\mathbf{x} \in \mathscr{B}^n$ such that $\mathbf{Ax} \geq 1$ and $\mathbf{c}^T\mathbf{x}$ is minimum. Figure 1(b) shows the matrix for the vertex covering problem of Figure 1(a) together with one possible minimum cover.

CNFs corresponding to covering problems such as vertex covering are unate Boolean expressions, i.e., each variable appears either always non-inverted or always inverted. Hence, such covering problems are called unate covering. The consequence of the equation being unate is that selecting all elements of S always results in a cover, however, not necessarily in a minimum cover. Covering problems can be extended to include cases where the selection of an element a of S implies the selection of another element b of S. Then a clause of the form $(\overline{a} + b)$ has to be added. This makes the CNF binate since some variables appear now in both non-inverted and inverted form. Such covering problems (covering with implications) are called binate covering. They can still be represented using matrices where rows including negated variables have a -1 entry in the corresponding column. A valid cover corresponds to selecting a subset of columns such that all rows have at least a 1 entry in that subset or a -1 entry in the complementary subset.

This paper is organized as follows: Section 2 surveys related work in instance-specific acceleration of combinatorial problems. In Section 3, we discuss exact software solvers for unate covering. In Section 4, we present the architectures of several instance-specific accelerators for exact unate covering. Section 5 discusses a design tool flow and empirical results derived from simulation and prototype implementation. Section 6 concludes the paper and lists lines of research for further work.

2. Related work

The brief overview of related work in instance-specific accelerators is grouped into architectures that solve SAT exactly, architectures that solve SAT by incomplete stochastic local search procedures, and architectures for problems other than SAT.

2.1. Exact SAT

In 1996, Yokoo et al. [20] proposed an instance-specific SAT accelerator that models variables in standard 2-valued logic and uses a forward checking technique to find a satisfying value assignment. In their later work, the authors devised a backtracking algorithm and experimented with variable ordering heuristics [18]. Zhong et al. [22] were the first to propose a reconfigurable accelerator that implements backtracking with Boolean constraint propagation [6] as basic

deduction strategy. The authors further suggested two extensions to their architecture with conflict analysis techniques that allow for non-chronological backtracking and dynamic clause addition [23]. Platzner and De Micheli [14] presented several SAT architectures based on backtracking with 3-valued logic. They used different deduction techniques, among them don't care variables and implications. A custom compilation framework for these SAT accelerators was discussed by Mencer et al. [13]. Abramovici and De Sousa [1] and Abramovici and Saab [2] presented an architecture based on the PODEM algorithm used for automatic test pattern generation. Their architecture uses backtracing rather than backtracking and propagates the required result of a Boolean formula back to the variables.

2.2. *Stochastic local search for SAT*

Stochastic local search procedures are incomplete algorithms used to solve large instances of the SAT and MAX-SAT problems. MAX-SAT is an optimization problem that tries to maximize the number of satisfied clauses of a CNF. There are two important families of stochastic local search procedures for SAT: GSAT and WSAT. GSAT starts with a random full value assignment and iteratively flips the values of variables in order to increase the number of satisfied clauses. The selection of variables to flip is based on the score of a variable. The score of a variable is the difference between the number of clauses unsatisfied by the current assignment and the assignment after flipping the variable. GSAT flips the variable with maximal score in one search step. If there are several variables with maximal score, one is randomly chosen. Hamadi and Merceron [11] proposed a reconfigurable architecture that implements an adaption of GSAT. An FPGA implementation of GSAT was presented by Yung et al. [21]. In contrast to GSAT, WSAT uses a two-stage procedure to select variables. In the first stage, a currently unsatisfied clause is randomly picked. In the second stage, one of the variables in this clause is randomly picked and flipped. Leong et al. discuss the implementation of a WSAT accelerator in Leong et al. [12].

2.3. *Other combinatorial problems*

Babb et al. [3] presented reconfigurable architectures for computing the transitive closure and the shortest path in graphs. They map variants of the Bellman-Ford algorithm to instance-specific hardware. A recent architecture that also aims at accelerating the Bellmann-Ford algorithm was given by Dandalis et al. [5]. Other authors have discussed reconfigurable accelerators for constraint satisfaction problems [10, 16].

3. Unate covering

3.1. Branch & bound

Covering problems are usually solved with general search procedures such as branch & bound. Branch & bound constructs a search tree by iteratively picking a variable and branching on it, i.e., generating two subtrees with the variable set to 1 and 0, respectively. When a variable is assigned a value, deduction techniques are used to infer knowledge from the partial assignment and to reduce the problem. For example, we could conclude that the partial assignment already led to a solution or that we have run into a contradiction, i.e., the current path cannot lead to a solution. In case of a contradiction, backtracking is performed. When there are no more reductions possible, a cost bound is computed. The cost bound is the sum of two components: The cost of the current partial assignment plus an estimate for the cost of the minimum number of columns required to cover the remaining rows. When this cost bound exceeds the cost of the current best solution, the whole path can be pruned off. Otherwise, the algorithm selects the next variable and branches on it.

It has been shown that tight cost bounds are crucial for the performance of covering solvers. As the computation of the optimal cost bound is a hard problem itself, efficient heuristics are applied [4, 8]. In the worst case, branch & bound shows exponential complexity. However, for many applications reductions and bounds are quite effective.

3.2. Reductions

In the following we focus on unate covering problems with unit cost. In unit cost problems, each variable has an associated cost value of 1. Such covering problems are frequently solved in exact two-level logic minimization, for example by the ESPRESSO-EXACT algorithm [15]. The covering problem is defined by the covering matrix **A**, where the rows of **A** represent the elements of the set to be covered and the columns of **A** correspond to the groups of elements that can be selected. Several reduction rules can be applied to simplify the matrix **A**. The most-prominent reduction rules are:

- *Essential columns.* An essential column is a column having the only 1 entry of some row. An essential column must be part of any cover because selecting it is the only way to cover the corresponding row.
- *Dominated columns.* A column a dominates a column b if the entries of a are larger or equal than the entries of b. Dominated columns can be discarded from consideration, because selecting the dominating column instead covers more rows.
- *Dominant rows.* A row r dominates a row s if the entries of r are larger or equal than the entries of s. Dominant rows can be discarded, because any cover of the dominated row is also a cover of the dominant row.

