

# Instance-Specific Accelerators for Minimum Covering

Christian Plessl and Marco Platzner  
Computer Engineering & Networks Lab  
Swiss Federal Institute of Technology (ETH) Zurich, Switzerland

**Abstract** *In this paper we present instance-specific accelerators for minimum-cost covering problems. We first define the covering problem and discuss a branch & bound algorithm to solve it. Then we describe an instance-specific hardware architecture that implements branch & bound in 3-valued logic and uses reduction techniques usually found in software solvers. Results for small unate covering problems reveal significant raw speedups.*

**Keywords:** reconfigurable computing, minimum covering, instance specific accelerator

## 1 Introduction

In the last years, several reconfigurable accelerators have been presented that speed up combinatorial problems. These accelerators are instance-specific and generate circuits on the fly that depend on problem instances rather than problems. It has been shown that instance-specific accelerators outperform software solvers for many hard instances of combinatorial problems, provided compilation time is kept short.

The best-investigated problem so far is the Boolean satisfiability problem (SAT) [7]. Given

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

SAT quests for an assignment of truth values to the variables that makes the Conjunctive Normal Form (CNF)  $C_1 \cdot C_2 \cdot \dots \cdot C_m$  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 targets discrete optimization problems. We concentrate on exact solvers for *minimum covering* problems. Minimum 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 [6].

## 1.1 Minimum covering

The minimum covering problem (set cover) is defined as follows: Given a set  $S$  of subsets of a finite set  $U$ , find the smallest subset  $T$  of  $S$  that covers  $U$ , i.e.,

$$\bigcup_{i=1}^{|T|} T_i = U.$$

A special instance of this general covering problem is, for example, vertex covering. Given a graph  $G = (V, E)$  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  $S_i$  as set of edges incident on vertex  $v_i$ . Instead of referring to the sets  $S_i$ , we can also refer to the vertices  $v_i$ . Figure 1a) shows an example with four vertices and four edges. The set  $\{v_2, v_3\}$  forms a minimum cover.

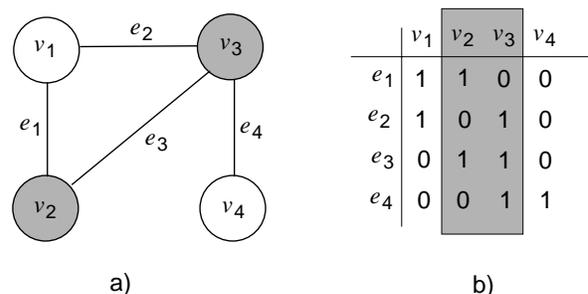


Figure 1: Vertex covering example: a) graph, b) matrix representation

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  $(x_1 + x_2) \cdot (x_1 + x_3) \cdot (x_2 + x_3) \cdot (x_3 + x_4)$ .

Both problems can be modeled using a matrix representation. Given  $\mathbf{A} \in \mathcal{B}^{m \times n}$ , where the set of rows corresponds to the elements of  $U$  ( $m = |U|$ ) and the columns correspond to the elements of  $S$  ( $n = |S|$ ), a cover corresponds to a subset of columns, having at least a 1 entry in all rows of  $\mathbf{A}$ . In SAT notation, a minimum-cost cover is a selection  $\mathbf{x} \in \mathcal{B}^n$ , such that  $\mathbf{A}\mathbf{x} \geq \mathbf{1}$  and  $\mathbf{c}^T\mathbf{x}$  is minimum. Figure 1b) shows the matrix for the vertex covering problem as well as one possible minimum cover.

CNFs corresponding to covering problems such as vertex covering are always *unate* Boolean expressions, i.e., all variables appear either non-inverted or inverted. Hence, these 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 albeit not necessarily 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  $(\bar{a} + b)$  has to be added, making the CNF *binate* since some literals 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.

## 1.2 Outline

Section 2 surveys related work in instance-specific accelerators for combinatorial problems. In Section 3, we discuss exact software solvers for covering problems and in Section 4 we present our reconfigurable accelerator. Section 5 discusses empirical results from simulation and prototype implementation.

