Custom Computing Machines for the Set Covering Problem

Christian Plessl and Marco Platzner
Computer Engineering and Networks Laboratory
Swiss Federal Institute of Technology (ETH) Zurich
CH-8092 Switzerland

Abstract

We present instance-specific custom computing machines for the set covering problem. Four accelerator architectures are developed that implement branch & bound in 3-valued logic and many of the deduction techniques found in software solvers. We use set covering benchmarks from two-level logic minimization and Steiner triple systems to derive and discuss experimental results. The resulting raw speedups are in the order of four magnitudes on average. Finally, we propose a hybrid solver architecture that combines the raw speed of instance-specific reconfigurable hardware with flexible bounding schemes implemented in software.

1 Introduction

During the last years, a number of prototypes demonstrated that custom computing machines can speed up hard combinatorial problems by some orders of magnitude. The key for achieving impressive speedups lies in the exploitation of massive parallelism at fine levels of granularity. The prime example is the various bit operators used in the evaluation of Boolean expressions.

Currently, reconfigurable accelerators are competitive to software techniques only for hard problems with rather long runtimes. This is because reconfigurable accelerators pay a significant overhead in latency due to reconfiguration and downloading times. Moreover, many of the proposed accelerators for combinatorial problems are instance-specific and generate and map circuits on the fly to reconfigurable hardware. This allows for optimal customization but suffers additionally from the latency overhead due to hardware compilation.

The problem that received most attention so far is Boolean satisfiability (SAT) [7]. SAT is a discrete decision problem that takes a Boolean expression in Conjunctive Normal Form (CNF) as input and tries to find truth values for the variables such that the CNF evaluates to true. Reconfigurable accelerators have been presented that solve SAT exactly, e.g., in [15][16][1][12], and with inexact stochastic local search procedures [9] [10]. Other combinatorial problems attacked include polynomial graph problems [2] and constraint satisfaction problems [8].

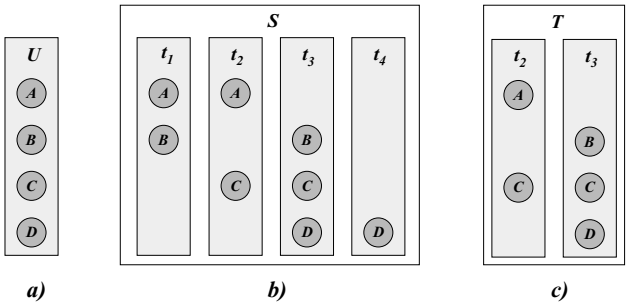

Figure 1: Set covering example. a) set $U = \{A, B, C, D\}$, b) set $S = \{t_1, t_2, t_3, t_4\}$ of subsets of U, c) a minimum-cost cover consisting of $T = \{t_2, t_3\}$

Most of the related work on instance-specific accelerators targets discrete decision problems. This paper presents an accelerator for the discrete *optimization* problem of *set covering* (minimum-cost covering):

Definition 1.1 (Set Cover [14]) *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:*

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

Figure 1 shows an example set covering problem. Set covering problems can be considered as minimum-cost SAT problems. Minimum-cost SAT seeks for a satisfying solution for a given CNF that minimizes a linear cost function over the variables, $\mathbf{c}^T\mathbf{v}$, where $\mathbf{c}$ is a cost vector. A set cover problem is transformed to a minimum-cost SAT problem by identifying the elements of S with the binary decision variables v_i and the elements of U with the clauses of the CNF. The CNF corresponding to the set covering problem of Figure 1 is $(v_1 + v_2) \cdot (v_1 + v_3) \cdot (v_2 +$

Published in: *Proc. Int. Symp. on Field-Programmable Custom Computing Machines (FCCM)*
 DOI: 10.1109/FPGA.2002.1106671

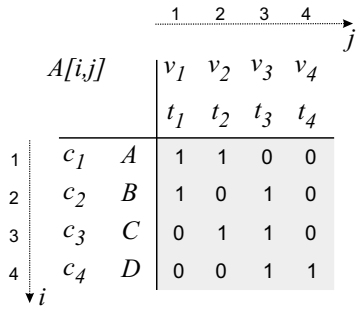

Figure 2: Covering matrix for the set covering example in Figure 1.

$v_3) \cdot (v_3 + v_4)$.

Covering problems are often modeled using a matrix representation. Given $\mathbf{A} \in \mathcal{B}^{m \times n}$, a cover corresponds to a subset of columns, having at least one 1 entry in all rows of $\mathbf{A}$. Figure 2 shows the covering matrix $A[i, j]$ for the set covering problem in Figure 1.