3.3. *Algorithm*

Algorithm 1 shows a recursive implementation of an exact unate covering algorithm. Matrix **A** defines the covering problem, the vector **v** represents the current variable assignment, and **b** is the lowest cost solution found so far. The algorithm starts with (**A**, **0**, **1**). Line 2 iteratively reduces the matrix by applying a set of reduction rules. If no more reductions are possible, a cost bound is computed in line 3. The algorithm returns the current best cost vector if the cost bound exceeds the current best cost. If the current best cost is not exceeded and **A** contains no more rows, the current assignment forms a new solution with best cost. Otherwise, branching is required. The algorithm selects a variable v_x and assigns it to 1 (line 10) and 0 (line 16), respectively. In both cases, the matrix is modified and the exact covering procedure is called recursively.

4. Covering in hardware

4.1. *Accelerator architectures*

The basic architecture of our covering accelerators is shown in Figure 2. The architecture is divided into four blocks: state machines (sm), checkers, cost counter and controller. The state machines are arranged in a linear array, where each state machine implements one search level of the branch & bound algorithm. A state machine connects only to its immediate neighbor state machines above and below the checkers, and the controller. The output of the state machines are the current variable values that are fed into the checkers. The checkers deduce information from a partial variable assignment. There are checkers for the result of the CNF (solution found) and for various reductions such as don't cares, essentials, and dominated columns. The cost counter computes the cost of the current partial assignment. The controller initializes and stops the search procedure and computes the cost bound, i.e., decides for each assignment whether to continue search on the current path or not.

The basic backtracking mechanism is similar to the one described in Platzner and Micheli [14] and uses 3-valued logic to model the variables, the clauses, and the CNF. Values can be in $\{0, 1, X\}$, X denoting an unassigned variable, which allows to analyze partial assignments. At the beginning, all variables are X. After each value assignment, the CNF checker inspects the CNF result. If the CNF is 0, then we have identified a contradiction and backtrack. If the result is 1, a new valid cover has been found. If this cover has the least cost so far, the corresponding variable assignment is saved. In either case the accelerator backtracks to continue search on another path of the search tree. If the CNF is X, we proceed with the search on the current path. The searching procedure is started by the controller that triggers the first state machine. Depending on the checker results and the result from evaluating the bound condition, a state machine changes its assignment (continue search with different value), triggers the next state machine (continue search by branching), or triggers the previous state machine (backtrack due to bound or reduction).

Algorithm 1 Exact covering algorithm [7]

```
 1: exact_cover (A, v, b) {
 2: reduce matrix A and update v
 3: if (cost_bound (A, v) ≥ |b|) then
 4:   return (b)
 5: end if
 6: if (A has no more rows) then
 7:   return (v)
 8: end if
 9: select a branching variable v_x
10: v_x ← 1
11: A' ← A after deleting column x and incident rows
12: v' ← exact_cover (A', v, b)
13: if (|v'| < |b|) then
14:    b ← v'
15: end if
16: v_x ← 0
17: A' ← A after deleting column x
18: v' ← exact_cover (A', v, b)
19: if (|v'| < |b|) then
20:    b ← v'
21: end if
22: return (b)
23: }
```

4.2. *Checker modules*

4.2.1. CNF checker. The CNF checker takes the current variable assignment as input vector and computes the result of the CNF in 3-valued logic. Each clause is evaluated individually and the results of the clauses are fed into a wide AND-gate, see Figure 3(a).

4.2.2. Reductions. We have designed checkers for the reduction techniques don't cares, essential columns, and dominated columns. The results of the checkers are functions of the current variable assignment and not of previous variable states. Thus the checkers can be implemented in pure combinational logic. The checkers' logic is derived from the CNF during the circuit synthesis step which is part of the accelerator's runtime. Figure 3 shows the structure of the checkers. For each reduction rule, there is a dedicated checker module that takes the current variable assignment vector as input, evaluates the reduction rule for all variables in parallel, and outputs a vector of predicates indicating whether the respective conditions apply under the current variable assignment. The output of the reduction checkers is in 2-valued Boolean logic. In the

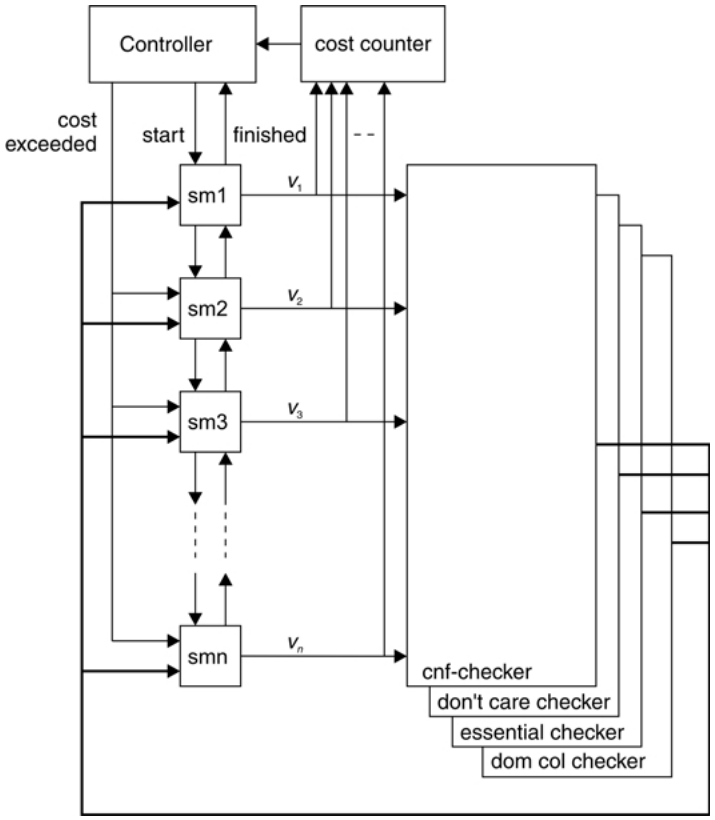

Figure 2. Architecture of the covering accelerator.

following, the reduction techniques are explained on the example of the covering matrix shown in Figure 1(b) which corresponds to the CNF $(v_1 + v_2) \cdot (v_1 + v_3) \cdot (v_2 + v_3) \cdot (v_3 + v_4)$.