## 2 Related Work

**Exact SAT** Zhong et al. [14] were the first to propose a reconfigurable accelerator that implements backtracking with Boolean constraint propagation [5] as basic deduction strategy. The authors presented two extensions to their architecture with conflict analysis techniques that allow for non-chronological backtracking and dynamic clause addition [15]. Platzner and De Micheli [9] presented several SAT architectures based on backtracking with 3-valued logic and don't care variables as deduction techniques. Suyama et al. [11] 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. This strategy is known to be weaker in its deductive power than Boolean constraint propagation. Abramovici and Saab [2] [1] presented an architecture based on the PODEM algorithm for automatic test pattern generation. Their architecture uses backtracing rather than backtracking and propagates the required result of the Boolean formula back to the variables.

**Stochastic local search for SAT** Hamadi and Mercer [8] describe an architecture that implements an adaption of GSAT, a greedy local search algorithm. GSAT starts with a random full value assignment and iteratively flips the values of variables in order to increase the number of satisfied clauses. GSAT is an incomplete algorithm and by proper settings of parameters software runtimes can be controlled. GSAT is applied to solve both the SAT and MAX-SAT problems. MAX-SAT is an optimization problem that tries to maximize the number of satisfied clauses of a CNF. An implementation of GSAT was presented by Yung et al. [13].

**Others** Babb et al. [3] presented reconfigurable architectures for computing the transitive closure and the shortest path in graphs, competing with the Bellman-Ford algorithm. A recent architecture that also aims at accelerating the Bellman-Ford algorithm was given by Dandalis et al. [4]. Other authors have discussed reconfigurable accelerators for constraint satisfaction problems (CSPs).

## 3 Unate Covering

### 3.1 Branch and bound

Covering problems can be solved with general search procedures such as branch & bound. A search tree is constructed 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 bound is computed that estimates the cost of a potential solution on the current path. When this bound exceeds the cost of the current best solution, the whole path can be pruned off. Otherwise, the algorithm selects a next variable and branches on it. In the worst case, branch & bound shows exponential complexity.

For many applications, however, reductions and bounds are quite effective.

## 3.2 Reductions

In the following, we focus on unate covering problems with unit cost. Each variable has an associated cost value of 1. Such covering problems have to be solved in exact two-level logic minimization, for example by the ESPRESSO-EXACT algorithm [10]. Several reduction rules can be applied to simplify the matrix  $\mathbf{A}$ .

- **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.
- **Dominating 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.
- **Dominated 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 the exact unate covering algorithm. Matrix  $\mathbf{A}$  defines the covering problem, the vector  $\mathbf{x}$  represents the current variable assignment, and  $\mathbf{b}$  is the vector with the lowest cost found so far. The algorithm starts with  $(\mathbf{A}, \mathbf{0}, \mathbf{1})$ .

In line 2, the matrix is reduced by iteratively applying the reduction rules. Further, the algorithm tries to reorder rows in a way such that the matrix becomes block-diagonal. This would allow to split the covering problem into two independent subproblems.

If no more reduction is possible the bounding condition *Current\_estimate* is computed in line 3. *Current\_estimate* gives a lower bound on the cost of a potential solution on the current path. If the bound does not allow to prune the current path, a branch is performed by selecting a variable  $x_c$  and assigning 1 (line 10) and 0 (line 16) to it, respectively. In both cases, the matrix is modified and the covering procedure is called recursively.