A CNF where all variables appear either non-negated or negated is a *unate* Boolean expression; the corresponding covering problem is *unate covering*. The consequence of a CNF being unate is that selecting all elements of S always results in a cover, but not necessarily a minimum cover. *Binate covering* problems contain literals that appear in both non-negated and negated forms. We concentrate on unate covering problems with unit cost, i.e., each variable has cost 1. Most of the discussed algorithms and techniques have modified versions that apply to integer cost and binate covering problems [4].

Set covering problems have many applications [14]. In computer-aided design, they are found in synthesis and optimization of program code and digital circuits [4], e.g., two-level logic minimization, state minimization and encoding. As set covering is NP-hard [6], practical interest concentrates on the design and implementation of fast heuristics and approximation algorithms. However, our work on exact covering solvers is motivated by following reasons: First, speedups in the order of several magnitudes enables us to solve a lot more problems of practical interest to optimality in reasonable time frames. Second, fast exact solvers are required to evaluate the performance of heuristics and approximation algorithms. Third, devising new exact methods can lead to the development of improved heuristics [3].

We introduced first concepts of a reconfigurable accelerator for minimum-cost covering in a previous publication [13]. This paper extends the previous one and includes following additional contributions:

- a detailed presentation of the accelerator architectures
- additional reduction technique of dominated columns
- more benchmark classes and experiments
- concept of a novel hybrid hardware/software solver

2 Software Techniques for Covering

Set covering problems are solved exactly by search procedures such as branch & bound. Branch & bound constructs a search tree by iteratively selecting a variable and branching on it. This generates two subtrees with the variable set to 1 and 0, respectively. Two different techniques prune the search space. First, reduction techniques infer knowledge from the current partial assignment in order to reduce the problem. Second, a lower bound for the cost of a potential solution on the current path is computed. The whole subtree can be pruned off, if the bound exceeds the cost of the current best solution,

Algorithm 1 shows a recursive implementation of an exact unate covering algorithm. Matrix $\mathbf{A}$ defines the covering problem, the vector $\mathbf{v}$ represents the current variable assignment, and $\mathbf{b}$ is the lowest cost solution found so far. The algorithm starts with $(\mathbf{A}, \mathbf{0}, \mathbf{1})$. 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 $\mathbf{A}$ 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.

Branch & bound can employ different heuristics for the variable order and the order of value assignments, i.e., 0 first or 1 first. The most prominent variable order heuristic picks the variable next that covers the largest number of currently uncovered rows. The basic reductions are essentiality and dominance:

- **Essential columns** Essential columns are columns having the only 1 entry of some row. An essential column must be part of *any* cover. For example, setting v_4 to 0 in Figure 2, makes v_3 essential.
- **Dominating columns** Column v_x dominates column v_y if the entries of v_x are larger than the entries of v_y. Dominated columns can be discarded from consideration, because selecting the dominating column instead covers more rows. For example, v_3 dominates v_4 in Figure 2.
- **Dominant rows** Row c_x dominates row c_y if the entries of c_x are larger than the entries of c_y. Dominant

Algorithm 1 Exact covering algorithm [4]

```
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: }
```

rows can be discarded, because any cover of the dominated row is also a cover of the dominant row. For example, setting v_4 to 0 in Figure 2 makes c_3 dominant over c_4.

The cost bound is the sum of the cost of the current partial assignment plus an estimate for the number of columns required to cover the remaining rows. The standard technique for the lower cost bound is to compute the size of the maximally independent set of rows that remain to be covered. As this is itself a hard problem, heuristics are applied.

Most of the improvements in solving set covering problems have been made by additional and more sophisticated reductions, bounding schemes, and heuristics for the selection of the branching variable [5][3][11].

3 Reconfigurable Architecture

The architecture for the covering accelerator is shown in Figure 3 and consists of four blocks:

- an array of *finite state machines (FSMs)*
- the *checker* modules
- the *cost counter*
- the *controller*

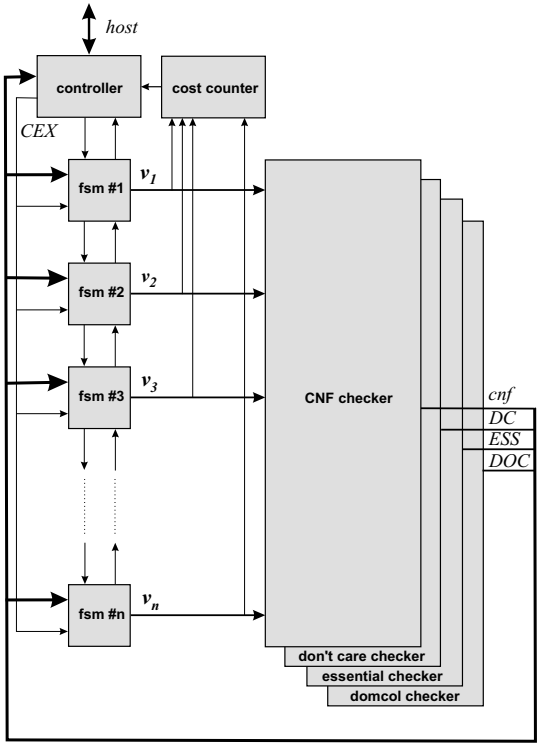

Figure 3: Block diagram for the set covering accelerator.