- *don't cares.* A don't care variable is a variable that has no influence on the result of the CNF. Hence, it can be set to 0 to minimize cost. For example, when v_3 in Figure 1(b) is set to 1, v_4 becomes a don't care variable. The don't care condition for v_3 is $(v_1 \cdot v_2 \cdot v_4)$. The don't care condition for a variable v_s can be derived from the clause values c_i of the CNF and the covering matrix A:

$$dc_{v_s} = \bigwedge_{i=1}^{m} c'_i; \qquad c'_i = \begin{cases} c_i, & \text{if } A[i,s] = 1 \\ 1, & \text{else.} \end{cases} \tag{2}$$

  The don't care checker operates directly on the clause values rather than on the variable values, see Figure 3(b). As the CNF checker already computes the 3-valued clause values c_i, our implementation of the don't care checker reuses these intermediate signals.

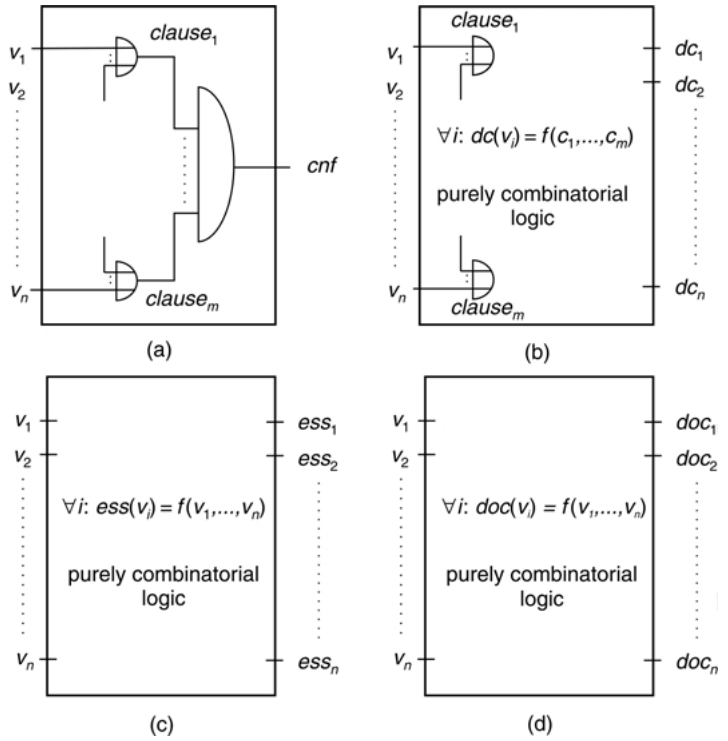

Figure 3. Structure of the different checker modules: (a) CNF checker, (b) don't care checker, (c) essential checker, (d) dominated column checker.

- *essential columns.* Essential columns correspond directly to implications in SAT problems. Generally, an implied variable may imply other variables in turn. For unate covering problems, however, the situation is simpler as an implied variable cannot imply another variable. For example, v_3 set to 0 in Figure 1(b) implies that v_4 is set to 1 as this is the only way to cover row e_4. The essential checker module generates an essential condition for each variable. For example, the essential condition for v_2 is $(\bar{v}_1 + \bar{v}_3)$. Figure 3(c) show a schematic view of the essential checker.

  Algorithm 2 presents a pseudo code for the generation of the essential logic for a single variable v_s. The procedure returns a string with the expression ess_{v_s}. The operator | denotes concatenation. For a matrix with n variables and m clauses, the procedure is called n times and runs overall in $O(mn^2)$.
- *dominated columns.* A variable that corresponds to a dominated column can be set to 0. In the matrix of Figure 1(b), column v_2 is dominated by v_1 if row e_3 is covered and by v_3 if row e_1 is covered. This results in the dominated column condition $doc_{v_2} = (v_1 + v_3)$. The dominated column checker module generates a condition for each variable. Figure 3(d) shows a schematic view of the checker.

Algorithm 2 Generation of an essential condition

Require: Matrix $A[i,j]$, variable index s

Ensure: essential logic ess_{vs}

```
1: sum ← "0"
2: for (i = 1 to m) do
3:   if (A[i,s] = 1) then
4:     product ← "1"
5:     for (k = 1 to n; k ≠ s) do
6:       if A[i,k] = 1 then
7:         product ← product | "·" v̄_k"
8:       end if
9:     end for
10:    sum ← sum | "+" | product
11:  end if
12: end for
13: return (sum)
```

To derive a generator procedure for the dominated column logic, we start with considering two variables and four rows of the matrix, and then generalize to m rows and n variables. Figure 4(a) shows two columns v_i and v_j with four rows $c_1, \ldots, c_4$ that represent all possible pairs of matrix entries. When the search procedure starts, all variables will be initialized to X. During search, variables will be set to 1 and 0, or again relaxed to X. The row values are computed in the CNF checker and are initially also X. A row value of 0 means that the corresponding clause is contradicted and thus the CNF is contradicted. In such a case, backtracking occurs and some of the current value assignments will be changed. Hence, in order to determine whether and which column dominates, we need to consider only the values X and 1 for the rows. Figure 4(b) shows the possible value combinations for the rows (clauses) of Figure 4(a). The table entries indicate the dominated columns. An entry i or j means that columns v_i and v_j are dominated, respectively. An entry i^* states that column v_i

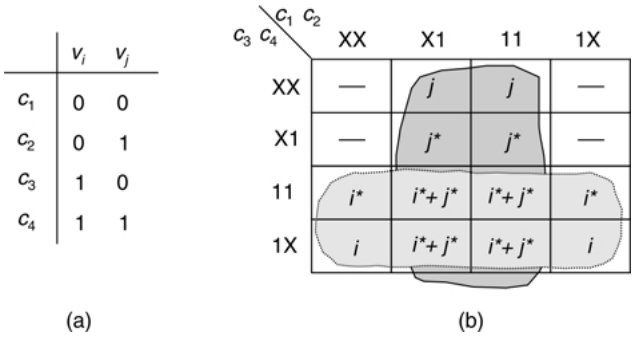

Figure 4. Possible row values for two columns and Karnaugh map for the dominated column condition.

is dominated and, additionally, a don't care. In this case we are free to choose whether this combination is included in the dominated column logic for v_i. The entry $i^* + j^*$ denotes two columns that would dominate each other. It can be shown that in the presence of don't care and essential checkers both columns are already determined. Therefore, we can choose to cover none, one, or both of them. Finally,—indicates that no column is dominated.

