Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Christian Plessl

Reconfigurable Accelerators for Minimum Covering Problems

Diploma Thesis DA-2001.10
Winter Term 2000/2001

Tutor: Dr. Marco Platzner

Supervisor:
Prof. Dr. Lothar Thiele

15.3.2001

Institut für Technische Informatik und Kommunikationsnetze
Computer Engineering and Networks Laboratory

Contents

List of Figures

List of Tables

Abstract

In this report the design and implementation of an instance-specific accelerator for solving minimum covering problems will be presented. After an introduction to configurable computing in general, the minimum covering problem is defined and a branch and bound algorithm to solve it in software is presented.

The remainder of the report shows how this branch and bound algorithm can be adopted to hardware. Specifically it is stressed how the various sophisticated strategies for deducing conditions for variables used by software solvers can be adopted to hardware and how a system which uses 3-valued logic to solve this problem can be designed. In addition to these considerations focusing on the architecture of the system, some important details of the actual implementation are given.

A prototype has been implemented for showing the feasibility of the concept and for gaining information about speed and size of the hardware implementation. Cycle-accurate simulations for a set of benchmark problems have been done for determining the performance of the accelerator.

The speed of the resulting accelerators has been compared to the time a reference software solver (espresso) needs and the resulting speedups have been calculated. I have shown that a raw speedup of several orders of maginitude can be achieved for many problems; for some problems no speedup is achieved yet. After a discussion of the results, ideas for future work are presented.

Zusammenfassung

In dieser Diplomarbeit wird ein instanz-spezifischer Hardware-Beschleuniger zur Lösung von Minimum Covering Problemen vorgestellt. Nach einer kurzen Einführung ins Gebiet Reconfigurable Computing im allgemeinen wird das Minimum Covering Problem definiert und Algorithmen zu dessen Lösung mittels Branch and Bound vorgestellt.

Im weiteren wird aufgezeigt, wie der Branch and Bound Algorithmus auf Hardware übertragen werden kann und wie die verschiedenen, raffinierten Methoden zur Reduktion des Problems umgesetzt werden. Eine Implementation unter Verwendung von 3-wertiger Logik wird vorgestellt. Zusätzlich zu den Betrachtungen bezüglich der Architektur des Systems werden auch einige Details der Implementierung aufgezeigt.

Um die Funktionsfähigkeit des Systems zu demonstrieren und um Informationen über das Mass der Beschleunigung zu erhalten, wurde ein Prototyp implementiert. Um die Performance abschätzen zu können, wurden zyklusgetreue Simulationen des Beschleunigers für eine Menge von Benchmark Problemen durchgeführt.

Die Geschwindigkeit der resultierenden Beschleuniger wurde mit der Laufzeit des Referenz-Solvers (Espresso) für die Lösung dieser Benchmarks auf einer Workstation verglichen. Es konnte gezeigt werden, dass für viele Probleme (unter Vernachlässigung der Synthesezeit) Beschleunigungen von mehreren Grössenordnungen erzielt werden können; hingegen wurde bei gewissen Problemen noch keine Beschleunigung erreicht.

Nach einer Diskussion der Resultate werden Ideen für zukünftige Arbeiten präsentiert.

Problem Task

This chapter describes the problem task for my diploma thesis as given.

Background

A Field-Programmable Gate Array (FPGA) is built of a large number of cells which contain combinational logic blocks implemented by lookup tables, multiplexors and registers. These fine-grained computing structures are well suited to implement algorithms that exploit massive parallelism on the bit level or map to highly pipelined architectures.

Many combinatorial problems show a high degree of parallelism using data types with small bit-widths. This would make combinatorial problems good candidates for FPGA implementation. On the other hand, FPGA implementations for combinatorial problems are challenging for two reasons. First, combinatorial problems are typically solved with control-flow oriented search methods, such as backtracking or branch & bound. Second, the actual degree of parallelism that can be exploited depends on the problem instance.

Recently, architectures have been presented that implement search algorithms in reconfigurable hardware and massively exploit fine-grained parallelism by instance-specific circuit compilation. Up to now, the problem investigated best is the Boolean satisfiability problem (SAT) [GPFW97]. SAT is a fundamental problem in mathematical logic and computing theory with many practical applications in computer-aided design, automated reasoning, machine vision, etc. Since the general SAT is NP-complete, exact methods to solve SAT have an exponential worst-case complexity. This limits the applicability of exact SAT solvers in many areas.

Examples for reconfigurable accelerators can be found in [Pla00] [PM98] [ZMAM99] [ZMA00] [SYS96] [ADS00]. Currently, there are several open problems in this area:

- Reconfigurable accelerators should be developed for a wider range of difficult problems and embedded into real-world applications, e.g., into computer-aided design tools.
- Speed-ups achieved by reconfigurable accelerators strongly depend on the benchmark class. Practical accelerators should combine solvers in software and reconfigurable hardware to deliver sound and reasonable performance over many benchmark classes.
- Novel compilation/synthesis frameworks for instance-specific accelerators must be developed. To be able to compete with software solvers, instance-specific accelerators require an extremely fast compilation process as well as high-performance circuits. Compilers and synthesis tools are needed that address these issues.

- Problems that are too large to fit on a single FPGA device must be split into smaller subproblems that can be computed sequentially or, if several FPGAs are available, in parallel. It must be investigated whether the particular problems can be effectively partitioned into sets of smaller subproblems.

Project Goals

The goal of this project is to design, implement and evaluate a reconfigurable accelerator for *covering problems*. Unate and binate covering problems with unit or integer cost can be cast as minimum-cost satisfiability problems and appear quite frequently in computer-aided design of digital circuits [DM94]. The project splits up into different phases:

- **Analysis** of software techniques for unate and binate covering problems with unit and integer cost.
- **Design** of an accelerator architecture for covering problems. An emphasis should be put on the investigation of different reconfiguration techniques, such as compile-time reconfiguration and various forms of instance-specific compilation.
- **Implementation** of an accelerator prototype on a PC-based reconfigurable system. The systems consists of a PCI carrier board that can be equipped with FPGA and DSP modules. The FPGA modules contain an FPGA of type Xilinx Virtex XCV1000 and memory.
- Selection of appropriate benchmark problems and **experimental evaluation** of the accelerator.

1 Introduction

1.1 Reconfigurable Computing

1.1.1 Motivation

In the last twenty years computers have made huge technical advances. The concept of the simple microprocessor has evolved to very sophisticated CPU architectures and computer systems. Performance doubles in less than two years, while the cost per performance ration is decreasing rapidly.

This development did not lead to a saturation of the demand for computing power, in contrary: the newly available computing power was the basis for new technologies as digital signal processing, mobile computing and communication. Algorithms which were considered impractical in the past because of algorithmic complexity are widely used today, such as real time audio or video data decompression, handwriting recognition or advanced modulation techniques in digital cell phones.

In spite of rapidly increasing computing power, there are still a lot of algorithms, that are very challenging to implement on today's general purpose CPUs. For some application fields specific CPU architectures have been developed to overcome some shortcomings of general purpose CPUs, for instance digital signal processors (DSPs).

To achieve highest performance for specific applications it is common today to develop Application Specific Integrated Circuits (ASICs). Since today's tools for designing an ASIC are increasingly advanced and manufacturing of ASICs using standard-cell processes gets cheaper, using of ASICs has become more attractive for a broader spectrum of applications. Today ASICs are very popular for high-volume products with high computing demand.

Basically, using ASICs leads always to the fastest possible implementation for any computing problem, since the circuit is completely problem adapted. The serious drawback of using ASICs is the inflexibility as ASICs are usable for one single application.

The advent of high-density, programmable logic devices, namely field-programmable gate arrays (FPGAs) has led to a new idea: Reconfigurable Computing. FPGAs can be thought of successors of simple programmable logic devices like GALs or PALs that were used for many years to implement boolean functions in a programmable device. FPGAs can be thought of a device having a collection of configurable logical blocks that are connected by a programmable interconnection network. Todays high-density FPGAs are large enough to implement the same circuit as smaller ASICs. Typically FPGAs are SRAM-cell based and the configuration can be changed very fast (some milliseconds) by writing a configuration bitstream to their configuration memory.

Reconfigurable logic devices are very popular and are currently the fastest

growing segment of the semiconductor market. The most frequent use of FPGAs today is ASIC replacement. The possibility to change the hardware by reprograming is a convenient mean for doing hardware upgrades in the case of design errors oder specification changes. The ability to change the configuration of the FPGA very fast is not of any importance in this application.

From a different point of view, FPGAs open up a completely new application domain, *Reconfigurable Computing*.

1.1.2 Hardware Accelerators

In some way, reconfigurable computing means bringing the software to the hardware. The basic idea is to make explicit use of the reconfigurability of the hardware. This means that we can change the way we build computing systems: Instead of writing software for a microprocessor or designing an ASIC, we generate a circuit exactly for the task needed and download it to an FPGA. After downloading, the circuit behaves just as an ASIC which was optimized to this specific task. When we need a different task, a different configuration is downloaded to the FPGA. Reconfigurable computing allows to combine the performance of ASICs with the flexibility of software.

A promising idea is to use FPGAs as a kind of coprocessors for computing-intensive algorithms. For that purpose, a circuit adapted to the problem is generated and downloaded to the FPGA, i.e. we use the FPGA as *Hardware Accelerators*. For example consider an accelerator for video decompression using a PCI extension card in a PC. Since there are several compression standards which shall be supported, a set of predefined configurations for video decompression are defined, one configuration per algorithm. When using the accelerator, the suitable configuration is downloaded and the PCI card is used to do all the video decompression taking this burden from the CPU.

It is important to distinguish two kind of accelerators: *problem specific accelerators* and *instance specific accelerators*. Problem specific accelerators implement a general circuit that is able to solve lots of instances of a given problem. Instance specific accelerators are customized to solve exactly one single instance of a given problem. The difference can be illustrated with a simple example, a prime number checker, which shall determine, whether a given number is a prime number. There are two implementation variants to consider: We could implement a generic prime number checker circuit, which can check whether *any given number* that is passed to the circuit is a prime number. This is called “problem specific configuration”. The alternative is to generate a different circuit for every instance of the problem, that is if we want to check whether the number 247 is a prime number, a circuit which solves exactly this instance of the problem is generated. This kind of configuration is called “instance specific configuration”. The resulting circuit is not usable to check whether any other number is prime or not.

Which kind of accelerator configuration is useful depends largely on the application.

There are a many ideas on how to integrate reconfigurable hardware with general purpose CPUs ranging from making the whole CPU or parts of the data-path reconfigurable, up to using FPGAs as coprocessors coupled via an IO-bus. Current research projects focus on how this coupling of reconfigurable accelerators and general purpose CPUs can be done and whether problem or instance specific configurations are used.

While the idea of using hardware accelerators looks quit obvious, there are also some serious limitations that have to be considered: A challenge to deal with if instance specific configuration is used is that the time to create a configuration can be quite long. Synthesis tools for digital circuits are traditionally optimized for generating small and fast circuits but not for short synthesis times. Therefore when new configurations have to be created frequently attention has to be payed to synthesis time. High-density FPGAs are quite expensive. This prevents the use of these devices in low-cost systems today but this is expected to change with rapidly decreasing prices for semiconductors.

1.2 FPGA

1.2.1 Overview

FPGAs are todays most complex and powerful programmable logic devices. Since in this thesis Xilinx Virtex FPGAs were used, the following paragraph will explain the structure of a Virtex FPGA in more detail.

The basic building block of an FPGA is the Configurable Logical Block (CLB). The desired boolean function is not implemented by interconnection of dedicated logical gates, but by using lookup-tables, which are in principle RAM implementations of truth-tables. This is a distinctive feature of FPGAs compared to their ancestors, Programmable Array Logic (PAL) and Generic Array Logic (GAL) devices which used programmable wired-and and wired-or planes to implement logic equations.

For the implementation of sequential circuits the CLBs also implement D-style-flipflops. For ease of implementation of fast adders, each CLB does provide dedicated carry logic. The terminology for the naming of the CLBs is somewhat confusing at the first sight: Each CLB consists of 2 Slices, each Slice consists of 2 4-input LUTs, 2 D-flip-flops and 2 carry&control blocks. Each slice can implement any function of 6 variables or 2 functions of 5 variables. For a simplified schematic of a Xilinx Virtex CLB see figure 1.

The CLBs are arranged in a rectangular matrix, in this thesis we used a Virtex XCV1000, which has a matrix of $64 \star 96$ CLBs. The CLBs are surrounded by IO-Blocks which are configurable drivers or input buffers for IO-Signals. For the efficient implementation of RAM structures Virtex

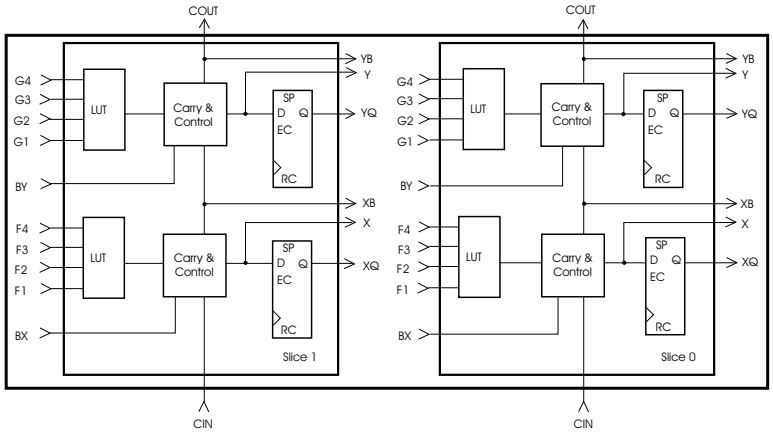

Figure 1: Xilinx Virtex CLB (Source: Xilinx)

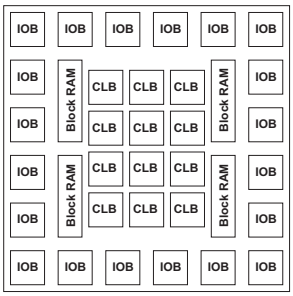

Figure 2: Xilinx Virtex FPGA

FPGAs contain fast internal Block RAM. Figure 2 shows the structure of a Virtex FPGA. The CLBs are connected by an interconnect network. Each CLB is connected directly to its neighbors with distance 1, 2 and 6. Additionally there are so-called "longlines" which allow a fast connection of all CLBs in a row or column respectively.