The architecture is similar to previous SAT architectures and implements a backtracking search procedure in 3-valued logic [12]. 3-valued logic allows a variable to take on the values $\{X, 0, 1\}$, where X denotes an unassigned variable.

3.1 Basic Branch & Bound

Figure 3 shows that the state machines are arranged in a linear array, where each state machine connects only to its immediate neighbors above and below, the checkers, the cost counter, and the controller. Each FSM controls one variable and provides the current variable value. The basic architecture relies only on the CNF checker which deduces information from a partial assignment by computing the 3-valued cnf. The cnf is fed back to the state machines and the controller. The cost counter computes the cost of the current partial assignment. The controller handles host I/O, initializes and stops the search procedure and computes the cost bound (*CEX*), i.e., decides for each assignment whether to continue search on the current path or backtrack due to exceeded cost.

Figure 4 displays the state machine for the basic architecture with assignment order $0{-}1$. The FSM initializes its variable with X. When an FSM is triggered from the

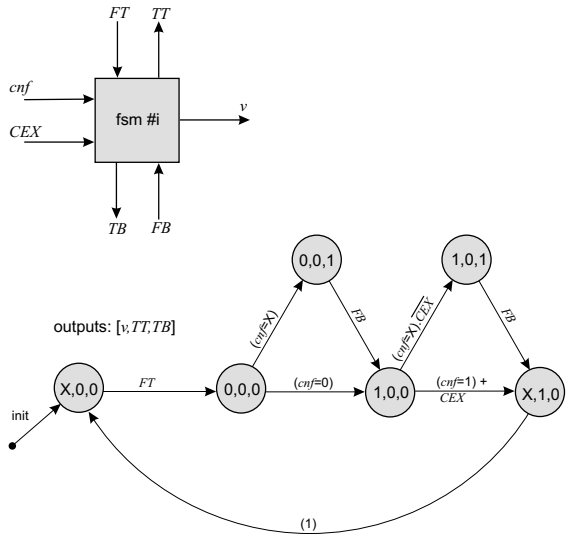

Figure 4: FSM input/outputs and state diagram for the basic branch & bound architecture with assignment order 0–1.

FSM above (*FT*), it assigns the value 0 and then checks the CNF result. If cnf is X search proceeds on the current path. To this end, the FSM triggers the next FSM below (*TB*) and then waits until it gets triggered again from the FSM below (*FB*). If cnf is 0, the complementary value 1 is tried. After assigning 1 – which may increase cost – the cnf value is checked again. If the cnf is X and the current cost does not exceed the best cost found so far, search proceeds on the current path. If the cost exceeds the best cost, backtracking is required to continue search on another path of the search tree. If the cnf is 1, backtracking is also required. In the latter case a new cover with minimum cost is found when the current best cost is not exceeded. Then the controller saves the satisfying variable vector and the new best cost.

Search Heuristics There are two decisions that influence search efficiency: the *variable order* and the *assignment order*. Our compilation system allows to select different static heuristics at circuit compile time. The FSMs can be placed according to:

- *index order:* The variables are placed in order of their index in the problem instance.
- *frequency order:* Variables that appear in many clauses are placed on top of the array of state machines. The motivation is that by the first few assignments many clause values can be determined.

For the assignment order we provide:

- $0-1$: The FSM assigns 0 first and then 1. This is likely to run in a contradiction very soon. Finding a solution is hard, but when one is found its cost will be rather low.
- $1-0$: The FSM assigns 1 first and then 0. This will lead to a monotonically increasing number of satisfied clauses and no contradiction can ever happen. A solution with high cost will be found rather quickly.
- *random:* For each variable, either the order $0-1$ or $1-0$ is selected.

Cost Bound In its current version, the cost estimate of our accelerator is the cost of the partial assignment plus one for the bound on the additional cost on the current path. This is a rather poor cost estimate and cannot compete with the bounding techniques of software solvers. In section 5 we therefore propose a hybrid accelerator that allows to use more sophisticated bounds.

3.2 Reductions in Hardware

The following sections list the reductions that are available for our accelerator: don't cares, essentials and dominated columns. Variants of our don't cares and essentials have been used previously in SAT accelerators; dominated columns in hardware are specific to set covering. Each reduction requires its own checker module which is derived from the covering matrix and implemented in pure combinatorial logic during compile time. Additionally, each architecture using a specific combination of reductions requires a specific FSM. Figure 5 shows the inputs/outputs and the state diagram for the FSM handling the complete set of reductions. Don't cares and dominated columns force variables to 0, essentials to 1. If no reduction applies, the FSM assigns 1 first to its variable. The state diagram shows a wait state after assigning 0. This is necessary because setting a variable to 0 may cause essentials. These essentials will force their variables to 1 in the next state, which in turn may satisfy the CNF.