---

### Algorithm 1 Exact cover algorithm [6]

---

```

1: exact_cover ( $\mathbf{A}, \mathbf{x}, \mathbf{b}$ ) {
2: Reduce matrix  $\mathbf{A}$  and update corresponding  $\mathbf{x}$ 
3: if (Current_estimate  $\geq |\mathbf{b}|$ ) then
4:   return( $\mathbf{b}$ )
5: end if
6: select a branching column  $c$ 
7: if ( $\mathbf{A}$  has no rows) then
8:   return( $\mathbf{x}$ )
9: end if
10:  $x_c = 1$ 
11:  $\tilde{\mathbf{A}} = \mathbf{A}$  after deleting column  $c$  and rows incident
    to it
12:  $\tilde{\mathbf{x}} = \text{exact\_cover}(\tilde{\mathbf{A}}, \mathbf{x}, \mathbf{b})$ 
13: if  $|\tilde{\mathbf{x}}| < |\mathbf{b}|$  then
14:    $\mathbf{b} = \tilde{\mathbf{x}}$ 
15: end if
16:  $x_c = 0$ 
17:  $\tilde{\mathbf{A}} = \mathbf{A}$  after deleting column  $c$ 
18:  $\tilde{\mathbf{x}} = \text{exact\_cover}(\tilde{\mathbf{A}}, \mathbf{x}, \mathbf{b})$ 
19: if  $|\tilde{\mathbf{x}}| < |\mathbf{b}|$  then
20:    $\mathbf{b} = \tilde{\mathbf{x}}$ 
21: end if
22: return( $\mathbf{b}$ )
23: }
```

---

## 4 Covering in Hardware

### 4.1 Accelerator Architecture

The architecture of our covering accelerator is shown in Figure 2 and divided into four blocks: *state machines (sm)*, *checkers*, *cost counter* and *controller*. The state machines control the variables' values. Each state machine implements part of the branch & bound algorithm. All state machines are arranged in a linear array, where each state machine is connected only to its immediate neighbors, the checkers, and the controller. The outputs of the state machines are the current variable assignments that are fed into the checkers. The checkers deduce information from partial value assignments. 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.

**Backtracking with 3-valued logic** The basic backtracking mechanism is similar to the one in [9] 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 cover with minimum cost has been found and we backtrack again to continue search in another path of the search tree. If the CNF is  $X$  we have to 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, a state machine changes its assignment, triggers the next state machine (continue search), or triggers the previous state machine (backtrack).

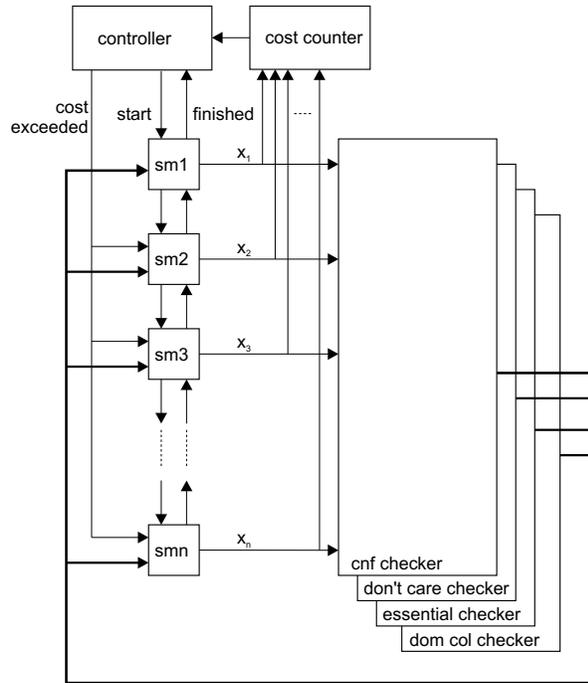


Figure 2: Architecture of the covering accelerator

**Reductions** We have designed checkers for the reduction techniques *essential columns*, *don't cares*, and *dominated columns*. These techniques are explained on the example of the covering matrix shown in Figure 1b) which corresponds to the CNF:  $(x_1 + x_2) \cdot (x_1 + x_3) \cdot (x_2 + x_3) \cdot (x_3 + x_4)$ .

- **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 1b) 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. The logic for these conditions can be computed from the CNF at compile time. For example, the essential condition for  $v_2$  is  $(\bar{x}_1 + \bar{x}_3)$ .

- **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 1b) is set to 1,  $v_4$  becomes a don't care variable. Don't care conditions are derived from the CNF for each variable at compile time and are implemented in the don't care checker. The don't care condition for  $v_3$  is  $(x_1 \cdot x_2 \cdot x_4)$ .
- **dominated column** A variable that corresponds to a dominated column can be set to 0. In the matrix of Figure 1b, column  $v_3$  dominates column  $v_4$ . The checker module implements logic for each variable to indicate the dominated condition. Column  $v_2$  is dominated by  $v_1$  if row  $e_3$  is covered, by  $v_3$  if row  $e_1$  is covered, and by  $v_4$  if both rows  $e_1$  and  $e_3$  are covered. This results in a dominated condition  $dom(x_2) = (x_1 + x_3)$ .