For the design of FPGA circuits, similar oder even the same synthesis tools as for ASIC design are used. The tools map the circuit description in VHDL or Verilog to the available FPGA elements and generate a configuration bitstream. This bitstream has to be downloaded to the FPGA at power up.

1.2.2 Application of FPGAs for accelerators

Since it is possible to reprogram FPGAs very fast (in the order of some milliseconds) and todays high-density FPGAs provide more the 1 million gate-equivalents, FPGAs have been proven to be suited very well for building

hardware accelerators. However, the structure of FPGAs imposes restrictions on the kind of algorithms that are suitable for FPGA implementation.

The main restriction results from the fact that the structure of an FPGA is very fine-granular, i.e. the circuit is laid out to operate on bit level using logical equations rather than operating on numerical data flow oriented operations. It is hard to get an efficient implementation of complex arithmetical operations, e.g. multipliers for larger bit-widths. However, it has been shown, that numerous digital signal processing algorithms can also be implemented on FPGAs very efficiently with architectures that make explicit use of the characteristics of FPGAs. The other difficulty is with algorithms that need a lot of temporary memory or even worse, recursive algorithms. Since using LUTs as memory cells is very inefficient Xilinx integrated dedicated memory cells into Virtex devices. However, as this RAM is rather small, external RAM might be necessary.

The real advantage of FPGAs is the possibility to build circuits that consist of parts working in parallel. This parallelism is the key to high performance. Although state-of-the-art CPUs might run at clock rates of several hundred MHz, FPGAs running at much lower clock rates may outperform them by using an amount of parallelism that a general purpose CPU cannot achieve.

In summary, FPGAs are most efficient for accelerating algorithms that are *highly parallel, not recursive and have little memory requirements.*

Furthermore FPGAs can be particularly interesting when implementing algorithms that can be described as many small statemachines. If a complex statemachine is implemented on a CPU, a significant amount of time may elapse in evaluating which state transition to perform because the checking of all conditions and acting upon the state-transition has to be done sequentially. An FPGA implementation can check all conditions and maybe even execute the associated actions in parallel.

As will be outlined further in this thesis, this accelerator for minimum covering problems can make good use of the FPGA benefits as it bases on parallel evaluation of large logical equations and is strongly control-flow oriented.

1.3 Boolean Satisfiability (SAT) and Minimum Covering

1.3.1 SAT

Definition 1 (Boolean Satisfiability (SAT)) *The Boolean Satisfiability Problem is a fundamental problem in theoretical computer science and can be formulated as follows [GPFW97], given:*

- *a set of n boolean variables:* $x_1, x_2, \ldots x_n$,

- *a set of literals, where each literal is a variable x_i or the complement of a variable $\bar{x}_i$, and*
- *a set of m clauses $C_1, C_2, \ldots, C_m$ where each clause consist of arbitrary literals combined by the logical* or *connective '+'.*

Determine, whether there exists an assignment of truth values to the variables that makes the Conjunctive Normal Form (CNF)

$$C_1 \cdot C_2 \cdot \ldots C_m \tag{1}$$

true, where '$\cdot$' denotes the logical and *connective. This form of equation is sometimes called* product of sums *form.*

□

For example, given the two equations

$$(x_1 + x_2 + x_3) \cdot (\bar{x}_2 + x_4) \cdot (x_4 + x_3) \tag{2}$$

and

$$\bar{x}_1 \cdot (x_1 + x_2) \cdot \bar{x}_2 \tag{3}$$

Equation 2 *is satisfiable*, since assigning x_1 and x_3 to true and assigning x_2 to false results in the equation evaluating to true. Note that this assignment is not the only solution, assigning x_1 and x_4 to true is another solution. SAT is a decision problem: either an equation is satisfiable or not. If an equation is satisfiable the solution does not need to be unique; there may be many solutions.

A remarkable property of SAT is that problems may have not only several solutions, but the solution can be determined by a subset of all variables. For instance, when assigning x_1 and x_4 to true, the value of x_3 doesn't matter, i.e. x_3 has become a don't care variable.

In contrary, equation 3 is *not* satisfiable, since there is no assignment of values to x_1 and x_2 that could satisfy the equation.

The importance of SAT is for two reasons: Firstly, it is important in theoretical computer science, because SAT was the first problem, that was proven to be NP-complete. The proof of NP-completeness of other problems is done by reduction from SAT. Secondly, SAT is also important in practice as it has many applications for instance in CAD for electronic circuits, graph coloring, scheduling and many more.

As we can see from equation 2 the first solution requires that a value is assigned to 3 variables whereas the second requires only 2 variables to be assigned. This leads to the problem of finding the solution that needs the smallest number of assignments to variables. When we introduce the notion of cost of variable assignments, i.e. each variable assigned to true has

costs of 1 and all other variables have costs of 0, we call this a *Minimum-Cost Satisfiability Problem*. This problem is closely related to the *Minimum Covering Problem*.

1.3.2 Minimum Covering Problem

The "minimum-cost satisfiability problem" as defined in [DM94] can be regarded as an extension of the SAT problem.

Definition 1 (Minimum Cost SAT Problem) *Given is*

- *a set of variables $x \in \mathcal{B}^n$ (where $\mathcal{B}$ is the set of boolean values $\{0, 1\}$) and*
- *a boolean equation in Conjunctive Normal Form (CNF).*

The "minimum-cost satisfiability problem" is to find a solution which satisfies the clauses while having the minimum cost, where the cost is a weighted sum of the variables. The cost function can thus be written as $\mathbf{c}^T\mathbf{x}$ where $\mathbf{c}$ is the weight vector.

□

In contrary to the SAT problem which is a decision problem, the minimum-cost SAT problem is an optimization problem. As in the case of SAT, the solution of minimum-cost SAT has not to be unique. There may be several assignments to the variables that have the same cost.

In [DM94] De Micheli uses a slightly different view on the same problem. He formulates the minimum cost SAT problem as *minimum covering problem*, which can be regarded as an alternative representation. This representation operates rather on matrices than on boolean equations and is the base for the minimum covering solver used in the 2-level logic minimization tool *espresso*. The basic idea is to make an analogy between set theory and boolean algebra and identify finding a satisfying variable assignment with finding a set cover.

Definition 1 (Minimum Covering Problem (MinCov)) *Given is a Collection C of subsets of a finite set S. The minimum covering problem is the search for the minimum number of subsets that cover S. That is, the conjunctive of all subsets is equivalent to the set S.*

□

The elements of C are called groups to distinguish them from the elements of S.

Covering can be modeled and illustrated using a matrix representation. When $\mathbf{A} \in \mathcal{B}^{m \times n}$, where the set of rows corresponds to S ($m = |S|$) and

the columns correspond with C ($n = |C|$), then *a cover corresponds to a subset of columns, having at least a 1-entry in all rows of A* or more formal a cover corresponds to selecting $x \in \mathcal{B}^n$, such that $\mathbf{Ax} \geq \mathbf{1}$. A minimum weighted cover corresponds to selecting $x \in \mathcal{B}^n$, such that $\mathbf{Ax} \geq \mathbf{1}$ and $\mathbf{c}^T\mathbf{x}$ is minimum.

Consider the following example, figure 3 (a) shows a hypergraph, with 5 vertices $\{v_i, i = 1, 2, \ldots, 5\}$ and five edges $\{a, b, c, d, e\}$. This graph can be described with the following vertex/edge incidence matrix $\mathbf{A}$:

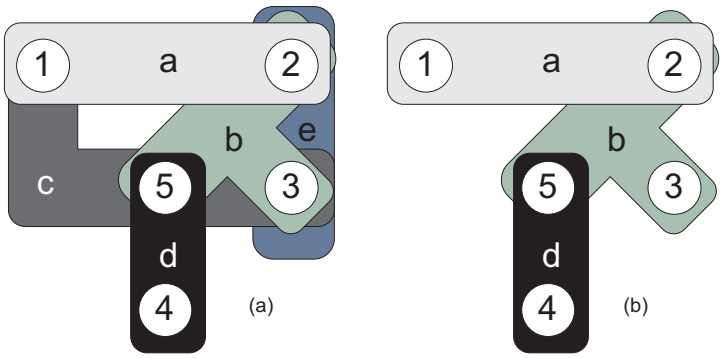

Figure 3: Minimum Covering of hypergraph

$$\mathbf{A} = \begin{bmatrix} 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 \end{bmatrix}$$

We are looking now for the minimum number of edges that cover all vertices of this graph. Thus the set of all vertices corresponds to S and the edges correspond to C (columns of $\mathbf{A}$). All edges have the same cost, thus weight vector $\mathbf{c} = (1, 1, 1, 1, 1)$. The minimum set of edges, which cover all vertices is a, b and d, in other words, $\mathbf{x} = (1, 1, 0, 1, 0)$ is a minimum cover.

Relationship SAT / Covering Covering problems can be regarded as minimum-cost SAT problems by identifying the whole CNF with S, the individual clauses of the CNF with the elements of S and the elements of C with the boolean variables.

Applying this to our previous example "covering of a hypergraph", we can formulate matrix $\mathbf{A}$ as CNF:

$$\Phi(x) = (x_1 + x_3) \cdot (x_1 + x_2 + x_5) \cdot (x_2 + x_3 + x_5) \cdot x_4 \cdot (x_2 + x_3 + x_4)$$

The boolean variables $x_1, \ldots, x_5$ represent the edges of the hypergraph and the individual clauses denote conditions that the respective vertex is

covered. For instance, the first clause $(x_1 + x_3)$ denotes that vertex v_1 can be covered by edge a oder c. If the CNF evaluates to true the graph is covered.

Note that when transforming this kind of covering problems into a corresponding SAT problem, there is a major difference: The CNFs corresponding to the covering problems are always unate-boolean expressions, where *unate* means that all variables appear either all non-inverted or all inverted. In this specific application only non-inverted variables appear. Therefore these kind of covering problems are called *unate covering*. The consequence of the equation being unate is that selecting all subsets C of set S results always in a cover albeit not necessarily a minimum cover.

When extending the problem, that the selection of group a can imply the selection of a different group b, a clause of the form $(\bar{a}+b)$ is added. This makes the equation binate since literals are now appearing in non-inverted and in inverted form. This variant of a covering problem (covering with implications) is called *binate covering*.

When extending our previous example so that selection of edge a shall imply the selection of edge b, the clause $(\bar{x}_1 + x_2)$ has to be added:

$$(x_1 + x_3)(x_1 + x_2 + x_5)(x_2 + x_3 + x_5)(x_4)(x_2 + x_3 + x_4)(\bar{x}_1 + x_2)$$

Evaluating the equation we see that $\mathbf{x} = (1, 1, 0, 1, 0)$ is a cover whereas $\mathbf{x} = (1, 0, 1, 1, 0)$ is not a cover.

Binate covering can still be represented by using matrices, elements corresponding to negated variables have simply a -1-entry in the column which includes the complemented variable, i.e. matrix $\mathbf{A}$ with this clause added is:

$$\mathbf{A} = \begin{bmatrix} 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ -1 & 1 & 0 & 0 & 0 \end{bmatrix}$$

The determination of a valid cover using the matrix has to be modified slightly for the binate problem: 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. It can be seen, that $\mathbf{x} = (1, 1, 0, 1, 0)$ since every row contains at least one 1. $\mathbf{x} = (1, 0, 1, 1, 0)$ is not a cover since these rows doesn't have a 1-entry and the complementary column-set (columns 2 and 5) do not have a -1 in row 6.

Numeric tractability of SAT and Minimum Covering SAT is NP-complete, i.e. the time to solve a SAT problem exactly grows exponentially

for any known algorithm. This means, that large SAT problems often cannot be solved by exact algorithms.

There are other algorithms, that use heuristics and stochastic search algorithms for solving SAT. These *inexact* algorithms cannot guarantee that a solution is found even if there exists a solution, but they allow to target much larger problems because they scale better with problem size. It depends on the application whether an inexact algorithm is useful or not.

1.4 Thesis outline

This thesis will present an architecture for solving Minimum Covering Problems in hardware.

Since the architecture is inspired by software algorithms for solving SAT and Covering, the basic ideas of software algorithms are introduced in Chapter 1.5. After that, an overview of related research activities in this field is given.

Chapter 4 focuses on the design and architectural aspects of the accelerator. The chapter gives an outline of the complete system, shows how a particular problem is processed to obtain an FPGA design and how the problem is solved by the FPGA. Furthermore the architecture of the accelerator is explained from a behavioral point of view. This shows how an efficient architecture of a hardware accelerator can look like, how we can make use of the specific properties of hardware and how the building blocks are working.

In Chapter 5 the focus moves from the more general architectural considerations to the actual implementation, by giving an overview of the VHDL implementation files and discussing some building blocks in more detail. The code-generator and the tools needed for reconfiguring the prototyping board and read-back of the results are introduced.

The evaluation of the design is presented in Chapter 6, by explaining the test methodology used, i.e. how the performance of the design was tested, what benchmark problems were chosen and what results have been achieved. The measurements are compared to a software solver implementation.

This leads to the Conclusion on what has been achieved in Chapter 7 and an outlook to future work in Chapter 8.

1.5 Acknowledgments

The project was not only instructive as it was my first contact with real research activities in the interesting field of reconfigurable computing, it was also a lot of fun.

I wish to express my most sincere thanks to Marco Platzner for being my supervisor for this work. The introduction to the topic of instance specific accelerators and the valuable discussions on his work on accelerators for

boolean satisfiability, as well as motivating hints and comments on my work made were of great help.

Projects like this require quite a lot of costly infrastructure. Many thanks to the Computer Engineering Group of the Computer Engineering and Networks Lab (TIK), particularly to Prof. Dr. Lothar Thiele for making this work possible and being my supervising professor.

2 Algorithms for solving SAT and covering

Algorithms for solving SAT and covering can be divided in 2 classes: *complete (exact) algorithms* and *incomplete (inexact) algorithms*. Complete algorithms can be thought of algorithms that scan all the search space for a solution. The classical algorithm for exact solving SAT problems is the *David-Putnam* algorithm. The David-Putnam algorithm solves the problem by recursively simplifying CNF by detecting conditions for variables, splitting up the problem into independent sub-problems and assigning variables.