3.2.1 Don't Cares

Don't care variables are unassigned variables that appear only in already satisfied clauses. For example, when v_3 in Figure 2 is set to 1, v_4 becomes a don't care variable. Such variables cannot contribute to a solution and should thus be assigned 0 to minimize cost. The don't care checker computes a binary signal DC for each variable indicating whether a don't care condition currently applies. If a state machine is triggered from above while the don't care condition applies, the state machine sets its variable to 0 and passes control directly to the state machine below (see Figure 5). If triggered from below, the state machine relaxes

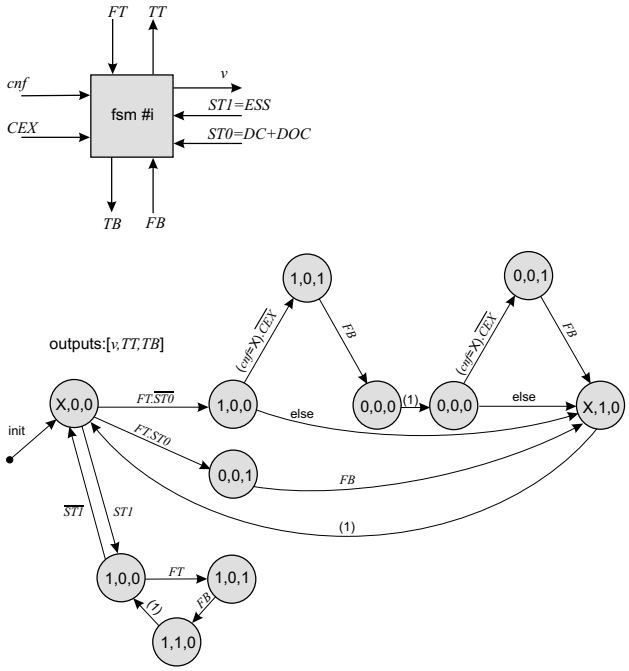

Figure 5: FSM input/outputs and state diagram for the covering accelerator using don't cares, essentials and dominated columns. The assignment order is $1{-}0$.

its variable to X and passes control to the state machine above.

The 3-valued clause values are intermediate signals of the CNF checker. Hence the don't care checker logic is efficiently generated $O(nm)$ time. The don't care condition for a variable v_s is given by:

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

3.2.2 Essentials

Essentials are unassigned variables that are implied by the current partial assignment and thus must be set to 1. Essentials correspond to implications in solving SAT. The difference is, however, that in unate covering problems there is no constraint propagation. An essential variable that is set to 1 cannot imply another variables. The essential checker computes a binary signal ESS for each variable indicating whether an essential condition currently applies. If an essential condition occurs, the FSM immediately sets its variable value to 1 (see Figure 5). This is necessary as an implied variable can lead to a satisfying assignment or further don't cares and dominated columns. If later on the state machine is triggered, it passes control directly to the state machine below and above, respectively. The variable is only relaxed to X when the essential condition disappears.

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 $O(mn^2)$.

Algorithm 2 Generation of the essential condition

input: Matrix $A[i,j]$, variable index s
output: essential logic ess_{v_s}

```
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)
```

3.2.3 Dominated Columns

Dominated columns refer to variables that can be set to 0 because there is another column that covers at least as many rows and should thus be preferred. The dominated column checker computes a binary signal DOC for each variable indicating whether a dominated column condition currently applies. The state machines handle the dominated column condition in the same way as a don't care condition.

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 6a) 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

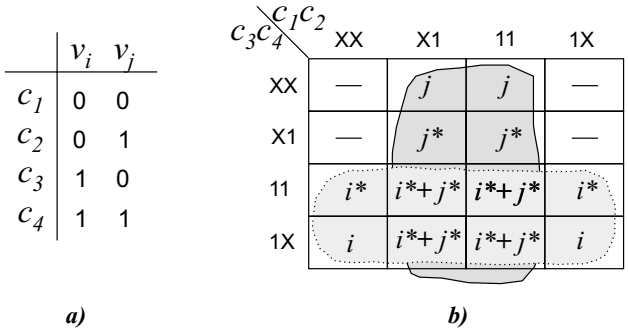

Figure 6: Possible row values for two columns and Karnaugh map for the dominated column condition.