Each checker result is a function of the current variable assignment and not of previous variable states. Thus the checkers can be implemented in pure combinatorial logic that is derived from the CNF at compile time. 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, and outputs the resulting condition vector. The CNF checker differs from the other checkers since it computes only a single 3-valued logic signal, the CNF result.

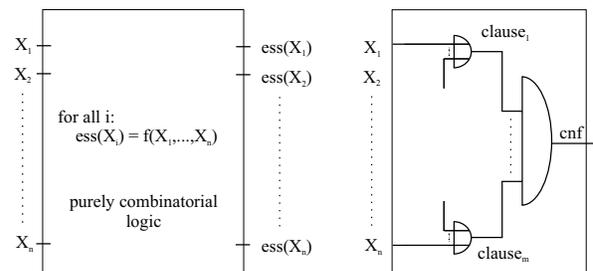


Figure 3: Architecture of the checker modules

**Bound** Our architecture uses the current best cost as lower bound. There is no estimation of the cost added by variables not yet searched. While this leads to a straight-forward hardware implementation, it lacks the sophisticated bounding techniques applied in most software solvers.

**Cost counter** For evaluating the bound 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 variable assignment, i.e., potentially every cycle, an efficient implementation of the cost counter is of utmost importance for the overall performance of the accelerator. To compute the current cost, a structure called *n-bit-parallel counter* is used. 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 [12]. Figure 4 shows the implementation of a 15-bit parallel counter.

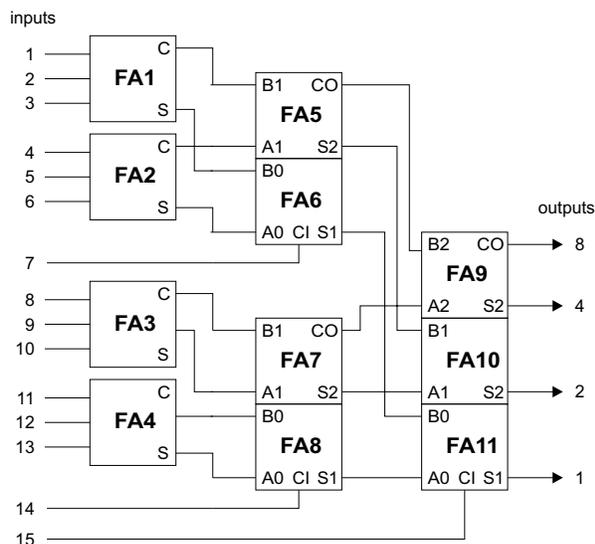


Figure 4: Architecture of a 15-bit cost counter

The counter is built from full-adders (FA) organized in stages. The first stage uses 1-bit full-adders. Their outputs are fed to the 2-bit full-adders in the next stage and so forth. The regular structure of the adders allows for an implementation using the fast-carry chain routing resources in FPGAs. A counter with  $n$  input bits results in  $\lfloor \log_2(n) \rfloor$  levels. Hence the delay of a  $n$  bit parallel counter is given as:

$$\tau_{n\text{-bit-ctr}} = \frac{l(l+1)}{2} \cdot \tau_{\text{adder}} \quad \text{with } l = \lfloor \log_2(n) \rfloor,$$

where  $\tau_{\text{adder}}$  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.

## 4.2 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 reduction and 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 implement backtracking search in 3-valued logic.

Binare covering problems require slightly different reduction techniques. While don't cares apply to both unate and binare problems, essentials differ as in binare 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 be adapted to integer cost by replacing the cost counter with a cost adder.