SAT and Covering are NP-complete problems which means that solving large problems is computational infeasible when using exact algorithms. Therefore procedures called *Stochastic Local Search Algorithms* are used. These incomplete algorithms cannot guarantee that a solution is found (at least in finite time), but using random search and heuristics can often lead to a solution that could not be solved using complete algorithms.

2.1 Complete algorithms

Branch and bound algorithms Branch and bound algorithms are very general algorithms for problems that can be solved iteratively which is the case for both SAT and Covering. Suppose we want to solve a SAT problem, which is given by n boolean variables and a boolean equation. The problem whether a solution exists can be solved by enumerating all possible assignments to these variables and check for each assignment whether this is a solution.

A slightly different view on this enumeration procedure leads to a tree-like presentation of the problem: When assigning a value to a variable, the problem is reduced by one variable. Assigning a value to a variable splits up the problem into 2 subproblems: one problem with this variable set to 1, the other subproblem with this variable assigned to 0. This systematic testing of all solutions can be represented as a tree, which is in this case a binary tree. Searching for a solution can be thought of traversing a *decision-tree* which assigns one more variable at each step on the path from the root to the leaves. The leaves represent all possible solutions.

An algorithm that searches all the leaves of the tree for a solution has exponential complexity in the worst and average case. The idea of a branch and bound algorithm is to reduce this complexity by not visiting all the leaves. When walking stepwise from the root to the leaves, the algorithm makes at each inner node a local decision whether this path can finally lead to a solution at all. If the path can lead to a solution, the traverse to the leaf is continued and the problem splits up into subproblems. This is called *branch*. If the path cannot lead to a solution of the problem, i.e. a contradiction was found, this branch of the tree is pruned, which is called *bound*.

The worst case complexity for branch and bound algorithms is still exponential, but for the average case, the complexity *can* be much better. The gain of using branch and bound instead of simply enumerating the solutions depends largely on the bound-conditions used. The earlier branches can be pruned, the faster a solution can be found because of the shrinking search space.

Solving SAT with branch and bound Branch and bound algorithms can be used to solve SAT. While the branch maps very naturally to assigning a variable either 1 or 0 the bound condition looks more complex. The fundamental idea for the bound condition is to prune the tree if the current branch cannot lead to a solution anymore, i.e. if the variables already assigned make the equation not satisfiable for every variable assignment of the other variables (contradiction). This can be tested for instance by looking at the CNF after every variable assignment. As soon as one single clause of the CNF is not satisfied, the complete CNF is no longer satisfiable.

Solving Minimum Cover with branch and bound Minimum Cover can be solved by extending the bound condition of the procedure of solving SAT. When traversing the path from root to the leaves, at each step the cost of the current variable assignment and whether it satisfies the equation whether it can lead to a solution in future is evaluated. A recursive scheme that applies a branch and bound algorithm to solve minimum cover is given in algorithm 1.

The algorithm works as follows: It starts with a given variable assignment (which is empty at the beginning). If the current assignment is a solution and the cost of this solution is less than the best cost that has been found yet, the current assignment is saved as best solution. If the current assignment *is* a solution but not the best solution it is discarded.

If the assignment is not a solution, but there are no more variables to assign, i.e. the current node is a leaf, the algorithm goes back to the previous (ancestor) node. If a contradiction the alogrithm also tracks back one level. After checking these rather obvious bounds a more complex bound is evaluated. A lower bound for the additional cost that could lead to a solution is computed. If the current cost plus this minimal additional cost for a solution exceeds the best cost found, the branch is pruned as well. The efficient computation of a lower bound on the additional cost for a solution is the key for fast search.

If no bound condition applies a branch is done by selecting another unassigned variable and the branch and bound function is called recursively for this variable assigned to both 0 and 1.

Algorithm 1 Solving Minimum Covering with branch and bound (basic idea)

```
1: branch_and_bound_recursive(crt_asgm) {
   {bst_cost global variable which saves the cost of the best solution bst_asgm
   that has been found yet}
2: crt_cost := calc_crt_cost()
3: lower_bnd := calc_lower_bnd()
   {if we have a solution save solution if best ever, else discard solution and
   prune branch}
4: if (crt_asgm = solution) then
5:    if (crt_cost < bst_cost) then
6:       bst_cost := crt_cost
7:       bst_asgm := crt_asgm
8:    end if
9:    return
10: end if
   {if all variables are assigned, but crt_asgm is not a solution: backtrack}
11: if (variables_to_assign = empty) then
12:    return
13: end if
   {prune branch, since CNF is not satisfiable for any assignment of currently
   not assigned variables}
14: if (crt_asgm cannot satisfiy equation anymore) then
15:    return
16: end if
   {backtrack if crt_asgm can lead to a solution, but the cost of this solution
   is higher than bst_cst (bound)}
17: if ( (crt_cost + lower_bnd) ≥ bst_cst) then
18:    return
19: else {branch}
20:    x_i = select variable which is unassigned yet
21:    remove x_i from set of unassigned vars
22:    x_i := 1
23:    add x_i to crt_asgm
24:    branch_and_bound(crt_asgm)
25:    remove x_i form crt_asm
26:    x_i := 0
27:    add x_i to crt_asgm
28:    branch_and_bound(crt_asgm)
29: end if
30: }
```

Operations on covering matrices When working with covering matrices as described be De Micheli in [DM94] a similar branch and bound algorithm can be used. The popular two-level logic minimizer *espresso* solves *unate* covering problems and uses a variant of the algorithm presented here.

The covering problem shall be defined by matrix $\mathbf{A}$, where the rows of $\mathbf{A}$ represent the elements of the set to be covered and the columns of $\mathbf{A}$ correspond to the groups of elements.

We define the following terms: 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 this column is the only way to cover the corresponding element.

A *column a dominates* another column b if all entries of a are larger or equal than the corresponding entries of b. The dominated column can be discarded from consideration, because column b covers only a subset of the elements covered by a which means that selecting column a is preferred.

A *row r dominates* another row s if the entries of r are larger or equal than the corresponding entries of s. Dominant (domin*ant* not domin*ated*!) rows can be ignored, because any cover of the *dominated* row is also a cover of the *dominant* row.

The above reduction rules can be used to reduce the problem to a simpler form by repeatedly applying the reduction rules to problem matrix $\mathbf{A}$.

Exact_cover algorithm Algorithm 2 shows the recursive *exact_cover* method. The problem is given by matrix $\mathbf{A}$. Binary vector $\mathbf{x}$ represents the current variable assignment, $\mathbf{b}$ is the best vector found so far. The algorithm is started with arguments $\mathbf{x} = \mathbf{0}$ and $\mathbf{b} = \mathbf{1}$.

Firstly, the algorithm tries to reorder the clauses in a way that the matrix is in block-diagonal form which splits up the problem into 2 independent subproblems. After that, the algorithm tries to repeatedly reduce matrix $\mathbf{A}$ in size by applying the above reduction rules. Dominated columns, dominant rows and essential columns are iteratively removed. For any essential column the corresponding variable is set to 1 and for any dominated row the corresponding variable is set to 0.

If no more reduction is possible the bound condition *Current_estimate* is computed. Current_estimate gives a lower bound on the cost of a potential solution. An simple bound is $number\_of\_variables - variables\_assigned\_so\_far$, for a more rigid condition see [DM94].

If the bound is not taken, a branch is performed by selecting a variable, assigning values 0 and 1 to it and recursively processing the new $\widetilde{\mathbf{A}}$ matrices. Consider how the matrix changes when a variable is assigned: when assigning 1 to a variable, the corresponding column of $\widetilde{\mathbf{A}}$ is removed and all incident rows (rows that have a 1 entry in that column) are deleted because when selecting this group all the elements associated to it are covered. When

assigning 0 to a variable, the corresponding column is deleted, but no rows are deleted because no additional elements are covered.

Algorithm 2 Exact cover algorithm [DM94]

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

Example Reconsider the covering matrix of the example from chapter 1:

$$\mathbf{A} = \begin{bmatrix} 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 \end{bmatrix}$$

The second column dominates the fifth column, thus selecting column 2 covers all elements that are covered when selecting column 5. So column 5 can be dropped from consideration and x_5 is set to 0. We get matrix:

$$\mathbf{A_1} = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 \end{bmatrix}$$

The fifth row dominates the fourth row, so the fifth row can be dropped because every solution that covers v_5 (the fifth element of the set) also covers v_4, so the matrix reduces to:

$$\mathbf{A_2} = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Column 4 is essential, so x_4 is set to 1 and the fourth column and row are deleted. The resulting matrix is:

$$\mathbf{A_3} = \begin{bmatrix} 1 & 0 & 1 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \end{bmatrix}$$

This matrix cannot be reduced anymore by the reduction rules, so we have compute the lower_bound (*Current_estimate*) and branch.

2.2 Incomplete algorithms

Hoos and Stützle present an overview of many algorithms and a detailed performance comparison in [HS00]. All stochastic local search procedures for solving SAT base on algorithm 3.

The stochastic local search (SLS) procedure can be described as follows: starting with an initial variable assignment, variables of $\mathbf{x}$ are repeatedly flipped. The selection of the variable is a function of Φ and the current assignment $\mathbf{x}$. If no solution is found for maxSteps loops, a new starting value for $\mathbf{x}$ is chosen and the procedure is repeated. This can happen if the algorithm is stuck in a local minimum. The various SLS algorithms differ in how the initial assignment is found and how the variable to flip is chosen.

Since papers which suggest architectures for solving SAT using SLS in hardware are presented later in this thesis, two algorithms are discussed here: GSAT and GWSAT.

GSAT The Greedy SAT (GSAT) algorithm tries to minimize the number of unsatisfied clauses with every flip. A *score* is associated with each variable, which is calculated as the number of satisfied clauses when a variable is

Algorithm 3 General procedure for stochastic local search algorithms for SAT [HS00]

```
1: stochastic_local_search for SAT {
2: input CNF formula Φ, maxTries and maxSteps
3: output Satisfying variable assignment of Φ or "no solution found"
4: for i := 1 to maxTries do
5:    x := initAssign(Φ)
6:    for j := 1 to maxFlips do
7:       if x satisfies Φ then
8:          return x
9:       else
10:         n :=chooseVariable(Φ, x)
11:         x := x with truth value of x_n flipped
12:      end if
13:   end for
14: end for
15: }
```

inverted minus the number of clauses that are are satisfied under the current variable assignment. This means that the score of a variable is the gain in satisfied clauses if this variable is flipped.

Greedy SAT uses a simple scheme and calculates the score of the variables after each flip and flips the variable with the largest score. Hence the name *Greedy*. If several variables have the same score, one of them is randomly chosen.

A problem of GSAT is that it can easily get stuck in a local minimum, since the (hamming) distance of two variable assignments is always 1. These local minima are not easily detectable, so the restart mechanism after maxFlips unsuccessful flips might be the only way to escape from there. It has been shown experimentally that GSAT has the tendency to stick around the plateaus of local minima most of the time.

GWSAT GWSAT is an extension to GSAT with a so-called *random-walk step*. A random walk step is performed by randomly selecting an unsatisfied clause C. A random variable in C is flipped thus forcing this clause to become satisfied. The GSAT procedure is extended in a way that in each iteration of the inner loop, a flip is performed with a given probability p and a random walk is performed with probability $1 - p$. As this algorithm can escape from local minima, the outer-loop in algorithm 3 is not necessarily needed. GWSAT will find the solution of the SAT problem if existent as run-time approaches infinity.

3 Related Work

This survey of related instance-specific accelerators is grouped into architectures that solve SAT exactly, architectures that solve SAT by incomplete stochastic local search, and architectures for problems other than SAT.

Exact SAT Zhong et al. [ZAMM98] were the first to propose a reconfigurable accelerator that implements backtracking with Boolean constraint propagation [DP60] 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 [ZMAM99]. Platzner and De Micheli [PM98] presented several SAT architectures based on backtracking with 3-valued logic and don't care variables as deduction techniques. Suyama et al. [SYN99a] 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 [AS97] [ADS00] 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.

An overview of all papers that deal with the design of hardware accelerators for solving exact SAT is given in figure 4.

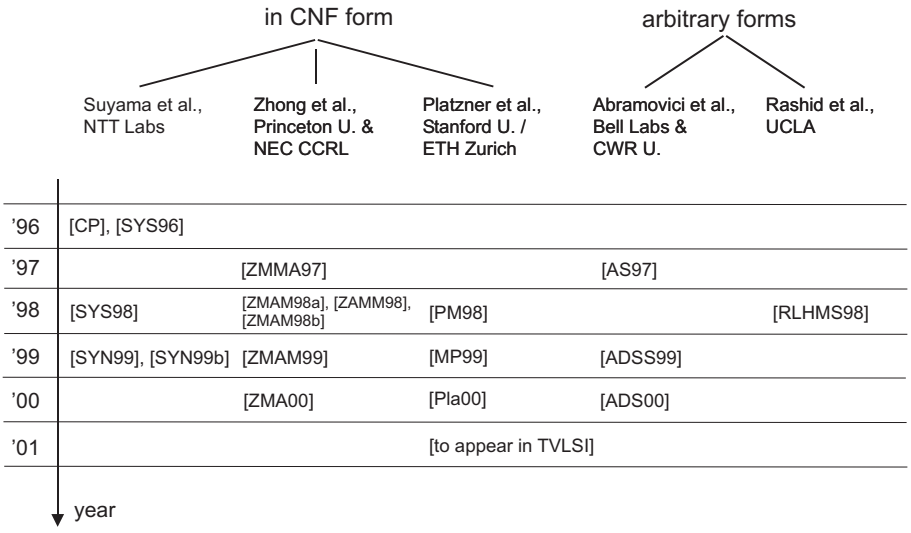

Figure 4: Related work: Exact SAT accelerators

Stochastic local search for SAT Hamadi and Merceron [HM97] 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. [YSLL99].

Others Babb et al. [BFA96] 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 Bellmann-Ford algorithm was given by Dandalis et al. [DMP99]. Other authors have discussed reconfigurable accelerators for constraint satisfaction problems (CSPs).