whether and which column dominates, we need to consider only the values X and 1 for the rows. Figure 6b) shows the possible value combinations for the rows (clauses) of Figure 6a). 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 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 6b) 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:

$$doc_{(v_i,v_j)} \leftarrow (c_3 = 1) \wedge (v_j \neq 0) \quad (3)$$
$$doc_{(v_j,v_i)} \leftarrow (c_2 = 1) \wedge (v_i \neq 0) \quad (4)$$

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 for some other variable is assigned 0. This assignment breaks the column dominance. Consequently, above conditions include terms of type $(v \neq 0)$. At this point our reduction is weaker than column dominance in software. In software, 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, i.e., the corresponding rows are 1. For v_i to dominate v_j, all k pairs $(0,1)$ must have been covered.

$$doc_{(v_i,v_j)} \leftarrow (\bigwedge_{t=1}^{l} (c_{\beta_t} = 1)) \wedge (v_j \neq 0) \quad (5)$$
$$doc_{(v_j,v_i)} \leftarrow (\bigwedge_{s=1}^{k} (c_{\alpha_s} = 1)) \wedge (v_i \neq 0) \quad (6)$$

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:

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

Overall, there are $\binom{n}{2}$ dominated column conditions to generate which takes $O(mn^2)$ time.

Algorithm 3 Generation of a dominated column condition

input: Matrix $A[i,j]$, variable indices s and t
output: dominated column logic $doc_{(v_s,v_t)}, doc_{(v_t,v_s)}$

```
1: lproduct ← "1"
2: kproduct ← "1"
3: for (i = 1 to m) do
4:    if ((A[i,s] = 1) ∧ (A[i,t] = 0)) then
5:       lproduct ← lproduct | "·(c_i = 1)"
6:    else if ((A[i,s] = 0) ∧ (A[i,t] = 1)) then
7:       kproduct ← kproduct | "·(c_i = 1)"
8:    end if
9: end for
10: doc_(v_s,v_t) ← lprod | "·(v_t ≠ 0)"
11: doc_(v_t,v_s) ← kprod | "·(v_s ≠ 0)"
12: return(doc_(v_s,v_t), doc_(v_t,v_s))
```

3.2.4 Checkers vs. Software Reductions

Most software solvers operate on the covering matrix as basic data structure. For example, the reduction technique essentials involves following steps in software:

(1) search for columns that provide the only one 1-entry for some row; for each such column i:

a) set $v_i \leftarrow 1$ and delete column i
b) delete all rows incident to column i, i.e., rows with 1-entries in column i
c) search for columns that have no more 1-entries; for each such column j:
   (i) set $v_j \leftarrow 0$ and delete column j

In contrast to that, the only elements having state in our accelerator architectures are the FSMs. The checkers implementing the reductions are state-less. The basic covering accelerator utilizing don't cares and essentials is functionally fully equivalent to the software technique essentials. Steps 1) and a) are computed in parallel for all variables in two clock cycles. Step c) is computed in parallel for all variables in one clock cycle. Step b) has no direct correspondence in hardware; step (i) is done when the FSMs are activated. The dominated columns reduction in software is stronger than ours (see section 3.2.3).

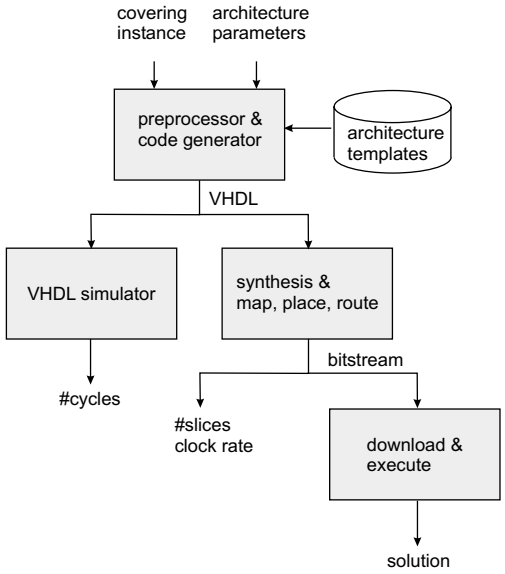

Figure 7: Experimental setup.

4 Experiments

4.1 Prototype Implementation

Figure 7 illustrates our experimental setup. The compilation system starts with preprocessing and code generation. Inputs to this step are the problem instance and parameters that select heuristics and reductions. The reductions setting determines the accelerator variant. We define four accelerator architectures:

- *CE:* basic architecture with CNF evaluation
- *CEDC:* CE plus don't cares
- *CEDCES:* CEDC plus essentials
- *CEDCESDCOL:* CEDCES plus dominated columns

The code generator is written in Perl and uses VHDL templates for the non-instance specific parts of the architecture. The instance-specific parts are generated at circuit compilation time. We use the generated VHDL code for both simulation with ModelSim and synthesis. The synthesis path calls Xilinx Foundation 3.1i tools including FPGA Express to produce a configuration bitstream for Xilinx Virtex. The last part of the tool flow downloads the bitstream onto the FPGA platform and starts the accelerator. A monitor program running on the host polls the accelerator for completion and reads back the result. Our prototyping hardware consists of a PC with a Sundance PCI carrier board and one SMT358 FPGA module populated with a Xilinx Virtex XCV1000-BG560-4 device.