## 5 Experiments

Figure 5 illustrates the overall tool flow for the covering accelerator. The 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 synthesis. In the synthesis path, the Xilinx Foundation 3.1i tools including FPGA Express is called to produce the FPGA configuration bitstream. The bitstream is downloaded onto the FPGA in our prototyping hardware and the accelerator is started. The algorithm completes when the top-most statemachine asserts its finished-signal to signal the controller (see Figure 2). The monitor, a host software routine, polls the controller continuously for completion. When the accelerator has completed, the host reads back the result.

For testing and benchmarking the accelerators circuits, we have chosen 21 small problems of the ESPRESSO distribution. We have instrumented ESPRESSO to output the *cyclic cores*, i.e., the covering matrices just before the first branch and after the first

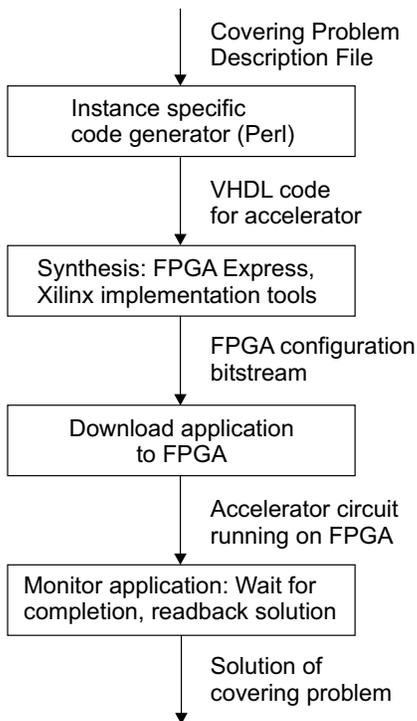


Figure 5: Tool flow for the covering accelerator

round of reductions. These test problems have between 6 to 45 variables and 6 to 39 clauses.

## 5.1 Simulation

Each of the 21 examples has been simulated using the Modelsim VHDL simulator. Figure 6 shows the number of problems over the resulting raw speedup. The speedup was calculated as  $S_{raw} = t_{sw}/t_{hw}$  and does not include hardware compilation time. The software execution time was determined by measuring the time ESPRESSO needs for covering the cyclic core on a workstation; the hardware execution time has been determined by VHDL simulation and an assumed clock rate of 25 MHz. The software execution times for the chosen problems are in the range of  $1ms$  to  $2.13s$ .

For three of the problems no speedup at all was achieved, other problems show speedups of several orders of magnitude. It must be noted that, besides the cnf checker, the simulated accelerators use only essentials and don't care as reductions, and the current best cost as lower cost bound. At the time of this writing, the reduction technique of dominated columns is implemented, but not fully tested. Software covering algorithm employs more reduction techniques and an improved lower bound. Depending on the problem instance this can either drastically improve the perfor-

mance or just add a large overhead.

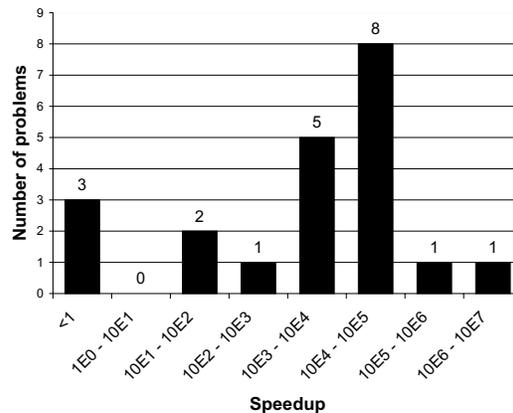


Figure 6: Raw speedup for the covering accelerator

## 5.2 Implementation

We are currently implementing accelerator circuits on a prototyping platform. The platform consists of a PC with the PCI carrier-board SMT320 from Sundance, a carrier that can be populated with four TIM standard compliant modules ([www.sundance.com](http://www.sundance.com)). For the accelerator prototype we use one FPGA TIM module, the Sundance SMT358, that 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 12288 slices.

The results show that the benchmark circuits achieve clock rates of 30–50 MHz which confirms the speedup numbers in Figure 6. Synthesis and design implementation tools run on a Pentium-III 550 MHz system with 640 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 example, a problem consisting of 27 variables and 25 clauses takes 8 minutes for code generation, circuit synthesis, place, and route. FPGA configuration and readback of the solution require some seconds. This problems required an area of 907 slices, which amounts to 7.3 % of the FPGA.