From Figure 4(b) we derive functions $doc_{(v_i,v_j)}$ and $doc_{(v_j,v_i)}$ that indicate whether v_j dominates v_i and vice versa:

$$doc_{(v_i,v_j)} \leftarrow (c_3 = 1) \wedge (v_j \neq 0), \tag{3}$$

$$doc_{(v_j,v_i)} \leftarrow (c_2 = 1) \wedge (v_i \neq 0). \tag{4}$$

In our architecture, a dominated column condition is evaluated only when the state machine for the variable (column) is activated. Hence, we have to check for the case where a variable causing a dominated column condition is assigned to 0. Such an assignment breaks the column dominance. Consequently, above conditions include terms of type $(v_i \neq 0)$. At this point our reduction is weaker than column dominance in software, where a dominated column is simply deleted from the covering matrix.

The scheme is now generalized to m rows. There are four possible pairs of entries in the matrix: (0, 0), (1, 1),

- (0, 1) with k appearances in rows $c_{\alpha_1}, \ldots, c_{\alpha_k}$
- (1, 0) with l appearances in rows $c_{\beta_1}, \ldots, c_{\beta_l}$.

If $\neg((k \geq 1) \wedge (l \geq 1))$ is true, one of the columns is clearly dominated. If $k = 0$, v_j is dominated; if $l = 0$, v_i is dominated. If both k and l are zero, one of the columns is redundant. The matrix is reduced at compile time until all pairs of columns have both k and l larger than zero. Then the dominated column logic is derived as follows: For v_j to dominate v_i, all the l pairs (1, 0) must have been covered; for v_i to dominate v_j, all k pairs (0, 1) must have been covered:

$$doc_{(v_i,v_j)} \leftarrow \left(\bigwedge_{t=1}^{l} (c_{\beta_t} = 1) \right) \wedge (v_j \neq 0), \tag{5}$$

$$doc_{(v_j,v_i)} \leftarrow \left(\bigwedge_{s=1}^{k} (c_{\alpha_s} = 1) \right) \wedge (v_i \neq 0). \tag{6}$$

Figure 3 shows a pseudo code for the generation of the dominated column conditions for a pair of variables. For a matrix with n columns there exist $n - 1$ dominated column conditions for each variable. A column is dominated, if any of these conditions applies:

$$doc_{v_i} \leftarrow \bigvee_{j=1; j \neq i}^{n} doc_{(v_i,v_j)}. \tag{7}$$

Overall, there are $(n/2)$ pairs of columns for which dominated column conditions must be generated, which takes $O(mn^2)$ time.

4.3. Cost counter

For evaluating the bounding condition, the cost of the current variable assignment is required. In covering problems with unit cost, the cost of a variable assignment is defined by the number of variables that are assigned to 1. Since a new cost bound must be computed after every single variable assignment, i.e., potentially every cycle, an efficient implementation of the cost counter is crucial for the overall performance of the accelerator. To compute the current cost, we use an n-bit-parallel counter structure. Such a counter is basically an adder that sums up n single bit inputs. A tree-style implementation for parallel counters was proposed by Swartzlander in Swartzlander [19]. Figure 5 shows the implementation of a 15-bit parallel counter.

Algorithm 3 Generation of a dominated column condition

Require: Matrix $A[i, j]$, variable indices s and t

Ensure: dominated column logic $doc_{(v_s,v_t)}, doc_{(v_t,v_s)}$
1: $lproduct \leftarrow$ "1"
2: $kproduct \leftarrow$ "1"
3: **for** $(i = 1$ **to** $m)$ **do**
4: **if** $((A[i,s] = 1) \wedge (A[i,t] = 0))$ **then**
5: $lproduct \leftarrow lproduct \mid$ "$\cdot (c_i = 1)$"
6: **else if** $((A[i,s] = 0) \wedge (A[i,t] = 1))$ **then**
7: $kproduct \leftarrow kproduct \mid$ "$\cdot (c_i = 1)$"
8: **end if**
9: **end for**
10: $doc_{(v_s,v_t)} \leftarrow lprod \mid$ "$\cdot (v_t \neq 0)$"
11: $doc_{(v_t,v_s)} \leftarrow kprod \mid$ "$\cdot (v_s \neq 0)$"
12: **return** $(doc_{(v_s,v_t)}, doc_{(v_t,v_s)})$

The counter is built from full-adders (FA) organized in stages. The first stage consists of 1-bit full-adders with outputs fed to the 2-bit full-adders in the next stage, and so forth. The regular structure of the adders allows for an efficient FPGA implementation, using the fast-carry chain routing resources. A counter with n input bits results in $l = \lfloor \log_2 (n) \rfloor$ levels. Hence the delay of an n bit parallel counter is bounded by:

$$\tau_{\text{n-bit-ctr}} = \frac{l(l+1)}{2} \cdot \tau_{\text{adder}},$$

where τ_{adder} is the delay of a 1-bit full-adder. For fast implementations of parallel counters, faster adders such as carry-look-ahead or carry-select adders may be used.