4.2 Benchmarks and Reference Solvers

We have chosen two sources for unate covering instances: *CAD* with two-level logic minimization and *Steiner triple systems*. The logic minimization benchmarks consist of 21 problems taken from 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 round of reductions. These test problems have from 4 to 62 variables and 4 to 70 clauses. As reference solver we used Espresso `exact` running on a Sun Ultra-10 workstation.

Set covering problems arising from Steiner triple systems (STS) are considered hard covering problems and have previously been used to test the quality of covering algorithms [11]. The covering matrix for STS_n has following properties:

- n columns
- every row contains exactly three one-entries
- for every pair of columns j and k there exists exactly one row i such that $A[i][j] = A[i][k] = 1$

These conditions imply a number of rows equal to $n(n-1)/6$. In our experiments, we used the instances STS_9, STS_{15} and STS_{27}, i.e. the largest problme consists of 27 variables and 117 clauses. As software reference we use both Espresso and a solver presented in [11].

4.3 Results

In the following we summarize results from several experiments. First, we investigate the influence of the heuristics on the performance of the accelerators. Then, we analyze absolute and relative raw speedups compared to software reference solvers. Third, we determine hardware cost and compare the performance in relation to the required hardware.

4.3.1 Heuristics

To analyze the influence of the heuristics for the variable order (index and frequency heuristic) and the assignment order ($0-1$, $1-0$, and random), we exhaustively simulated sets of smaller benchmark problems. We choose 16 smaller instances of two-level logic minimization and the three Steiner triple systems STS_9, STS_{15}, and STS_{27}.

Logic minimization

- **CE** The influence of the variable order on the performance is insignificant. In 77% of the examples, the assignment order $0-1$ performs best; in 23% random assignment order is superior.
- **CEDC** The frequency variable order is superior in 58% of the cases. For 46% of the examples, random assignment order is best; $0-1$ order is best in 31%.
- **CEDCES** The frequency variable order is superior in 60% of the benchmarks. Random assignment performs clearly superior in 74% of the cases; $0-1$ order in 19% and $1-0$ order in 7%.
- **CEDCESDCOL** There is no significant difference between index and frequency variable order. The $1-0$ assignment order is best in 63%; random assignment order in 30% and $0-1$ in only 7% of the cases.

These results make evident that for larger problems in two-level logic minimization the frequency heuristic for variable ordering should be selected and that the optimal assignment order is architecture-dependent. The assignment order $0-1$ is advisable for CE and CEDC, random assignment for CEDCES, and for CEDCESDCOL assignment order $1-0$ should be used.

Steiner triple systems For STS problems the specific selection of heuristics has a very low impact on the performance. The variable order has no influence at all. The assignment order leads to only minor differences in the accelerator performance. Averaged over all three instances and all architectures, the variation is $\pm 4.2\%$ with respect to the mean value. The maximum variation is 15%.

4.3.2 Speedups

Absolute raw speedups We determine the raw speedups by comparing the runtimes of the covering accelerators to the runtimes of software reference solvers. The reported raw speedups do not include times for code generation, synthesis, FPGA configuration and readback of the solution. The number of clock cycles for the accelerators were determined by VHDL simulation of the architecture variant CEDCESDCOL with frequency variable order and $1-0$ assignment order. We assume a clock frequency of 25 MHz. Subsequent FPGA synthesis experiments confirmed that the resulting accelerator circuits run at 25-50 MHz.

Figure 8 shows the raw accelerator speedups for logic minimization benchmarks. The graph is partitioned into smaller instances on the left and larger instances on the right.

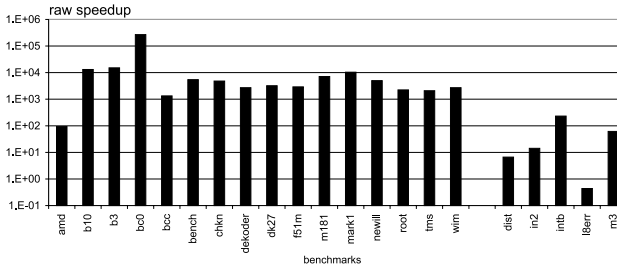

Figure 8: Raw speedups for logic minimization instances compared to Espresso.

Figure 9 shows the raw accelerator speedups for Steiner triple systems. For STS_{27} we can report the speedup compared to the reference solver in [11].