4 Design of Accelerator

4.1 Outline of the System

The procedure of solving a minimum-covering problem with the accelerator is as follows:

- The definition of the minimum covering problem, which is given completely by its CNF equation, is read from a file in a standardized CNF-file-format.
- The CNF-file is pre-processed by a program that interprets the file, makes certain optimizations and builds up internal data structures.
- The accelerator is generated in synthesizable VHDL. As explained in the introduction, an *instance-specific accelerator* is generated.
- The VHDL description of the accelerator is synthesized and a configuration bitstream for programming the FPGA is generated.
- The configuration bitstream is downloaded from the host to the FPGA and the circuit is executed.
- When a solution is found, the host reads back the solution from the FPGA.

4.2 Fundamental Ideas for the Accelerator

The design of the accelerator for minimum covering problems relies on 4 basic ideas:

1. Using 3-valued logic instead of regular binary boolean logic
2. Implementing a branch and bound algorithm
3. Using techniques from software algorithms
4. Exploit the parallel nature of FPGAs

4.2.1 3-valued logic

As explained in chapter 1.5 minimum covering can be solved using a branch and bound algorithm. It's possible to represent the problem as binary search tree. Starting at the root descending to the leaves all branches that cannot lead to a solution should be pruned as early as possible. The crucial point is how to detect early, that a given branch cannot lead to a solution, because the CNF is not satisfiable for any subsequent variable assignments. Using ordinary boolean logic this decision is not possible. Traversing the search

tree from the root to the leaves, the result of the CNF cannot be evaluated without actually advancing to the leaf.

Particularly it not possible to distinguish 2 fundamental different cases: a) the CNF is not satisfied for the given *partial* variable assignment (where partial shall stand for we're not on a leave node yet, thus not all variables have been assigned a value yet) and b) the CNF is not satisfiable for *any* assignment of the currently non-assigned variables that is the current variable assignment has led to a contradiction.

Lets illustrate this with the following (trivial) example:

Suppose we have a CNF:

$$\Phi(\mathbf{x}) = (x_1 + x_2 + x_3)(x_1 + x_2)\dots$$

If we assign x_1 to 0 the CNF reduces to:

$$\Phi(\mathbf{x}|x_1 = 0) = (x_2 + x_3)(x_2)\dots$$

i.e. Φ is not a function of x_1 anymore. When assigning x_2 to 0 the CNF is further reduced to:

$$\Phi(\mathbf{x}|x_1 = 0 \wedge x_2 = 0) = (x_3)(0)\dots$$

but since the second clause is 0 the CNF cannot evaluate to 1 whatever values $x_3, \dots, x_n$ are assigned to.

For using a branch and bound algorithm a method for detecting such a contradiction as early as possible is needed.

A very elegant way is to use 3-valued boolean logic instead common binary boolean logic.

We define the logical values of 3-valued boolean logic as shown in table 1 and the corresponding boolean operations shown in table 2.

Symbol	Interpretation
H	logical 1, true
L	logical 0, false
X	not assigned, unknown

Table 1: Definition of 3-valued boolean logic

Using these definitions we can reconsider the above example. All variables are now initialized by assigning X. Thus

$$\Phi(\mathbf{x}) = (x_1 + x_2 + x_3)(x_1 + x_2) = (X + X + X)(X + X) = X$$

where the result X has to be interpreted as the result of the equation being not assigned/unknown yet. Assigning L to x_1, the CNF reduces to

a	b	$a+b$	$a \cdot b$	$\bar{a}$
0	0	0	0	1
0	1	1	0	1
0	X	X	0	1
1	0	1	0	0
1	1	1	1	0
1	X	1	X	0
X	0	X	0	X
X	1	1	X	X
X	X	X	X	X

Table 2: Operations in 3-valued boolean logic

$$\Phi(\mathbf{x}|x_1 = L) = (L + x_2 + x_3)(L + x_2) = (L + X + X)(L + X) = X$$

what means that the CNF can still be satisfied depending on further assignments. When now assigning L to x_2, we end with

$$\Phi(\mathbf{x}|x_1 = L \wedge x_2 = L) = (L + L + x_3)(L + L) = (L + L + X)(L + L) = L$$

where L means that the CNF is not satisfiable anymore, regardless of any other assignment to the currently not assigned variables.

As we can see the use of 3-valued logic can help us to find out whether a partial variable assignment can lead to a solution. As soon as the equation evaluates to H, we know that we have found a solution regardless of any subsequent variable assignments. L means that the equation is unsatisfiable for any assignment of the currently non-assigned variables.

4.2.2 Branch and bound algorithm

Now that we have introduced 3-valued logic a branch and bound algorithm for solving minimum covering (or SAT) can be designed more efficiently. We come back to algorithm 1 presented in chapter 1.5. The informal condition *crs_asgm cannot satisfy equation anymore* given in line 14 translates now easily to *the CNF evaluates to L under the variable assignment crt_asgm*. When initializing all variables to X at the beginning and setting the current variable x_i back to X when backtracking, we end up with algorithm 4 which is the basis for the architecture presented.

The recursive formulation of the branch and bound algorithm is an intuitive way to explain the algorithm and to create a compact program for the implementation on a CPU. For a hardware implementation recursive

Algorithm 4 Solving Minimum Covering with branch and bound (using 3-valued boolean logic)

```
1: branch_and_bound_recursive(crt_asgm,cur_var) {
   {global bst_cost saves cost of the best solution bst_asgm that has been
   found yet, all variables are initialized to X in the beginning}
2: crt_cost := calc_crt_cost()
3: lower_bnd := calc_lower_bnd()
4: if (CNF(crt_asgm) = H) then
5:    if (crt_cost < bst_cost) then
6:       bst_cost := crt_cost
7:       bst_asgm := crt_asgm
8:    end if
9:    cur_var = X
10:   return
11: end if
12: if (variables_to_assign = empty) then
13:   cur_var = X
14:   return
15: end if
16: if (CNF(crt_asgm) = L) then
17:   cur_var = X
18:   return
19: end if
20: if (CNF(crt_asgm) = X) and ((crt_cost + lower_bnd) ≥ bst_cst) then
21:   return
22: else {branch}
23:   x_i = select next variable that is unassigned yet
24:   remove x_i from set of unassigned vars
25:   x_i := 1
26:   add x_i to crt_asgm
27:   branch_and_bound(crt_asgm,x_i)
28:   remove x_i form crt_asm
29:   x_i := 0
30:   add x_i to crt_asgm
31:   branch_and_bound(crt_asgm,x_i)
32: end if
33: }
```

algorithms are not suitable but it is possible to reformulate the algorithm for an iterative solution.

Instead of thinking of traversing a tree-like structure when assigning the variables one after the other, we can think of each variable as being controlled by a dedicated finite statemachine which assigns a value to it. If we require that the order in which the variables are assigned is fixed, each statemachine needs only to communicate with its neighbors, i.e. statemachine i controlling variable x_i does only need to communicate with statemachines x_{i-1} and x_{i+1}. The recursive procedure calls in algorithm 4 are mapped to triggering the next statemachine, backtracking is mapped to triggering the previous statemachine. This procedure maps the tree traversal to a linear array of communicating statemachines, where each statemachine implements the function of the algorithm above.

4.2.3 Adopting techniques of software algorithms to hardware

While using the branch and bound algorithm given above will lead to a fully functional solution, we still can do better by applying similar simplifications as used when working with covering matrices introduced in chapter 1.5. The more reduction rules and bound conditions we use, the smaller the search space gets.

Dominated Column We have defined a dominant column as a column that has entries that are all larger or equal than that of the dominated column. If the original covering matrix already contains dominated columns, they are removed by a preprocessing step and no hardware is ever generated for them.

For detecting dominated column at runtime we have to deduce conditions for dominance, dependent on the results of the respective clauses $C_i == \{L, H, X\}$. Consider the following covering matrix:

	x_i	x_j
C_1	0	0
C_2	0	1
C_3	1	0
C_4	1	1

For example, if we assume $C_3 == H$ we can delete row 3 from the covering matrix and we can see, that x_i is dominated by x_j now. So we can deduce a dominated condition for x_i: x_i is dominated by x_j if $C_3 == H$.

We can illustrate the effects of clauses values on the dominance of columns using a Karnaugh-map.

C_3C_4 \ C_1C_2	XX	$X1$	11	$1X$
XX	$-$	j	j	$-$
$X1$	$-$	j	j	$-$
11	i	dc	dc	i
$1X$	i	$i \oplus j$	$i \oplus j$	i

Legend

i	x_i is dominated by x_j
j	x_j is dominated by x_i
$i \oplus j$	x_i is dominated by x_j and vice versa, exactly one of this variables can be set to L.
$-$	no domination
dc	x_j and x_i are don't care.

From this Karnaugh-map we can deduce the conditions for x_i being dominated by x_j (notation: $dom(x_i, x_j)$) and vice versa:

$$\begin{aligned} dom(x_i, x_j) &= (C_3 == H) \\ dom(x_j, x_i) &= (C_2 == H) \cdot (C_3 == X) \end{aligned}$$

This can be generalized in the following way: We have a look on any pair of 2 columns of the covering matrix i and j. We consider only pairs of columns, that have at least one (0 1) and one (1 0) row [1].

Thus suppose there are k columns with (0 1), denoted as $C_1^{01}, \ldots, C_k^{01}$, and l columns with (1 0) denoted as $C_1^{10}, \ldots, C_l^{10}$. We can formulate the condition of x_i being dominated by x_j and vice versa in the general case as follows:

$$\begin{aligned} dom(x_i, x_j) &= \prod_{\alpha=1}^{l} \left(C_\alpha^{10} == H \right) \\ dom(x_j, x_i) &= \left(\prod_{\beta=1}^{k} \left(C_\beta^{01} == H \right) \right) \left(\prod_{\alpha=1}^{l} \left(C_\alpha^{10} == X \right) \right) \end{aligned}$$

Using these definitions its possible to derive the dominated conditions for any variable by calculating the dominance condition for any pair of variables.

[1] If this is not the case there were dominated columns in the original covering matrix already. These kind of dominance can be removed directly in a preprocessing step before creating an accelerator circuit

The procedure is shown in the following example. Consider the covering matrix:

	x_1	x_2	x_3
C_1	1	0	1
C_2	1	1	0
C_3	0	1	1

The dominance conditions can be derived as:

$$\begin{aligned}
dom(x_1, x_2) &= (C_1 == H) \\
dom(x_2, x_1) &= (C_3 == H) \cdot (C_1 == X) \\
dom(x_1, x_3) &= (C_2 == H) \\
dom(x_3, x_1) &= (C_3 == H) \cdot (C_2 == X) \\
dom(x_2, x_3) &= (C_2 == H) \\
dom(x_3, x_2) &= (C_1 == H) \cdot (C_2 == X)
\end{aligned}$$

From these "mutual dominance" conditions the total dominance conditions can be derived:

$$\begin{aligned}
dom(x_1) &= dom(x_1, x_2) + dom(x_1, x_3) \\
dom(x_2) &= dom(x_2, x_1) + dom(x_2, x_3) \\
dom(x_3) &= dom(x_3, x_1) + dom(x_3, x_2)
\end{aligned}$$

Essentials / Implications Essential columns were introduced as columns in the covering matrix, that have the only 1 entry of some row. This means that the corresponding group must be selected, otherwise the set cannot be covered.

Since the architecture doesn't make explicit use of covering matrices and therefore no matrix simplifications are done, essential columns can only come from the initial problem and can be detected in the preprocessing step before the accelerator is actually generated.

But we can even go on step further. Essential columns can be interpreted as implications in minimum cost SAT.

Implication can be thought of deriving conditions for non-assigned variables from already assigned variables. Consider the following example:

$$\Phi(\mathbf{x}) = (x_1 + x_3 + x_4)(x_1 + x_2)$$

Suppose x_1 is set to L. This implies the condition $x_2 = H$ for x_2, since otherwise the second clause is not satisfiable anymore. Thus x_2 must be set to H as long as the essential (implication) condition ($x_1 == L$) is true.

When looking at unate covering problems, deriving essential (implication) conditions for all variables is straight-forward. Lets extend the previous example:

$$\Phi(\mathbf{x}) = (x_1 + x_2 + x_3)(x_2 + x_3)(x_1 + x_3)(x_1 + x_2 + x_4)$$

The essential conditions for $x_1 \dots x_4$ are:

$$\begin{aligned} ESS(x_1) &= \bar{x}_2 \cdot \bar{x}_3 + \bar{x}_3 + \bar{x}_2 \cdot \bar{x}_4 \\ ESS(x_2) &= \bar{x}_1 \cdot \bar{x}_3 + \bar{x}_3 + \bar{x}_1 \cdot \bar{x}_4 \\ ESS(x_3) &= \bar{x}_1 \cdot \bar{x}_2 + \bar{x}_2 + \bar{x}_1 \\ ESS(x_4) &= \bar{x}_1 \cdot \bar{x}_2 \end{aligned}$$

Generalizing this: an essential condition for variable x_i can be derived by removing all occurrences of x_i in the CNF (thus setting x_i to L) inverting all remaining literals and replacing all $\cdot$ with $+$ and vice versa. In unate problems only the implication *set-to-H* can occur. This cannot create contradictions, since setting all variables to H *always* satisfies the CNF equation.

When looking at binate problems, the procedure is similar but there are two implication conditions: one for forcing the variable to L (set-to-L) the other condition for forcing the variable to H. This can lead to contradictions if both conditions are satisfied.

Don't Cares Don't cares can be considered as the opposite of essentials. Whereas essentials force the variables to a specific value, don't cares conditions imply that the value of a variable has no influence on the result of the CNF. This means that the variable can be set to X. Consider the following case:

$$\Phi(\mathbf{x}) = (x_1 + x_3 + x_4)(x_1 + x_2)$$

When x_1 is set to H the value of x_2 doesn't matter, since the first clause is not a function of x_2 and the second clause is already satisfied. We call x_2 a don't care variable. A more complex example shall illustrate the general procedure:

$$\Phi(\mathbf{x}) = (\bar{x}_1 + x_2 + x_3)(x_2 + \bar{x}_3)(x_1 + \bar{x}_3)(x_1 + x_2 + x_4)$$

$$
\begin{aligned}
DC(x_1) &= (x_2 + x_3) \cdot \bar{x}_3 \cdot (x_2 + x_4) \\
DC(x_2) &= (\bar{x}_1 + x_3) \cdot \bar{x}_3 \cdot (x_1 + x_4) \\
DC(x_3) &= (\bar{x}_1 + x_2) \cdot x_2 \cdot (x_1) \\
DC(x_4) &= (x_1 + x_2)
\end{aligned}
$$

The don't care condition for variable x_i can be derived by removing all clauses that do not contain x_i from the original CNF and removing the literal x_i from any remaining clauses. Don't cares are derived in the same way for both unate and binate problems.