4.4. Bound

Currently, our architecture uses current_best_cost -1 as cost bound. There is no estimation of the cost added by variables not yet searched. While this leads to a

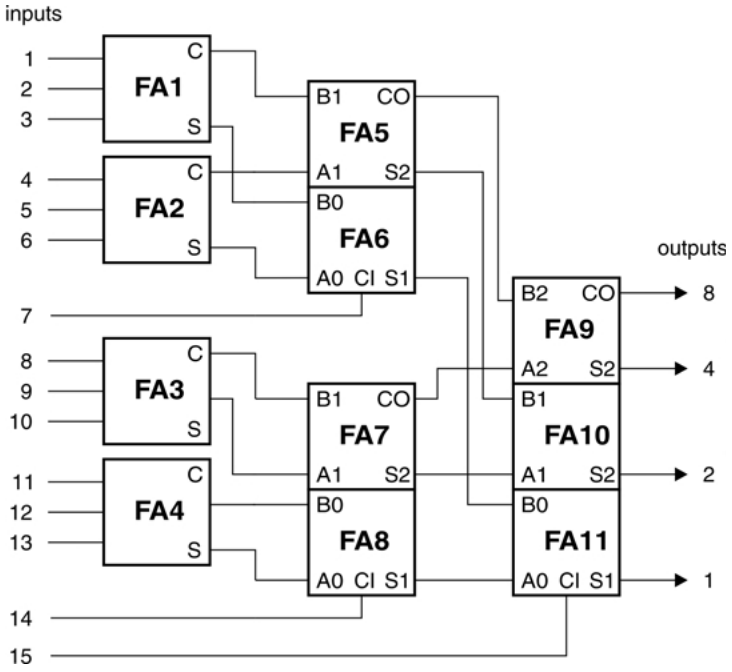

Figure 5. Architecture of a 15-bit cost counter.

straight-forward hardware implementation, it lacks the sophisticated bounding techniques applied in most software solvers.

4.5. Statemachines

As shown in Figure 2, our covering solvers include a linear array of identical state machines. Figure 6 shows the inputs/outputs and the state transition diagram for a state machine. The state machine is connected to the state machine above by the signals *FT* (from top) and *TT* (to top) and to the state machine below by the signals *TB* (to bottom) and *FB* (from bottom). This is the only way for the state machines to interact. The results from the checkers are fed back to the state machines via *ST0* (set to zero), in case the variable is don't care or the column is dominated, and *ST1* (set to one), in case the variable is essential. The 3-valued result of the CNF is read via the *cnf* input. *CEX* (cost exceeded) indicates that the cost bounding condition has occurred.

Each state machine controls the value of its associated variable. At initialization, the state machine assigns *X* to its variable. This value remains until the state machine is either triggered from above via *FT* or the variable becomes essential as result of other variable assignments. If the state machine is triggered from above and the variable is neither don't care nor dominated, the state machine assigns 1 to its variable. This results in a CNF value of either 1 (satisfying assignment) or *X*. In the

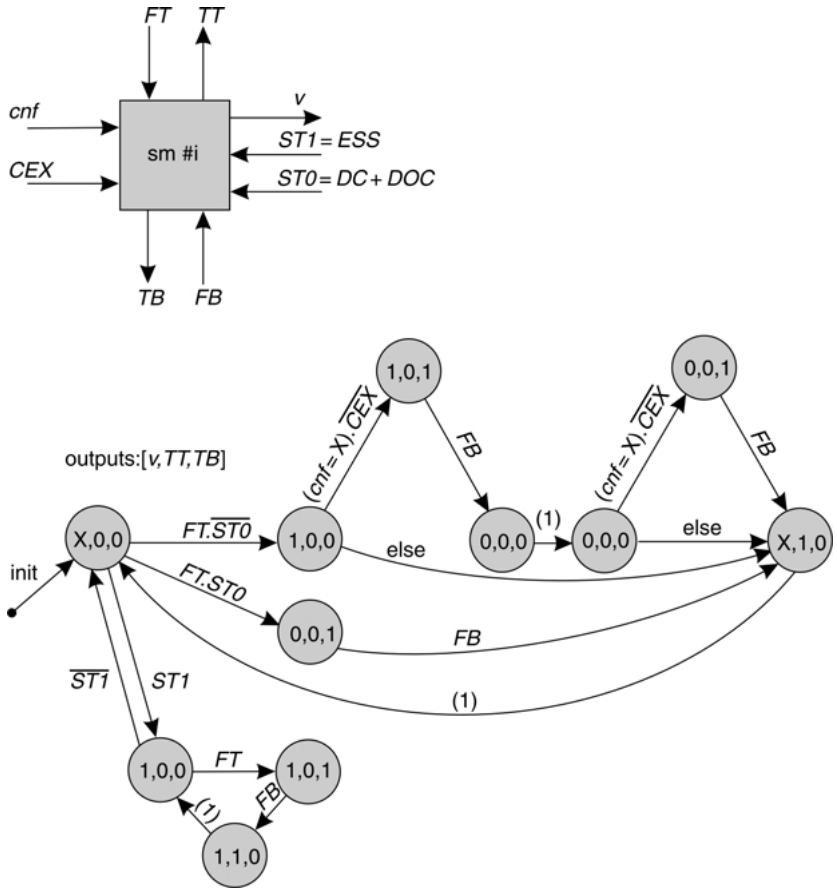

Figure 6. Statemachine input/outputs and state transition diagram for the covering accelerator using don't cares, essentials and dominated column reductions.

latter case, the next state machine is triggered and the control moves from the current state machine to the next. This is the equivalent to a branch on 1 in software. Then the current state machine waits for the signal *FB*, which indicates that the state machine below is initiating backtracking. The state machine now assigns 0 to the variable (branch on 0). At this point a wait state is inserted to allow the checkers to compute new results. This is required as deasserting a variable might remove essential conditions which in turn cause don't cares and dominated columns. After the wait state, the state machine checks the CNF again. If there is no contradiction and the cost is not exceeded, search continues by triggering the next state machine. In all other cases, backtracking is performed by signaling *TT* to the state machine above.

The don't care and dominated column reductions are handled as soon as the state machine is activated (*FT*). In this case, the state machine directly assigns 0 to its variable and passes control to the next state machine by signaling *TB*. If a variable is

essential, the corresponding state machine immediately assigns 1, even without being triggered by *FT*. If the state machine is triggered later on, is passes control directly to the next state machine (*TB*) and waits for the signal (*FB*). An essential value remains as long as the essential condition applies.

4.6. Adaption to other covering problems

An important design goal was to make the architecture extensible and adaptable to different covering problems that require different algorithms for their reductions and for calculating bounds. We support this by encapsulating the actual reductions in checker modules and the bound calculation in the controller. The backbone of the architecture is the array of state machines that implements backtracking search in 3-valued logic.