## 5.3 Discussion

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 hardware compilation time can be reduced.

It is difficult to predict the number of variables and clauses of the largest covering problem instance that might be accommodated in the FPGA. Some modules of the accelerator depend on the specific structure of the problem instance. Other parts, however, are either constant or a function of the number of variables,  $n$ . The constant modules of the accelerator take 210 slices. The statemachines require  $n \times 14$  slices, the cost counter  $n \times 0.5$  slices, and the controller  $n \times 1.5$  slices. The checkers strongly depend on the problem instance. Under the assumption that the relative area occupied by the checkers, in percent of the overall accelerator size, remains roughly constant with varying problem size, we could extrapolate from our small benchmarks that covering problems up to 600 variables should fit into the FPGA. We were able to implement a larger ESPRESSO benchmark problem with 313 variables and 302 clauses. The implementation utilizes 58% of the FPGAs resources and runs at 14 MHz without any speed optimizations. We failed, however, to implement a 550 variable problem due to space constraints. This demonstrates that the size of the checkers and thus the overall size strongly depends on the distribution of the variables over clauses.

## 6 Conclusion and Further Work

We have shown the potential of accelerating minimum cost covering with instance specific accelerators. While our architecture implements many of the techniques used in software, it still lacks reduction techniques such as elimination of dominant rows and sophisticated cost bounds. For small problems the achieved raw speedups are significant. Long hardware compilation times, however, make the accelerator unattractive for such a problem set. Therefore, ongoing work includes

- the investigation of more reduction techniques and cost bounds,
- reducing hardware compilation time by incremental synthesis and partial reconfiguration of the FPGA, and
- applying the accelerator to larger unate and binate covering problems.

## 7 Acknowledgments

This work was partially supported by Sundance Multi-processor 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 *Int'l Workshop on Field-programmable Logic and Applications*, pages 448–456. Springer, 1997.
- [3] J. Babb, M. Frank, and A. Agarwal. Solving graph problems with dynamic computation structures. In *SPIE: High-Speed Computing, Digital Signal Processing, and Filtering Using Reconfigurable Logic*, volume 2914, pages 225–236, 1996.
- [4] A. Dandalis, A. Mei, and V. K. Prasanna. Domain Specific Mapping for Solving Graph Problems on Reconfigurable Devices. In *Reconfigurable Architectures Workshop*, 1999.
- [5] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, (7):201–215, 1960.
- [6] G. De Micheli. *Synthesis and Optimization of Digital Circuits*. McGrawHill, 1994.
- [7] 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.
- [8] Y. Hamadi and D. Merceron. Reconfigurable Architectures: A New Vision for Optimization Problems. In *Int'l Conference on Principles and Practice of Constraint Programming*, pages 209–221. Springer, 1997.
- [9] M. Platzner and G. D. Micheli. Acceleration of Satisfiability Algorithms by Reconfigurable Hardware. In *Int'l Workshop on Field-Programmable Logic and Applications*, number 1482, pages 69–78, 1998.
- [10] R. Rudell and A. Sangiovanni-Vincentelli. Multiple-valued Minimization for PLA Optimization. *IEEE Transactions on CAD/ICAS*, 6(5):727–750, 1987.
- [11] T. Suyama, M. Yokoo, and A. Nagoya. Solving satisfiability problems on FPGAs using experimental unit propagation. In *Int'l Conference on Principles and Practice of Constraint Programming*, volume 1713, pages 434–445. Springer, 1999.
- [12] E. E. Swartzlander JR. Parallel counters. *IEEE Transactions on Computers*, C-22(11):1021–1024, November 1973.
- [13] W. H. Yung, Y. W. Seung, K. H. Lee, and P. H. W. Leong. A Runtime Reconfigurable Implementation of the GSAT Algorithm. In *Int'l Workshop on Field Programmable Logic and Applications*, pages 526–531. Springer, 1999.
- [14] 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 *Design Automation Conference*, pages 194–199. IEEE, 1998.
- [15] 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, June 1999.