4.2.4 Using the parallel nature of FPGAs

When solving SAT or Covering problems using branch and bound based algorithms a major trade-off has to be considered: the search space can be greatly reduced by applying sophisticated bound conditions but the evaluation of complex condition can be quite costly in terms of computing time which could nullify the benefit. This holds true for software implementations since all computation is done sequentially. When using an FPGA implementation things look different as all computing is done in parallel. This means that adding more bounding conditions of the same complexity as the existing ones, results in larger circuits but has no large effect on the speed of the circuit because the critical path is not lengthened. Anyhow, adding additional bound conditions that are more complex than the existing ones may increase the length of the critical path and therefore limit the maximum clock rate of the circuit.

As we have seen in the previous paragraphs the bound conditions presented there, e.g. don't-care and essential, are about the same complexity and require evaluations of two-level boolean equations. Thus adding such a additional bound-condition does not slow down the resulting circuit significantly.

4.3 Accelerator Architecture

The accelerator is divided into 4 blocks: *statemachines*, the *checkers*, the *costcounter* and the *controller* [2]. A schematic view of the accelerator is given in figure 5.

The system is controlled by the controller block, which is responsible for initializing the circuit, retaining information about the best solution found and finally detecting the end of the computation.

[2]The actual *implementation* of the accelerator requires some additional blocks for handling configuration and host communication issues. These blocks are described in chapter 5.

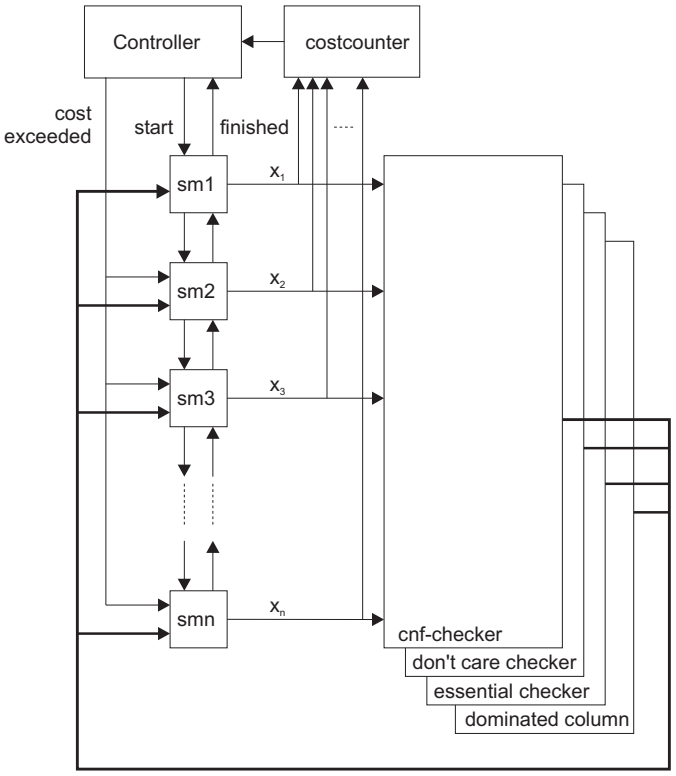

Figure 5: Schematic representation of System

Each variable of the covering problem is associated to a "controller" that controls its value. To avoid confusion between the controller of the whole accelerator and the controllers of the particular variables, the latter are referred to as *statemachines*.

The output of a statemachine which is the current assignment of the variable associated with this statemachine is fed into the checkers, which evaluate the CNF equation and the conditions the branch and bound algorithms relies on. Each statemachine implements the branch and bound algorithm related to its associated variable. The feedback from the checkers i.e. whether the current assignment satisfies the CNF and the bounding conditions, are used for the computation of the next state of the statemachine. All statemachines are arranged in a linear array, where each statemachine is connected only to its immediate neighbors and the checkers. A statemachine acts upon activation from the adjacent statemachines.

The solving procedure The procedure can be thought of as follows: Starting with an initial variable assignment "not-assigned", the first statemachine is triggered by the controller, all subsequent statemachines are trig-

gered by the previous or the next statemachine.

When the statemachine is triggered, it executes an algorithm similar to algorithm 4 using 3-valued boolean logic. The result of the CNF function, as well as the truth values of the bound conditions are computed. After that, the statemachine performs either a bound, which means stopping processing and returning control to the previous statemachine by triggering it, or a branch by assigning a value to the variable controlled by this statemachine, and triggering the next statemachine. A statemachine does *not* have to know where in the array it is located, all statemachines are completely the same. The whole procedure is started by triggering the first statemachine which is done by the controller. Now control is passed between the statemachines i.e. there is always only one single active statemachine. The last statemachine is connected to itself and represents a leaf of the search tree. The end of the algorithm is reached if the first statemachine tries to trigger its predecessor. This signals to the controller, that the algorithm is done.

One design goal was to make it easy to implement different variants of the branch and bound algorithm. This goal is reached by encapsulating the actual solving algorithm – i.e. which bounds are used – in the statemachine. Different variants can be used by simply replacing the statemachine blocks with different ones. Since each algorithm may require different bound conditions to be evaluated, the part of the checker that computes these conditions has to be changed as well. The rest of the system can be left completely unchanged.

4.3.1 Statemachines

Different variants of the statemachine have been implemented for evaluating the influence of various bounding conditions on the performance of the architecture. The statemachines are described as Moore statemachines.

The first statemachine that shall be presented here uses a statemachine that basically implements the procedure given in algorithm 1 and is given in figure 6. The figure gives a representation of the statemachine as state transition diagram as well as a view of the input and output signals.

When the cost of the current assignment exceeds the cost of the best solution found so far a bound is performed. Furthermore, the branching stops if the partial assignment leads to a contradiction, i.e. the CNF is already unsatisfiable.

As this statemachine is the basis for all other statemachines it shall be discussed a little more in detail. The three values quoted in the state transition diagram represent the value of the outputs of the statemachine in the form (VAR,TB,TT) where signal VAR is the value of the variable associated to this statemachine, TB is the 'to bottom' signal used to trigger the next statemachine if a branch is performed and TT is the 'to top' signal to trigger previous statemachine if a bound is performed (backtracking).

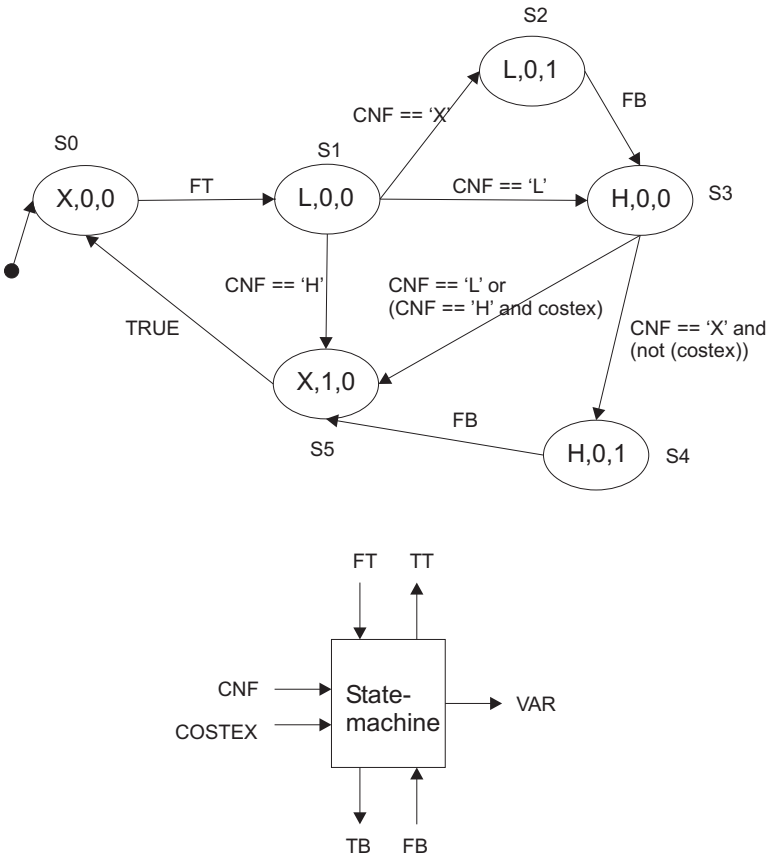

Figure 6: Statemachine basic version

At initialization, each statemachine enters state *S0* which assigns X to the output which corresponds to the variable controlled by this statemachine, henceforth called *variable* for brevity. The statemachine waits until it is triggered by the previous statemachine connected to it and assigns L to the variable after that (*S1*).

If this partial variable assignment satisfies the CNF we have found a new solution and do not need further branching. So we can backtrack by triggering the previous statemachine (*S5*). If the CNF evaluates to X we trigger the next statemachine by asserting TB and wait until the next statemachine has done its work (*S2*). Since setting a variable to L adds no cost, we were allowed to do this branch without considering a cost bound condition.

Now the other branch can be performed by setting the variable to H and branching again. But before branching the cost bound has to be checked, since setting a variable to H adds costs of 1 (*S3*). Branching is only done if the CNF is not satisfied yet but still feasible (CNF evaluates to X) and the cost of the current assignment does not exceed the cost of the best solution

found so far. If a contradiction was found or the cost bound was reached, the variable is immediately relaxed by assigning X (*S5*) and the previous statemachine is triggered by asserting TT, finally the statemachine returns to the initial state (*S0*).

This basic statemachine can also cope with don't cares, essential columns and dominated columns when adding certain extensions.

If a variable is don't care when the statemachine is triggered, control is passed directly to the next statemachine and the value of the variable kept at X. Since the don't care condition does necessarily hold true for any subsequent variable assignment, the value can be held at X until the statemachine is re-triggered from bottom by FB (*S6*). In this situation the statemachine can trigger the previous statemachine immediately (*S5*).

The changes to the support of dominated columns is quite straight forward. As soon as it is detected that a column is dominated, the checker asserts the DO signal, which causes the statemachine going directly in state *S7* where the variable is set to 0. Upon triggering from the previous statemachine, the triggering signal is passed directly to the next statemachine (*S8*). For essential variables the extension to the statemachine is almost identical besides the variable is set to H instead to L (*S10*).

Since the extensions to the statemachine are orthogonal, a reduced statemachine that doesn't make use of essential column detection or dominated columns can be derived easily be omitting the corresponding parts of the statemachine.

4.3.2 Checker

The variable outputs of all statemachines are fed into the checker and the checker checks the conditions for don't care, essential column or dominated column for every variable. The according DO, DC, ES signals are fed back to the statemachines.

The logic for detecting these conditions is instance specific and generated by infering these conditions from the CNF of the problem given.

4.3.3 Cost counter

Parallel counter Whenever a new variable assignment is made, the cost of this assignment has to be recalculated in order to check whether the cost bound has been reached or not.

The cost counter has to compute the number of variables that are set to H, i.e. a special kind of adder is needed. As opposed to the commonly used n-bit adders that add 2 numbers where each number has n bits, an adder that adds n 1-bit numbers is needed for this application. A circuit which accomplishes this task is called *parallel counter*. How efficient parallel counters can be designed was proposed in [SJ73].

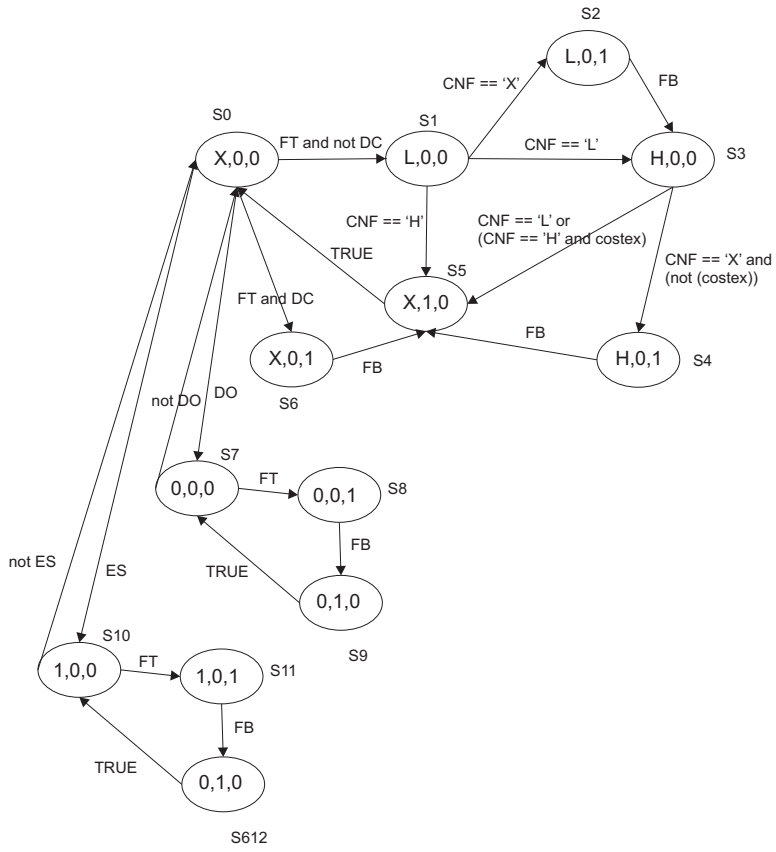

Figure 7: Statemachine final version

The basic idea is to regard the 1-bit full-adder as 3-bit parallel counter, since 3 bits (2 bits and a carry_in bit) are added in parallel and 2 output bits are computed, the sum and the carry_out bit. A larger parallel counter can be synthesized from two smaller counters and an adder. The results of the smaller size adders are fed into another adder where they are added again until all bits have been summed up. An example of a 15-input parallel counter is shown in figure 8.