Binate covering problems require slightly different reduction techniques. While don't cares apply to both unate and binate problems, essentials differ as in binate problems implied variables can in turn imply other variables. Moreover, covering problems can come as unit cost problems where all variables have the same cost value of 1, or as integer cost problems where each variable has a cost value in $\mathcal{N}$. Our architecture can easily be adapted to integer cost by replacing the cost counter with a cost adder.

5. Experiments

To evaluate the different accelerator architectures, we have implemented a design tool flow that allows us to derive experimental performance data by simulation and prototypical implementation. Figure 7 illustrates the overall design tool flow. A problem instance is described in text form according to the DIMACS CNF file format—a standard format for SAT problems. From this file, a Perl program generates the instance-specific VHDL code. The Perl program uses VHDL code templates describing the non-instance specific parts of the architecture and augments this with generated code for the instance-specific parts, e.g., the various checkers. The generated VHDL code is used for both simulation and implementation. The implementation path employs Synopsys FPGA Compiler II for synthesis and the Xilinx 4.1i back-end tools for place, route, and configuration bitstream generation. Then, the bitstream is downloaded onto the FPGA in our prototyping hardware and the accelerator circuit is started. The algorithm completes when the top-most state machine signals the controller (see Figure 2). The monitor, a host software routine, polls the controller for completion. When the accelerator has completed, the host reads back the result.

For testing and benchmarking, we have chosen 16 small and 5 medium sized problems from the ESPRESSO distribution. In order to obtain the benchmarks, we have instrumented ESPRESSO to output the *cyclic cores*, i.e., the covering matrices just before the first branch and after the first round of reductions. The resulting test problems have from 4 to 62 variables and from 4 to 70 clauses.

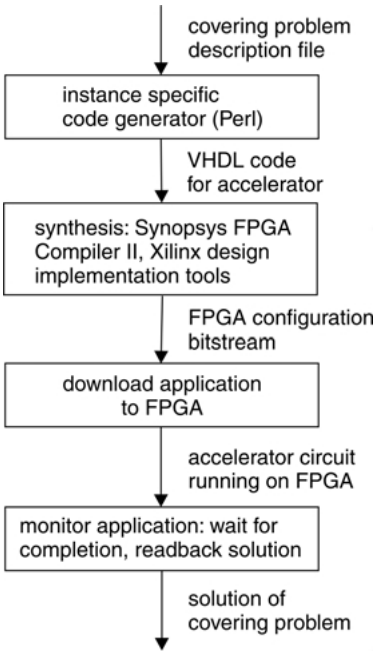

Figure 7. Tool flow for the covering accelerator.

5.1. Simulation

The 21 test problems have been simulated using the Modelsim VHDL simulator. Table 1 shows the detailed results of the simulation. Each benchmark is described by the number of clauses and variables and the cost of the optimal solution. The number of cycles reported corresponds to an accelerator architecture using don't care, essential columns and dominated column reductions and the state machine shown in Figure 6. The right-most columns of Table 1 present the hardware execution time, the software execution time and the resulting raw speedup.

The raw speedup has been calculated as $S_{raw} = t_{\mathrm{sw}}/t_{\mathrm{hw}}$ and does not include the hardware compilation time. The software execution time was determined by measuring the runtime of ESPRESSO for covering the cyclic cores on a Sun Ultra10 440 MHz workstation. The ESPRESSO execution times for our benchmarks range from 5 ms to 1.3 s. The hardware execution time has been calculated by multiplying the number of cycles with the clock period of an assumed clock rate of 25 MHz.

Figure 8 shows the raw speedups for the benchmarks in logarithmic scale. Figure 9 summarizes these results in a speedup histogram, grouping the problems into speedup classes of one order of magnitude. For one out of the 21 problems no raw speedup was achieved at all, the other problems show speedups of several orders of magnitude. The majority of benchmarks reveal a raw speedup of 1,000–10,000. The architecture investigated in this simulation uses don't cares, essentials and dominated

Table 1. Results from VHDL simulation of 21 EPSRESSO benchmarks

Benchmark	Vars	Clauses	Cost	Cycles	hw @ 25 mhz	sw espresso [s]	Raw speedup [s]
amd	27	25	12	2,673	1.07E − 04	1.00E − 02	9.35E + 01
b10	6	6	3	19	7.60E − 07	1.00E − 02	1.32E + 04
b3	9	10	4	33	1.32E − 06	2.00E − 02	1.52E + 04
bc0	8	8	4	41	1.64E − 06	4.40E − 01	2.68E + 05
bcc	12	12	6	94	3.76E − 06	5.00E − 03	1.33E + 03
bench	8	8	4	46	1.84E − 06	1.00E − 02	5.43E + 03
chkn	6	6	3	26	1.04E − 06	5.00E − 03	4.81E + 03
dekoder	7	7	4	46	1.84E − 06	5.00E − 03	2.72E + 03
dk27	8	8	4	39	1.56E − 06	5.00E − 03	3.21E + 03
f51m	14	15	7	172	6.88E − 06	2.00E − 02	2.91E + 03
m181	20	27	9	246	9.84E − 06	7.00E − 02	7.11E + 03
mark1	4	4	2	12	4.80E − 07	5.00E − 03	1.04E + 03
newill	6	6	3	25	1.00E − 06	5.00E − 03	5.00E + 03
root	14	16	7	112	4.48E − 06	1.00E − 02	2.23E + 03
tms	9	9	5	60	2.40E − 06	5.00E − 03	2.08E + 03
wim	7	7	4	46	1.84E − 06	5.00E − 03	2.72E + 03
dist	45	41	20	37,615	1.50E − 03	1.00E − 02	6.65E + 00
in2	38	42	17	8,800	3.52E − 04	5.00E − 03	1.42E + 01
intb	58	62	28	142,357	5.69E − 03	1.32E − 00	2.32E + 02
l8err	62	70	21	571,785	2.29E − 02	1.00E − 02	4.37E − 01
m3	42	39	19	8,115	3.25E − 04	2.00E − 02	6.16E + 01

columns as reduction rules. In contrast, the software covering algorithm uses more reduction techniques and much improved lower bounds. Depending on the problem instance, the sophisticated software methods either drastically improve the performance or just add a large overhead.