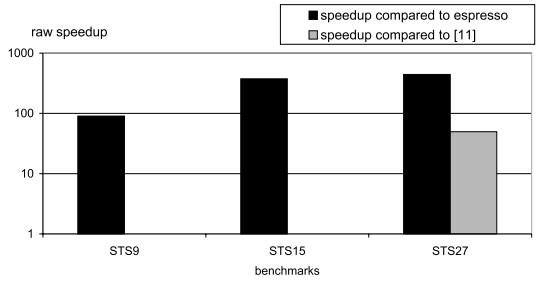

Figure 9: Raw speedups for Steiner triple systems compared to Espresso and, for STS_{27}, to the solver presented in [11].

Relative raw speedups Table 1 shows the speedups for the different architectures normalized to the basic architecture CE. The relative speedups have been averaged over

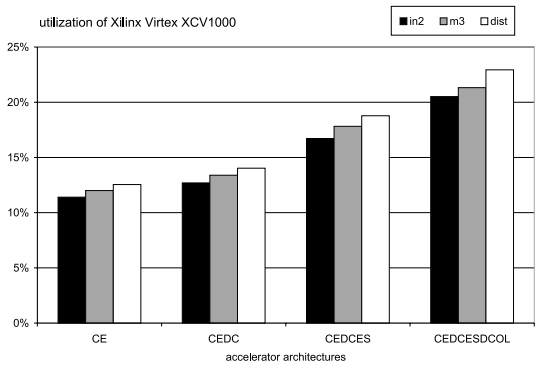

Figure 10: Hardware cost for the different architectures.

all instances of the corresponding benchmark classes. The software reference solver is Espresso. For each architecture the best settings for the heuristics have been used.

architecture	problem class	
	logic minimization	STS
CE	1	1
CEDC	1.14	1.03
CEDCES	3.02	3.78
CEDCESDCOL	4.07	3.78

Table 1: Relative speedups compared to the CE architecture for logic minimization and STS instances.

For logic minimization instances, there is an additional speedup associated with each additional reduction. STS instances, however, make insignificant use of don't cares. They achieve an additional speedup from essentials. Dominated columns again do not provide any improvement over the CEDCES architecture.

4.3.3 Hardware cost

Figure 10 expresses hardware cost for three larger Espresso problems relatively to the capacity of the Virtex XCV1000.

To investigate the relation between performance and required hardware for the different reductions, we have simulated and synthesized accelerators for three of the larger Espresso problems. Figure 11 shows that each additional reduction decreases the required number of cycles by roughly one order of magnitude. The hardware cost increases moderately, with the best performing architecture requiring only 79 % more hardware resources than simple CE.

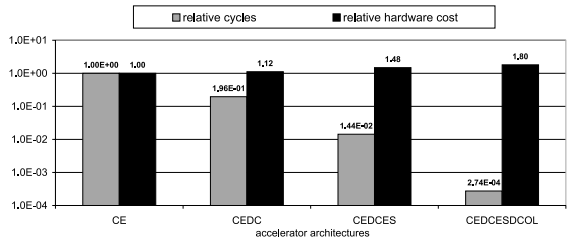

Figure 11: Relative speedup and relative hardware cost for the different accelerators compared to the CE architecture.

4.4 Discussion

One result from our experiments is that the performance of the reductions and heuristics is problem-class dependent. Hence, experimentation with small instances of some problem class is advisable to find the most useful combination of architectural parameters.

The raw speedups achieved are promisingly high. However, the problem instances used in our experiments are rather small and can be solved in software in the order of seconds. The times for solving the Espresso benchmarks in software range from 5ms to 1.3 s. With current FPGA synthesis times in the order of minutes, we cannot expect to achieve overall speedups for these small problems. Our accelerators certainly target larger problem instances for which more experimentation is needed. Further, we share a major requirement with all instance-specific custom computing machines: reducing hardware compilation times. Promising approaches in this area include custom compilation tools and partial reconfiguration.

Another observation from the experiments is that the accelerator performance decreases for larger problems from logic minimization (see Figure 8). The reason is that the accelerators lack in the algorithmic sophistication of software solvers. Consequently, further accelerator reductions, such as row dominance, should be investigated. The biggest improvement, however, can be expected by incooperating better cost bounds which is discussed in the next section.