A parallel counter constructed in this tree-like fashion still adds a significant delay for large widths of the input words. Each adder has a delay of τ_{adder}. Since ripple carry adders are used the delay resulting from the adders of stage n of the parallel counter is $n \cdot \tau_{\text{adder}}$ because the signals have to pass n full-adders at stage n. The overall delay of the complete parallel counter is the sum of the delays of all stages, i.e. the delay of an n stage adder is:

$$\text{delay(n)} = \Big(1 + 2 + \ldots + n\Big) \cdot \tau_{\text{adder}} = \frac{n(n-1)}{2} \cdot \tau_{\text{adder}}$$

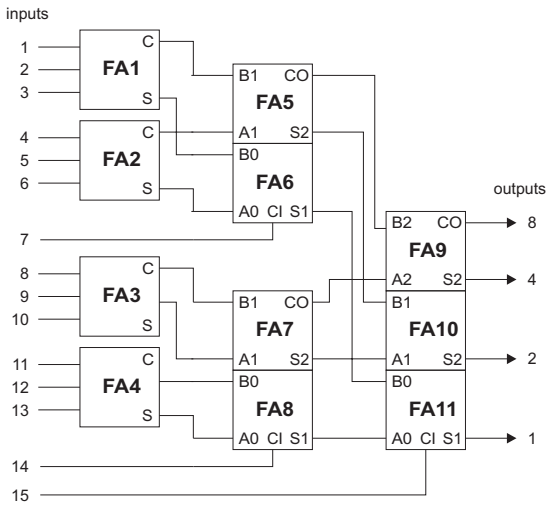

Figure 8: 15-bit parallel counter

As a counter with N input bits needs $\lfloor \log_2(N) \rfloor$ levels the delay of an N bit parallel counter is given as:

$$\tau_{\text{n-bit-ctr}} = \frac{l(l-1)}{2} \cdot \tau_{\text{adder}} \quad \text{where} \quad l = \lfloor \log_2(N) \rfloor.$$

For speeding up large parallel counters, faster adders such as carry-look-ahead or carry-select adders may be used.

Alternative counters As we can see, the parallel adder is one of the critical components in the architecture since it adds a significant amount of delay and is in the critical path. When reconsidering the original problem, we can come up with a different solution. In this architecture, new variable assignments are not chosen randomly at any point of time, but starting with all variables assigned to X only one variable after the other is assigned a value. We can make use of this property and implement an up/down counter, which is initialized with 0 at the beginning. Each time a variable is assigned the value H, the counter is incremented. Each time a variable is assigned to 0 or X, the counter is decremented. The statemachines have to be slightly extended in a way that the corresponding 'count up' and 'count down' signals are generated accordingly.

While this scheme will work well for the basic architecture even with the use of don't cares and dominated columns, the use of essential column checking makes things more difficult. The problem with essentials is that several statemachines can set their variables to H at once, as a statemachine reacts immediately on the dominated signal, without being triggered.

To determine how much variables have become essential, a counter will be necessary again.

4.3.4 Controller

The controller is used for:

- starting the computation and detecting when the end of the algorithm is reached
- keeping track of the best solution found so far and its cost
- checking the bound condition for the cost of the current assignment

The controller is connected to the first statemachine via start and finished. start is connected to the FT input of the first statemachine and finished is connected to TT. When the controller asserts start the algorithm begins execution. The of the algorithm is reached when the first statemachine asserts finished.

5 Implementation of Accelerator

5.1 System Perspective

The accelerator has been implemented on the Sundance SMT320 platform which was already used in the past for other projects. The Sundance platform consists of a full size PCI carrier card, which can be equipped with up to three different modules, depending on the application. A fourth module, which is a Texas Instruments DSP module, has alway to be present on the card, since it handles the communication between the host PC and the card and is used for configuring the other modules. Windows NT drivers allow to access the board from custom applications.

In our case the SMT320 was configured with 1 DSP module (SMT358) and a module called SMT358 which provides a Xilinx Virtex XCV1000BG560-4 FPGA, which is currently among the FPGAs with the highest capacity and fastest speed. Additionally, the module provides peripheral devices for establishing communication channels to the DSP and for configuration of the FPGA. 4 independent banks of fast RAM are provided for systems that need additional memory besides the internal Virtex Block-RAM.

All communication between the modules on the SMT320 is done via bidirectional *commport* interfaces, which are compatible with a communication interface that was introduced with Texas Instruments TMS320C40 DSP series. The SMT320 provides some kind of *virtual commport channel* between the host computer and the DSP module, i.e. commport 3 of the DSP module is connected via the PCI bus to the host. The host can communicate with the DSP using functions of the SMT320 driver, but for the DSP the communication looks like communication on any ordinary commport. Although the DSP devices that had integrated commports are obsolete today, the communication interface has become a standard interface for communication in multiprocessor systems consisting of TI processors. The SMT320 board provides connectors for flat-ribbon cables that allow flexible connections of the commports of the installed modules.

For the implementation of the minimum covering accelerator the connections were configured as shown in figure 9.

For keeping the design of the FPGA circuit and DSP software simple, 3 commports are used in this design, 1 dedicated FPGA re-configuration commport, and 2 commports for the communication between DSP and FPGA where each commport is used only unidirectionally for simplicity.

Looking from a system perspective at the accelerator, figure 10 shows how we proceed from a problem description to a solution.

5.2 VHDL generation for implementation and simulation

When using hardware description and simulation languages for designing architectures that shall be simulated and synthesized it's good practice to

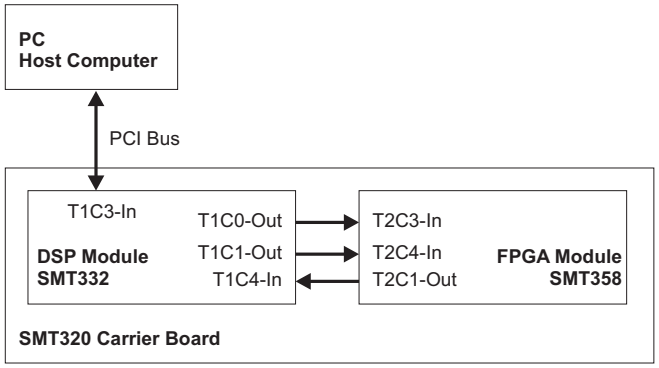

Figure 9: Configuration of Communication Channels on SMT320

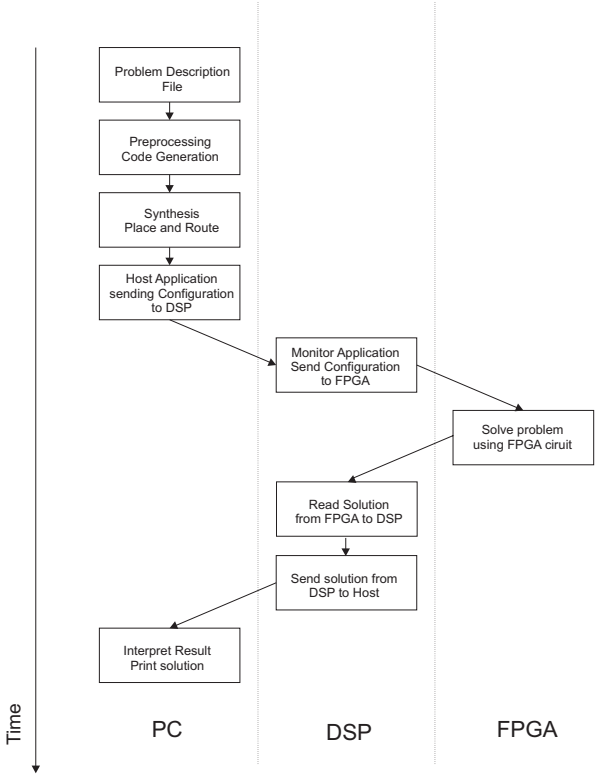

Figure 10: Tool Flow for Accelerator

use most of the code for both, simulation and synthesis. This allows a simpler verification of the circuit, since any change in the architecture can be simulated immediately.

Therefore the design decision has been made of encapsulating the common parts of the architecture for simulation and synthesis in a single VHDL component called *top*.

For a real implementation of the solver on the FPGA this top component and some additional FPGA specific design units can be combined to get an implementation, whereas the same top component can be combined with a test-bench architecture for VHDL simulation. Since the actual solving algorithm is included completely in the top component, an architecture that has been proven to work in simulation, is supposed to work also in an FPGA implementation.

The design goal was to make the design as parameterizable as possible by using VHDL's own means of describing parameterizable circuits: the use of constants declared in packages and instantiation of VHDL components using VHDL's generate statement. As these means are quite powerful, most of the instance specific peculiarities can be handled without dynamically generating VHDL code.

Many parameters of the design, i.e. the interfaces between the particular VHDL entities or the size of registers, depend only on the size of the covering problem, i.e. the number of clauses and variables. These kind of dependency on the problem instance can be covered easily by the use of constants in the corresponding declarations. Constants defining the problem are extracted from the problem description and placed in a file called *config.vhd* which is included in all other design units. As many design units depend only on these constants, most VHDL code does not have to be modified at all for targeting a different problem instance.

Some specifics of the problem instance are not coverable by constants and generate statements and really need dynamically generated VHDL code. Therefore the *checker* component has been introduced, which consists of all code, that – besides config.vhd – has to be generated dynamically. This is done by the program *genvhdl.pl* discussed in section 5.6.

5.3 VHDL Code Overview

To give an outline of the architecture from the implementation point of view, I will give a short overview of the files, their relations, the components that are implemented by them and whether they are used for implementation or simulation or both. A graphical representation of the top level blocks is given in figure 11.

Top.vhd encapsulates the architectural part of the accelerator and serves as some kind of top-level file for simulation *and* implementation. In top.vhd

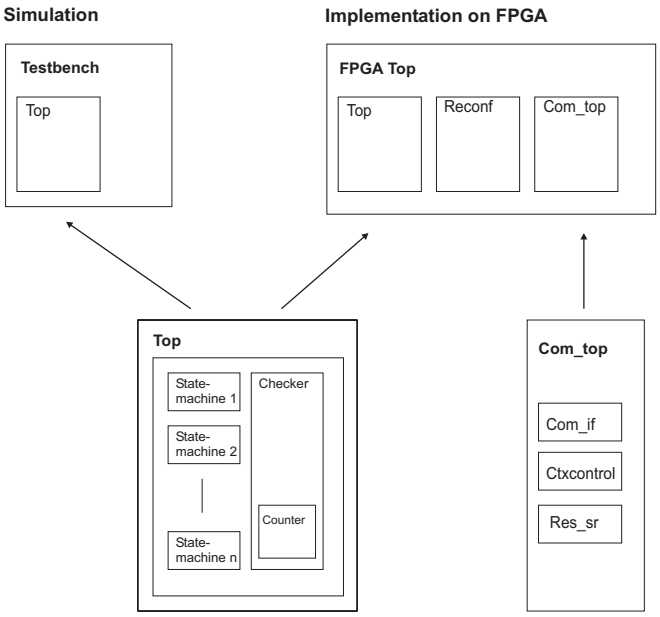

Figure 11: Outline of Top-level VHDL components

the checker component and the statemachines are instantiated.

Testbench.vhd is the top-level VHDL entity for simulation of the system and is the implementation of the *controller* for the accelerator simulation. Its purpose is to instantiate a top component, generate the clock signal and to start the computation. When the end of the algorithm is detected, the result is written to a file. Using the script *simulate* it's possible to automate the simulation using the Modelsim VHDL-simulator.

Fpgatop.vhd is the top-level VHDL entity for implementation on our Sundance prototyping platform. In addition to the top component two other components are instantiated: reconf and com_top, which are used implement the communication interfaces on the FPGA for configuring the FPGA and for read-back of the results. Additionally, fpgatop implements the controler for the synthesizable accelerator.

Config.vhd is a package that defines important constants that are specific for the problem instance given. This package is included in every other VHDL file. Thanks to these constants, not every VHDL file has to be generated depending on the problem instance, since most of the peculiarities of the problem instance given can be expressed simply as constants and therefore the VHDL code often can make use of these constants.

Statemachine.vhd is an implementation of the statemachine described in section 4.3.1. The current implementation does provide support for don't cares, essentials and dominated columns. The statemachines are connected to the checker which is implemented in checker.vhd that computes the values for the various conditions (don't care, essential, dominated) for any variable.

Checker.vhd Checker.vhd is the only instance specific VHDL file (besides config.vhd which consists only of constant definitions) and can be considered the heart of the architecture in conjunction with the statemachines. The checker component is fed by the variable states given by the statemachines and computes the truth value of the various conditions the statemachine needs (don't cares, essential columns and dominated columns). Besides that, the cost of the current assignment is computed by using a parallel counter shown in detail in section 4.3.3.

Adder.vhd is a generic design of a n-bit adder with carry-in and carry-out, used for building up the parallel counter. The Xilinx implementation tools synthesize the adders using the Virtex fast carry-chains.

Reconf.vhd is a general circuit that has to be implemented in the FPGA for the support of reconfiguration. Its purpose is to instantiate a commport communication interface and listening for a reconfiguration keyword. As soon as this keyword is received, the FPGA is set to reconfiguration mode by activating the reconfiguration mechanism controlled by a CPLD attached to the FPGA. For further details see section 5.4.3.

Com_top.vhd & res_sr.vhd are used for instantiating the commports used for the communication between DSP and FPGA. The protocol is implemented in com_if.vhd and ctxcotrol.vhd. As explained above a simple communication mechanism with 2 unidirectional commports is used. Commport *comr* is used for receiving a command from the DSP and placing it into a receive register. Commport *coms* waits until a word has been received from *comr* and sends the desired answer. The functionality includes echoing data for debugging purposes, sending the value of the counter that keeps track of the cycles passed since the start of the algorithm, sending the current status (algorithm done or still running) to the DSP. The most important function is to send the result of the algorithm back to the DSP when the end of computation is reached. This is done by sending the result bit by bit using a shift register implemented in res_sr.vhd.

Com_if.vhd & ctxcontrol.vhd are IP cores provided by Sundance that implement a standard TI C40 style commport interface. While com_if.vhd

already does implement a complete bidirectional commport, ctxcontrol is used to simplify the sending of data via the commports.