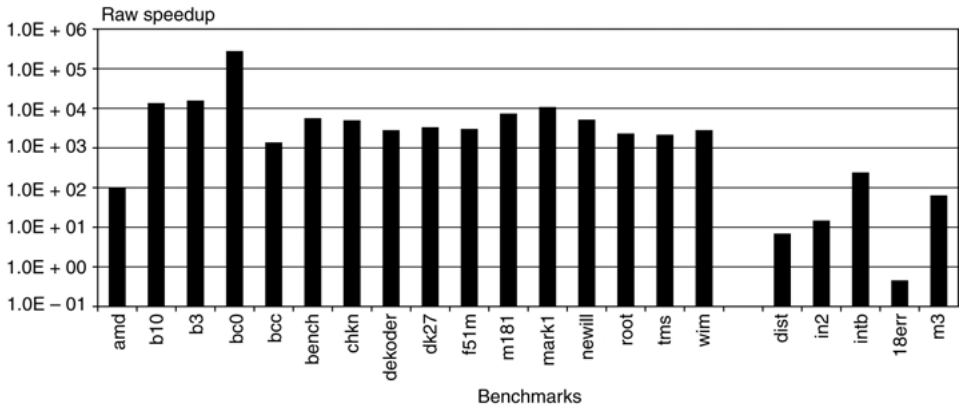

Figure 8. Raw speedup for the covering accelerator compared to ESPRESSO.

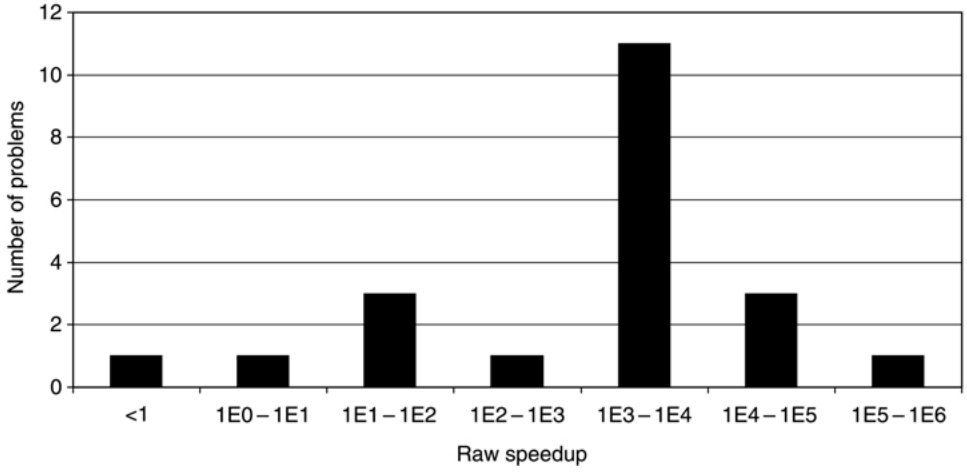

Figure 9. Raw speedup histogram.

5.2. *Implementation*

The accelerator circuits have been implemented on a prototyping platform consisting of a PC with a PCI carrier-board SMT320 and an FPGA module SMT358, both from Sundance (www.sundance.com). The FPGA module is equipped with a Xilinx Virtex XCV1000-BG560-4 device. The smallest configurable unit of a Virtex FPGA is a slice. The XCV1000 device provides 12,288 slices.

The results show that the benchmark circuits achieve clock rates of 30–50 MHz. This justifies the assumption of a 25 MHz clock rate, which formed the basis for the speedup figures. Synthesis and design implementation tools run on a Sun Ultra10 440 MHz workstation with 512 MB RAM. Without any optimizations and design constraints, the times required to map from the problem description to the accelerator bitstream are in the order of minutes. For the benchmark amd, which has 27 variables and 25 clauses, code generation takes 4 seconds, circuit synthesis 160 seconds, and place and route 360 seconds. The time used for FPGA configuration and readback of the solution is negligible. The circuit for the amd benchmark required an area of 1,072 slices, which amounts to 8% of the FPGA. Obviously, the long synthesis times render hardware acceleration of our small test problems useless. However, the goal for the accelerator are hard and large problems with long software runtimes. Further, the accelerators will greatly benefit from techniques that reduce hardware compilation times.

To evaluate the influence of the covering solver architecture, i.e., the number of used checkers, on the performance and hardware cost, we have implemented several accelerator versions for three larger benchmarks. For each benchmark, we have implemented following four variants: CNF checker only (CE), CNF checker and don't care checker (CEDC), CNF checker and don't care and essential checker (CEDCES), and the most sophisticated architecture including CNF, don't care, essential and dominated column checker (CEDCESDCOL). The state machines of

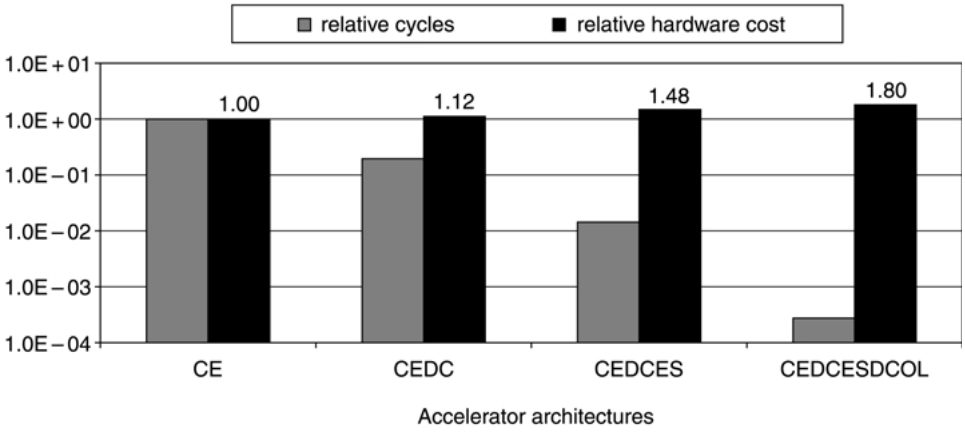

Figure 10. Relative performance and hardware cost for different accelerator architectures.

the accelerators have been simplified according to the actually used checkers. Figure 10 shows the relative size and performance of the different covering solvers, normalized to the CE architecture. Each additional reduction technique improves the speedup by roughly one order of magnitude, while the additional hardware effort is moderate. On average, the CEDCESDCOL architecture yields a speedup of 3,600 over the CE architecture with an increase in hardware resources of only 80%.