5 Hybrid Solver Concept

To combine the raw speed of the reconfigurable accelerator with more sophisticated and problem class specific bounding schemes, we propose the hybrid solver architecture shown in Figure 12. An accelerator circuit in reconfigurable hardware communicates via messages with a software task. The accelerator scans the search space S. When it hits the root variable of some subspace S' at

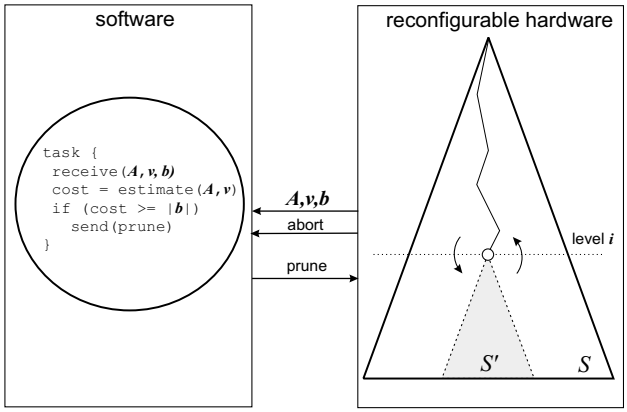

Figure 12: Hybrid solver architecture.

search level i, the current variable vector and the current best cost vector are sent to the task. The task computes a pruning condition by estimating a cost bound. If S' is to be pruned off, the task sends a *prune* message to the accelerator. The accelerator stops searching S' and returns to level i to change value assignments. This represents a form of non-chronological backtracking. Should the search of S' complete before the task results with a cost bound, the accelerator sends an *abort* message to kill the task.

We are currently working on a simulation to evaluate the hybrid solver. A challenging issue is the selection of the search level that triggers the bounding task. Possible extensions include several trigger levels as well as dynamically determined trigger levels. Dynamic levels allow for adaption which could result in better overall resource utilization as the solver makes more often use of bounds in hard subregions of the search space and relies more on pure accelerator speed in other subregions.

We envision a prototype that consists of a processor tightly coupled with a Xilinx Virtex FPGA. The processor has full control over FPGA reconfiguration. The accelerator state, i.e., variable values and current best cost, is transferred to the task by FPGA readback. Partial configuration restores the state of the accelerator to continue search at level i.

6 Conclusion

In this paper, we have presented instance-specific custom computing machines for the set covering problem. We have discussed the design of four accelerator architectures that implement a branch & bound algorithm and differ in their reduction techniques. Experiments with benchmarks from logic minimization and Steiner triple systems have been used to evaluate the accelerators. Although we have achieved raw speedups in the order of four magnitudes on average, two lines of work remain to make set covering accelerators practically applicable: Experimentation with larger benchmarks and reducing hardware compilation times.

References

[1] Miron Abramovici and Jose T. De Sousa. A SAT Solver Using Reconfigurable Hardware and Virtual Logic. *Journal of Automated Reasoning*, 24(1-2):5–36, 2000.

[2] Jonathan Babb, Matthew Frank, and Anant Agarwal. Solving graph problems with dynamic computation structures. In *Proceedings of SPIE: High-Speed Computing, Digital Signal Processing, and Filtering Using Reconfigurable Logic*, volume 2914, pages 225–236, 1996.

[3] O. Coudert. On Solving Binate Covering Problems. In *Proceedings of the IEEE/ACM Design Automation Conference*, pages 197–202, June 1996.

[4] Giovanni De Micheli. *Synthesis and Optimization of Digital Circuits*. McGrawHill, 1994.

[5] Farzan Fallah, Stan Liao, and Srinivas Devadas. Solving Covering Problems Using LPR-Based Lower Bounds. *IEEE Transactions on VLSI Systems*, 8(1):9–17, 2000.

[6] M.R. Garey and D.S. Johnson. *Computer and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, New York, 1979.

[7] Jun Gu, Paul W. Purdom, John Franco, and Benjamin W. Wah. Algorithms for the Satisfiability (SAT) Problem: A Survey. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 35:19–151, 1997.

[8] Zineb Habbas, Francine Herrmann, Daniel Singer, and Michael Krajecki. Methodological approach to implement CSP on FPGA. In *10th International Workshop on Rapid System Prototyping*, pages 66–71, 1999.

[9] 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)*, pages 209–221. Springer, 1997.

[10] P.H.W. Leong, C.W. Sham, W.C. Wong, W.S. Yuen, and M.P. Leong. A Bitstream Reconfigurable FPGA Implementation of the WSAT Algorithm. *IEEE Transactions on VLSI Systems*, 9(1):197–201, February 2001.

[11] Carlo Mannino and Antonio Sassano. Solving hard set covering problems. *Operations Research Letters*, 18:1–5, 1995.

[12] Marco Platzner. Reconfigurable accelerators for combinatorial problems. *IEEE Computer*, 33(4):58–60, April 2000.

[13] Christian Plessl and Marco Platzner. Instance-Specific Accelerators for Minimum Covering. In *Proceedings of the 1st International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, pages 85–91. CSREA Press, 2001.

[14] Steven Skiena. *The Algorithm Design Manual*. Springer, 1998.

[15] T. Suyama, M. Yokoo, H. Sawada, and A. Nagoya. Solving Satisfiability Problems Using Reconfigurable Computing. *IEEE Transactions on VLSI Systems*, 9(1):109–116, February 2001.

[16] PX. 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.