5.4 Hardware Blocks

The implementation of the hardware blocks that have been described in chapter 4 should be comprehensible since a clear coding style that shows the concepts was preferred over a design that might be somewhat faster and smaller by using highly Virtex FPGA optimized code. Many design ideas have already been explained in the design chapter, so in the following I want to stress some implementation related concepts.

5.4.1 3-valued logic

The use of 3-valued logic is one of the key concepts of this architecture as already explained in the design section. Since FPGAs evidently cannot operate on other than binary boolean logic, 3-valued logic has to be coded by using 2 bits per variable. Thus H,L and X are associated to a 2-bit binary vector. Since coding of these 3 values has an influence on the size of the logic needed for implementation of the particular 3-valued operations a good coding has to be chosen.

For the first implementation of the accelerator a new ternary VHDL data-type has been created, which allowed to directly declare 3-valued signals and to operate directly on them using overloaded boolean operators. For implementation on the FPGA, the design tools had to choose an appropriate coding themselves. Unfortunately, the design tools appear to be unable to find a optimal coding resulting in bloated logic. So, despite of the elegance of this data-types in ternary logic and overloaded operators, an explicit coding has been preferred.

Implementing a similar architecture for SAT accelerators [PM98], Platzner has spotted that coding $L = 00$, $H = 10$ and $X = 11$ (assignment to 00 is illegal) leads to an efficient implementation, which requires exactly twice the number of gates needed for binary boolean operations.

Applying these definitions to the truth tables for 3-valued logic cf. table 2 leads to the an explicit representation of the logic operations given in table 3.

If we have a closer look at the truth table, see table 3, we see that

$$\begin{aligned}(a \cdot b)(1) &= a(1) \cdot b(1)\\ (a \cdot b)(0) &= a(0) + b(0)\end{aligned}$$

what means that the 3-valued AND operation is the binary AND of the bits 1 and the binary OR of the bits 0. Further

a		b		$a \cdot b$		$a + b$		$\bar{a}$	
0	01	0	01	0	01	0	10	1	10
0	01	1	10	1	01	0	10	1	10
0	01	X	11	X	01	0	10	1	10
1	10	0	01	1	01	0	01	0	01
1	10	1	10	1	10	1	01	0	01
1	10	X	11	1	11	X	01	0	01
X	11	0	01	X	01	0	11	X	11
X	11	1	10	1	11	X	11	X	11
X	11	X	11	X	11	X	11	X	11

Table 3: Operations in 3-valued boolean algebra

$$\begin{aligned}(a+b)(1) &= a(1) + b(1)\\ (a+b)(0) &= a(0) \cdot b(0)\end{aligned}$$

i.e. the 3-valued OR operation is the binary OR of the bits 1 and the binary AND of the bits 0. and finally

$$\begin{aligned}\bar{a}(1) &= a(0)\\ \bar{a}(0) &= a(1)\end{aligned}$$

hence 3-valued NOT is simply a swapping of the bits.

5.4.2 Clausechecker

For evaluating the CNF a *clause checker* is implemented. As the CNF is built up as product-of-sums it consists of a huge AND-gate, that is fed by the results of the respective clauses. Using explicitly coded 3-valued logic as described above, a binary CNF results in two equations. Consider the following equation:

$$\Phi(\mathbf{x}) = (x_1 + x_2 + x_3) \cdot (x_2 + x_3) \cdot (x_1 + x_2 + x_4)$$

applying the rules for 3-valued logic above this equation is expressed as:

$$\begin{aligned}\Phi(\mathbf{x})(1) &= (x_1(1) + x_2(1) + x_3(1)) \cdot (x_2(1) + x_3(1)) \cdot (x_1(1) + x_2(1) + x_4(1))\\ \Phi(\mathbf{x})(0) &= (x_1(0) \cdot x_2(0) \cdot x_3(0)) + (x_2(0) \cdot x_3(0)) + (x_1(0) \cdot x_2(0) \cdot x_4(0))\end{aligned}$$

The implementation of the clause checker and the other checkers needs boolean functions of many variables. Since these functions are used frequently and lie on the critical timing path, an efficient implementation has

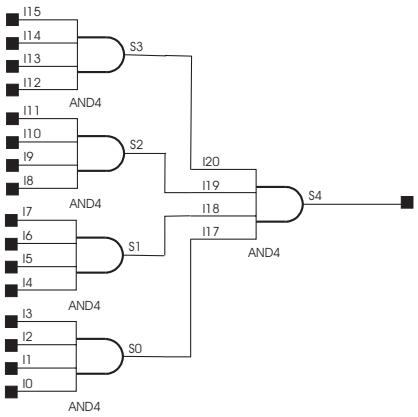

Figure 12: AND16 Tree-Implementation on Virtex

to be found. Since FPGAs are built up from 4-input LUTs any 4-input boolean function can be implemented using 1 LUT. A first idea to implement fast and wide logical boolean functions was to generate tree-like structures. For example a 16-bit AND function could be implemented using 4 4-input LUTs each AND-ing 4 bits and connecting the outputs to another LUT which combines the partial results again, see figure 12. As the signals need to pass $\log_4(n)$ LUT's, the delay of the n-input AND is $\lceil \log_4(n) \rceil \cdot \tau_{LUT}$ where τ_{LUT} is the delay of the signal passing a single LUT.

When considering the implementation of even wider boolean functions the question arises whether it is possible to speed up this computation on the one hand and how to make most efficient use of the resources on the FPGA on the other hand.

The first idea to solve these two issues quite elegant was the use of wired-and/or logic. The resources on the FPGA (long-lines that can be driven by three-state drivers and pull-up/down resistors that can be activated) could be used potentially to implement wired-and/or gates. Unfortunately it was rather complicated to get the Xilinx Tools to generate the circuit desired. A Usenet posting on how to trick the tools to generate the structure needed, revealed a completely different, very elegant way to implement boolean functions of many input variables.

Consider the 16-bit AND example again: The problem with implementing AND16 using the tree structure from above is that we create additional dependencies that limit the performance, because the result of a AND4 is fed into another AND4, i.e. although all information that is needed to evaluate the equation is known from the beginning. This means that additional delays are introduced by combining the signals for evaluation, because each stage has to wait for the results of the previous stage. The idea to circumvent these additional delays resulting from several logic levels is to use a

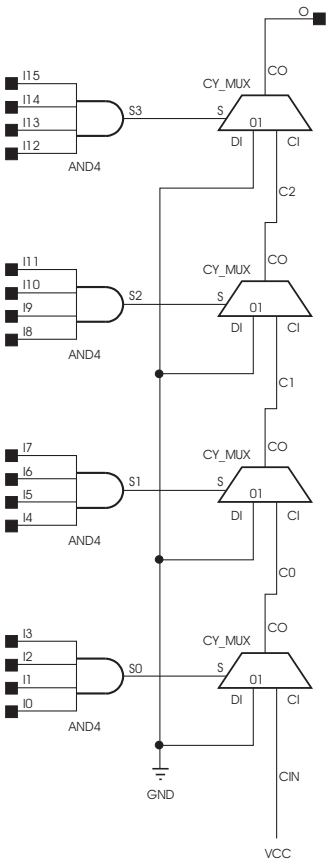

Figure 13: Efficient AND16 implementation for Virtex FPGA

combination of multiplexers and LUT's. The calculation of AND16 is split up into a chain of AND4 function-blocks. Each block has the following inputs: 4 bits that are the input to the function and the partial result of the computation of the adjacent block. The LUT is configured to implement AND4 and its output is used to control a multiplexer. If AND4 of the input bits equals to 1, the multiplexer is set to forward the result of the previous stage to the next stage. If the result of the current stage is false, 0 is forwarded to the next stage. The input of the first stage is fed by constant 1, the multiplexer output of the last stage is the result of AND16, see figure 13.

The big advantage of this implementation is, that the delay of an n-bit AND is reduced to only $1 \cdot \tau_{\mathrm{LUT}} + (\lceil \frac{n}{4} \rceil - 1) \cdot \tau_{\mathrm{routing}}$ where τ_{routing} is the delay introduced by routing the signals from one logic cell to the next. Since this structure maps perfectly to the fast Virtex carry-chain routing structure, very fast implementations can be achieved.

Implementations of test functions using this kind of implementation for

wide boolean functions showed significant advantages in speed and area over the tree style implementation. Due to time restrictions for this diploma thesis this scheme could not be implemented but is scheduled for future work.

5.4.3 Reconfigurator

Sundance has chosen a way to configure the FPGA on the SMT358 module by receiving the configuration via dedicated commport 3. The processing of the commport protocol and the configuration procedure is done in a CPLD attached to the FPGA. While this scheme allows simple configuration of the FPGA by simply sending the configuration bitstream to commport 3 of the SMT358 module, the user cannot completely control the reconfiguration procedure.

Before going into details it has to be clarified, that the procedure of reconfiguration that is described in the SMT358 user manual and in the following section is *not* the same as partial reconfiguration as described in the Xilinx Virtex manuals. Virtex partial reconfiguration allows to replace parts of the circuit of a configuration with alternative parts, where the rest of the circuit is unaffected and even continues to run during reconfiguration. The only way to reconfigure the SMT358 is to reset the FPGA and therefore resetting it to the same state as after power-up and to do a complete, new configuration cycle.

The documentation of the re-configuration process in the SMT358 user manual is not very precise and can be misunderstood, since it is assuming a rather specific implementation of reconfiguration control rather than explaining the general mechanism.

The confusion results from the fact that Sundance does not want to completely sacrifice a dedicated commport for configuration, but allow the reuse of the configuration commport as general purpose commport, once the configuration of the FPGA is done. This is achieved by the CPLD going into bypass-mode after the configuration.

Signal FPGARESET on the SMT358 module is a bidirectional signal, that can be driven to 0 by the CPLD and should be used as reset signal for the FPGA circuit. The other function of FPGARESET, and this is what makes things puzzling at the first sight, is to initiate a reconfiguration of the FPGA, which can (or better: *must*) be controlled by the FPGA itself. More precisely this means: If the FPGA wants to initiate a reconfiguration, the CPLD has to be set to configuration mode. This is done by driving FPGARESET to 0. After that, the FPGA can be reconfigured via commport 3. The mechanism how the FPGA detects that a reconfiguration is desired is left to the user (as opposed to the description in the SMT358 user manual). *One* way that is used in this architecture too, is to instantiate a commport interface on comport 3 which waits for the reception of a reconfiguration

keyword and drives FPGARESET to 0 after detection.

The VHDL code needed for this kind of reconfiguration is very simple, it consists primarily of the instantiation of a commport interface for commport 3, which is configured as receiver at reset. The comport is hard-wired to always receive and never send.

All the reconfiguration magic is done in the the third line given below, where FPGARESET is set to 0 as soon as a word on the reconfiguration commport is received and bit 0 of the word is set to 1, which corresponds to the magic reconfiguration keyword of 0x0001.

```
CTX_RDY <= '0';                -- never ready to send
CRX_RDY <= '1';                -- always ready to receive
FPGARESET <= '0' when ((REG_RX(0) = '1')
                  and (WRITE_REG = '1')) else 'Z';
```

Note that FPGARESET is *not* driven when no reconfiguration is required. Since there are apparently no external pull-up resistors attached to FPGARESET and the commport interface, internal pull-ups are attached to FPGARESET and 2 commport handshake signals by adding

```
NET     FPGARESET  PULLUP;
NET     C3P<9>     PULLUP;
NET     C3P<10>    PULLUP;
```

to the constraint file (UCF-file) used for circuit synthesis. The penalty of using this simple implementation is that commport 3 is sacrificed completely for configuration and cannot be used as general purpose commport anymore.

5.5 DSP Software

As mentioned before, the accelerator circuit doesn't use sophisticated strategies for re-use of commports, but uses the commports only in unidirectional way on the one hand and a dedicated port for configuration and re-configuration control on the other hand.

The current architecture uses the DSP mainly for managing the communication host-to-FPGA, configuration of the FPGA and reading status and results of the solver circuit on the FPGA.

The DSP software is designed as *monitor application* which is mainly a forwarder of commands from the host to the FPGA and vice versa. The functionality is described by the following pseudo code:

```
while(1){

   cmd = readCommandFromHost();
```

```
   switch(cmd){

   case NEW_CONFIGURATION:
      read configuration bitstream from host;
      send reconfiguration command to FPGA via configuration commport;
      send new configuration to FPGA via configuration commport;
      break;

   case SEND_COMMAND:
      forward command from host to FPGA via send port;
      break;

   case READ_RESULT:
      read result(s) from FPGA from receive port;
      send results to host;
      break;
   }
}
```

5.6 Host Software: Circuit Generation

Generating an instance specific solver for both simulation and synthesis is done by the program *genvhdl.pl* implemented in Perl.

As explained above in section 5.2, only 2 files that depend directly on the problem instance have to be generated: config.vhd and checker.vhd. All other files can be copied directly from a template directory.

genvhdl.pl is a command line application that is called as follows:

```
genvhdl.pl templatedir problem.cnf arch dir
```

where *templatedir* is the path to the directory containing all VHDL code-templates, *problem.cnf* is the problem description file in standard CNF-file notation, *arch* is the name of the accelerator architecture that shall be created. The files that are generated are put into directory *dir*.

The problem description file *problem.cnf* is read and the problem characteristics that can be expressed by constants are computed and a corresponding config.vhd file is generated. The generation of checker.vhd bases on a checker.vhd template. Depending on the features the architecture supports (CE,DC,...), corresponding code is generated by genvhdl.pl. The code for the features needed is created and inserted into the appropriate locations in the checker.vhd template.

5.7 Host Software: Host application

Downloading the accelerator for a new problem and reading back status and the results of the solver is done using an application on the host, called *hostapp*. Hostapp has been implemented using Visual C++ on Windows NT and uses the drivers for the Sundance SMT320 board.

The application allows to reset the FPGA board and upload a new configuration to the FPGA. After the circuit is uploaded, the solving procedure starts immediately and the host-application can be used to query whether a solution was found and how many clock cycles the algorithm was running to find the solution. Once the solution is found, the results can be read back using the host-application.

5.8 Using the Accelerator

The following list, shall give an overview on how the accelerator is actually used. It is shown how a given problem description file is transformed into an FPGA configuration and how the results of the circuit are read back:

- **Generating instance specific VHDL code** Using the application *genvhdl.pl* instance specific VHDL code for the accelerator is created. Additionally the testbench and the script for automated VHDL simulation are created.