The hardware required to implement an accelerator circuit depends strongly on the problem instance at hand, i.e., the distribution of the variables over the clauses. Although the size of some modules of the accelerator is either constant or linear in the number of variables, n, the checker modules depend on the specific structure of the problem instance. In our architecture, the constant modules take 210 slices. The state machines require $n \times 12$ slices, the cost counter $n \times 0.5$ slices, and the controller $n \times 1.5$ slices. The largest ESPRESSO benchmark for which we were able to implement an accelerator (CEDCES) in the XCV1000 FPGA has 313 variables and 302 clauses.

6. Conclusion and further work

We have discussed reconfigurable accelerators for the minimum-cost covering problem. These accelerators are instance-specific and generate customized circuits during runtime. Based on a standard tool flow leveraging on commercial design tools, we have conducted simulation and implementation experiments with small unate covering instances. Our experiments have demonstrated the high potential of accelerating minimum-cost covering problems with instance-specific hardware. We have achieved raw speedups up to five orders of magnitude for small covering instances. While these results are very promising, there are still challenges that must be answered to make instance-specific accelerators competitive to software solvers:

- *Improved architectures*. While our architecture implements many of the techniques found in software solvers, it still lacks reductions such as elimination of dominant rows and sophisticated cost bounds. Therefore, more reduction techniques and better cost bounds in hardware have to be investigated.
- *Reduced hardware compilation times*. For small problem instances, the high raw speedups are nullified by the long hardware compilation times. An important line of further research concentrates on bringing down hardware compilation times. A promising approach combines incremental, customized synthesis with partial FPGA reconfiguration.

Acknowledgment

This work was partially supported by Sundance Multiprocessor Technology Ltd.

References

1. M. Abramovici and J. T. De Sousa. A SAT solver using reconfigurable hardware and virtual logic. *Journal of Automated Reasoning*, 24(1–2):5–36, 2000.
2. M. Abramovici and D. Saab. Satisfiability on reconfigurable hardware. In *7th International Workshop on Field-programmable Logic and Applications (FPL'97)*, pp. 448–456, Springer, 1997.
3. J. Babb, M. Frank, and A. Agarwal. Solving graph problems with dynamic computation structures. In *Proceedings of SPIE: High-Speed Computing, Digital Signal Processing, and Filterung Using Reconfigurable Logic*, volume 2914, pp. 225–236, 1996.
4. O. Coudert. On solving binate covering problems. In *Proceedings of the IEEE/ACM Design Automation Conference*, pp. 197–202, June 1996.
5. A. Dandalis, A. Mei, and V. K. Prasanna. Domain specific mapping for solving graph problems on reconfigurable devices. In *Reconfigurable Architectures Workshop (RAW'99)*, 1999.
6. M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, (7):201–215, 1960.
7. G. De Micheli. *Synthesis and Optimization of Digital Circuits*, McGrawHill, 1994.
8. F. Fallah, S. Liao, and S. Devadas. Solving covering problems using LPR-based lower bounds. *IEEE Transactions on VLSI Systems*, 8(1):9–17, 2000.
9. J. Gu, P. W. Purdom, J. Franco, and B. W. Wah. Algorithms for the satisfiability (SAT) problem: A survey. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 35:19–151, 1997.
10. Z. Habbas, F. Herrmann, D. Singer, and M. Krajecki. Methodological approach to implement CSP on FPGA. In *10th International Workshop on Rapid System Prototyping*, pp. 66–71, 1999.
11. Y. Hamadi and D. Merceron. Reconfigurable architectures: A new vision for optimization problems. In *3rd International Conference on Principles and Practice of Constraint Programming (CP'97)*, pp. 209–221, Springer, 1997.
12. P. Leong, C. Sham, W. Wong, W. Yuen, and M. Leong. A bitstream reconfigurable FPGA implementation of the WSAT algorithm. *IEEE Transactions on VLSI Systems*, 9(1):197–201, 2001.
13. O. Mencer, M. Platzner, M. Morf, and M. J. Flynn. Object-oriented domain specific compilers for programming FPGAs. *IEEE Transactions on VLSI Systems*, 9(1):205–210, 2001.
14. M. Platzner and G. D. Micheli. Acceleration of satisfiability algorithms by reconfigurable hardware. In *8th International Workshop on Field-Programmable Logic and Applications (FPL'98)*, pp. 69–78, Springer, 1998.
15. R. Rudell and A. Sangiovanni-Vincentelli. Multiple-valued minimization for PLA optimization. *IEEE Transactions on CAD*, 6(5):727–750, 1987.

16. T. Shiozawa, K. Oguri, K. Nagami, H. Ito, R. Konishi, and N. Imlig. A hardware implementation of constraint satisfaction problem based on new reconfigurable LSI architecture. In *8th International Workshop on Field Programmable Logic and Applications (FPL'98)*, pp. 426–430, Springer, 1998.
17. S. Skiena. *The Algorithm Design Manual*, Springer, 1998.
18. T. Suyama, M. Yokoo, H. Sawada, and A. Nagoya. Solving satisfiability problems using reconfigurable hardware. *IEEE Transactions on VLSI Systems*, 9(1):109–116, 2001.
19. E. E. Swartzlander JR. Parallel counters. *IEEE Transactions on Computers*, C22(11):1021–1024, 1973.
20. M. Yokoo, T. Suyama, and H. Sawada. Solving satisfiability problems using field programmable gate arrays: first results. In *2nd International Conference on Principles and Practice of Constraint Programming (CP'96)*, pp. 497–509, Springer, 1996.
21. W. H. Yung, Y. W. Seung, K. H. Lee, and P. H. W. Leong. A runtime reconfigurable implementation of the GSAT algorithm. In *9th International Workshop on Field Programmable Logic and Applications (FPL'99)*, pp. 526–531, Springer, 1999.
22. 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 *35th Design Automation Conference (DAC)*, pp. 194–199, 1998.
23. P. Zhong, M. Martonosi, P. Ashar, and S. Malik. Using configurable computing to accelerate Boolean satisfiability. *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, 18(6):861–868, 1999.