- **Simulating the design** Simulation of the design can be done by using the *compile* and *simulate* scripts. Using the waveform configuration *wave.do* the most important signals are displayed in the simulator. Note that the communication hardware related files can not be used in simulation, instead testbench.vhd is used as top-level VHDL entity. The solution is written to a file after termination of the algorithm.

- **Synthesis** A new Xilinx Foundation HDL project has to be created using the Xilinx Project manager. All VHDL files except testbench.vhd have to be added to the project and synthesis can be started selecting FPGATOP as top-level file. For implementation the constraint file Sundance.ucf has to be used. It defines the pin-out of the FPGA module. Synthesis generates a configuration bit-stream file for configuring the FPGA.

- **Running DSP monitor application** The DSP monitor application *dspmtr* has to be downloaded to enable the configuration of the FPGA and to enable the communication host-to-FPGA.

- **Downloading the configuration to FPGA** The bit-stream generated by Xilinx implementation tools cannot be downloaded directly,

but has to be converted to a Sundance specific format using the program *bit2dat*, because configuration is done via commports and the CPLD attached to the FPGA. After converting, the host-application *hostapp* can be used to download the configuration to the FPGA. The accelerator starts immediately after downloading and the host-application can be used to check the current status of the FPGA, i.e. if the algorithm is still running or a solution has been found. Finally, the result of the computation is read back using *hostapp*.

6 Evaluation

6.1 Benchmark Problems

The architecture has been tested by simulation of the generated, instance specific VHDL code. For some benchmarks an FPGA configuration has been synthesized for validating the simulation results and gaining information about synthesis time and size of the synthesized circuit.

Benchmarking minimum covering algorithms and comparing the results to other algorithms has turned out more difficult than benchmarking SAT algorithms. For comparing SAT solving algorithms there exists a generally accepted benchmark set that goes back to the DIMACS SAT implementation challenge. In contrary to SAT, there is to my knowledge no comparable benchmark set for minimum covering problems. Therefore the benchmark problems for the accelerator had to found by some other way.

The approach was to start from an *Espresso*, which makes use of minimum covering and therefore can provide both: realistic benchmarks problems and a reference implementation of a software solver to compare the hardware implementation with. Espresso is the standard tool for the minimization of 2-level boolean logic. The algorithms used in Espresso base on transforming a representation of a boolean equation to a minimum covering problem. After the first round of reductions has been applied to the problem, the resulting covering matrix is called *cyclic core*. For solving this minimum covering problem an algorithm based on *exact_cover* given in algorithm 2 given on page 23 is used. Espresso has been instrumented, to dump the cyclic cores to a file and to measure the time for covering, isolated from pre- and post-processing steps.

For testing and benchmarking of the accelerator circuits, 21 small problems of the ESPRESSO distribution have been chosen. These test problems have from 6 to 45 variables and from 6 to 39 clauses. Since VHDL simulation is 10^3 to 10^6 times slower than the resulting circuit on the FPGA, only these smaller examples were tested, otherwise simulation time would be excessive.

The time Espresso used to find a minimum cover for the examples, measured on a workstation, was used as reference for comparing the performance of the hardware accelerator.

6.2 Simulation

All of the 21 examples have been simulated using the Modelsim VHDL simulator. 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 cycle accurate VHDL simulation and an assumed clock rate of 25 MHz. The software execution times for the chosen problems are in the range of $10ms$ to $2.13s$.

Table 4 lists the results and the characteristics of the benchmark examples. The columns vars, clauses and cost describe the size of the benchmark problem with regard to the number of variables and clauses of the associated CNF, cost refers to the cost of the solution of the problem. For comparing the influence of the different covering algorithms, for each problem 3 different architectures were implemented and measured: CE (clause evaluation and cost bound), CEDC (same as CE but with support for detecting don't cares), CEDCES (same as CEDC plus support for essential columns). The columns cd, cedc and cedces quote the number of cycles the alogrithm was running. The execution time on hardware was computed by taking the number of cylcles for the fastet architecture and assuming a clock frequency of 25 MHz for the resulting circuit. The raw speedup was calculated as $S_{\text{raw}} = t_{\text{sw}}/t_{\text{hw}}$ and does not include hardware compilation time.

Figure 14 summarizes the results of table 4 and shows the number of problems over the resulting raw speedup. 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 the measured accelerator uses only essentials and don't cares as reductions, and the current best cost as lower cost bound. The software covering algorithm employs more reduction techniques and an improved lower bound. Depending on the problem instance this can either drastically improve the performance or just add a large overhead.

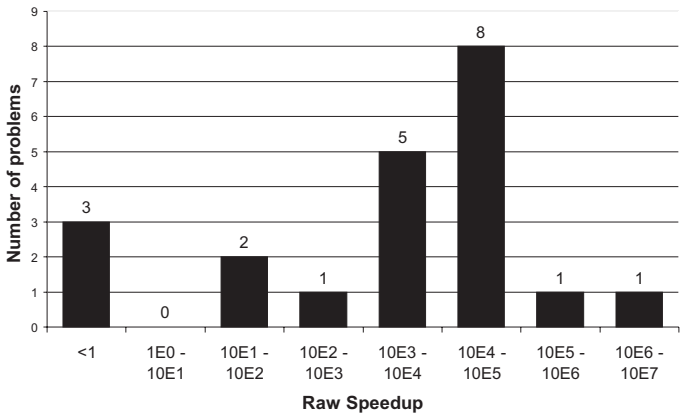

Figure 14: Raw speedup for the covering accelerator

6.3 Implementation

Several problems have been implemented on a Xilinx Virtex XCV1000-BG560-4 device. The VHDL source for the accelerator is the same as used

problem	vars	clauses	cost	ce	cedc	cedces	hw @ 25mhz	sw espresso	speedup
				[cyc]	[cyc]	[cyc]	[s]	[s]	
amd	27	25	12	4.05E+05	2.25E+05	1.00E+05	4.01E-03	6.00E-02	1.50E+01
bcc	12	12	6	6.12E+02	5.74E+02	2.06E+02	8.24E-06	1.00E-02	1.21E+03
chkn	6	6	3	6.40E+01	6.40E+01	1.70E+01	6.80E-07	2.00E-02	2.94E+04
dekoder	7	7	4	1.20E+02	1.08E+02	1.90E+01	7.60E-07	1.00E-02	1.32E+04
dk27	8	8	4	1.20E+02	1.20E+02	2.30E+01	9.20E-07	1.00E-02	1.09E+04
duke2.pla	16	16	8	2.71E+03	2.71E+03	8.09E+02	3.24E-05	4.00E-02	1.24E+03
in0	6	6	3	6.80E+01	6.80E+01	1.70E+01	6.80E-07	2.00E-02	2.94E+04
mark1	4	4	2	2.40E+01	2.40E+01	1.10E+01	4.40E-07	2.00E-02	4.55E+04
newcpla	5	5	3	5.20E+01	5.20E+01	1.50E+01	6.00E-07	3.00E-02	5.00E+04
newill	6	6	3	5.60E+01	5.60E+01	1.70E+01	6.80E-07	1.00E-02	1.47E+04
root	14	16	7	1.59E+03	1.43E+03	6.08E+02	2.43E-05	1.00E-02	4.11E+02
squar5.pla	20	20	10	1.37E+04	1.20E+04	5.55E+03	2.22E-04	1.00E-02	4.51E+01
table5.pla	7	7	4	1.16E+02	1.16E+02	2.10E+01	8.40E-07	1.00E-02	1.19E+04
tms	9	9	5	2.88E+02	2.82E+02	6.70E+01	2.68E-06	1.00E-02	3.73E+03
wim	7	7	4	1.24E+02	1.14E+02	1.90E+01	7.60E-07	1.00E-02	1.32E+04
bc0	8	8	4	1.20E+02	1.20E+02	2.30E+01	9.20E-07	2.13E+00	2.32E+06
b12	20	27	9	2.87E+04	1.97E+04	2.75E+03	1.10E-04	2.40E-01	2.18E+03
in4	9	10	4	1.24E+02	1.12E+02	2.30E+01	9.20E-07	1.20E-01	1.30E+05
f51m	14	15	7	1.16E+03	1.11E+03	2.71E+02	1.08E-05	8.00E-02	7.38E+03
m3	42	39	19	1.20E+08	4.06E+07	5.11E+06	2.05E-01	7.00E-02	3.42E-01
dist	45	41	20	5.45E+08	5.89E+07	1.29E+07	5.16E-01	6.00E-02	1.16E-01
in2	38	42	17	1.39E+07	9.38E+06	1.48E+06	5.94E-02	5.00E-02	8.42E-01

Table 4: Results from VHDL Simulation for Espresso Benchmarks

Criterion	dist	mark1
Variables	54	4
Clauses	41	4
Cost of solution	20	2
t_{espresso}	60 ms	20 ms
Slices used	1294 (10%)	451 (3%)
Flip Flops used	1305 (5%)	523 (2%)
Equivalent gate count	23481	8219
Max. Frequency	25.3 MHz	51.3 MHz
Details [1]		
LUTs used total	2361	736
LUTs used by checker	324	15
FFs used by checker	0	0
LUTs per statemachine	28	28
FF per statemachine	13	13

Table 5: Implementation Results for *dist* and *mark1*

for the simulation. The design has been synthesized and implemented using Xilinx Foundation Tools 3.1i on a Pentium-III 550 MHz system with 640 MB RAM. The hardware compilation times, including circuit synthesis, place, and route, are in the order of several minutes. Obviously, this renders hardware acceleration of small test problems useless. However, the target for this accelerator are hard and large problems with long software runtime. Further, the hardware compilation time can be reduced.

The implemented circuits achieve clock rates of 30–50 MHz, which shows that our assumption of the hardware running at 25 MHz was rather pessimistic.

To receive an impression of the I will exemplarily show the implementation results for two examples, *dist* and *mark1*. Variables, clauses and cost denote the size of the benchmark problem. The time epresso needs to solve the benchmark on a workstation is given by τ_{espresso}. The size of the resulting circuit is given by slices and flip flops used, the numbers in parentheses denote the fraction of the FPGA ressources that have been used. Max. Frequency states the maximum operating frequency. Both examples were synthesized with target clock frequency constrained to 25 MHz using low effort for synthesis and implementation. Since example mark1 is very small, the results of mark1 can be used as approximation of the offset for the size of the circuit, resulting from communication interfaces and reconfiguration control.

[1] as reported by FPGA express, there are minor deviations of the data reported by

Extrapolation of this data for other problems is not trivial as the correlation of variables and clauses and the size of the resulting checker is certainly not linear. The problem is not only characterized by the number of variables and clauses but also by the structure of the clauses themselves, for example how many variables are used per clause. As the code for the checker can be generated very fast, an explicit description of the checker logic can be deduced immediately. Using these equations an estimation of the size of the implementation getting an accurate estimation for the circuit size should be much easier. The drawback is that estimating the circuit size this way is quite complex. Extracting a rule-of-thumb estimation of the circuit size will require more data from actual implementations.

Xilinx Foundation and FPGA Express

7 Status and Conclusion

7.1 Status

The current status of the work is as follows:

- **VHDL code** The VHDL code implements all features of the accelerator architecture that have been described in this report.
- **Code generator** The code generator can generate checkers for the support of don't cares and essentials. Support for dominated columns is prepared, but the code for inference of the corresponding logic is not implemented yet. Currently no preprocessing steps, for instance reordering of the variables or eliminating dominant columns in the initial problem, are done.
- **DSP software** The monitor application on the DSP is completely implemented and tested. The program has to be uploaded using TI Code Composer.
- **Host application** The host application is implemented and tested. Currently the application supports only interactive usage. For complete automation of the solving procedure a non-interactive version can be derived easily.

7.2 Conclusion

Starting from the ideas that Platzner has presented for accelerating SAT using instance specific accelerators in [PM98], an architecture for solving SAT has been designed and implemented.

On the basis of the classical branch & bound algorithm for solving minimum covering problems *exact_cover*, it has been investigated if and how the ideas used by this software algorithm can be adopted to hardware. It has been shown that it is in fact possible to adopt many of the concepts of the efficient software solvers to hardware. Above all, most of the covering matrix reduction rules could be adopted by analyzing the relationship of covering matrix and boolean equation representation.

A code generator, which creates a configurable architecture that can support different strategies for variable deduction has been implemented. The architectures can be both, simulated and synthesized. An actual implementation consists of course of more than just the covering accelerator circuit, it needs also reconfiguration support and communication interfaces for downloading a new instance specific configuration and for sending the results of the computation. A complete prototype system has been implemented, that demonstrates all steps involved, from generating a VHDL description of the

accelerator from the problem description file, synthesizing a FPGA circuit, downloading it to the FPGA and reading back the solution.

Evaluating the results it has been shown, that the use of instance specific accelerators for solving minimum covering problems can lead to significant raw speedups. For some benchmark examples, speedups from several orders of magnitude could be achieved over runtime of the reference solver. For some examples, no speedup was achieved at all. As this shows the potential of the accelerator on the one hand, on the other hand it shows deficiencies of the current architecture as well.

It is obvious that it is hard to accelerate small problems of size as the benchmarks that I have used, since synthesis exeeds the time for actually solving the problem by far. For the target problems of this accelerator that are hard and large problems, synthesis time will become less important. Furthermore, it will be possible to target even smaller problems by reducing synthesis time by the methods that have been proposed.

Adding further matrix reduction rules to the architecture will improve the accelerator further. Potentially a large gain of performance could be achieved by using a sophisticated estimation for the additional cost of a solution descending the branch and bound tree.

Due to the restricted length for a diploma thesis, not all ideas could be implemented, but some ideas for enhancements of both, architecture and implementation are given in chapter 8.

8 Future Work

Algorithm Not all reduction rules used by software solvers have been adopted in hardware. It is not clear yet, wether all reduction rules can be implemented in hardware. Each implemented method will further reduce the search space.

A shortcoming of the current architecture is that is doesn't make use of a sophisticated *cost bound* condition. A good estimation of the additional cost of a solution given a partial variable assignment can lead to much tighter bounds and therefore increase speed.

Support for dominated columns has to be added. The architecture is already prepared for dominated column support, but the necessary logic to detect dominance is not be generated by the current version of the code generator. Since this report shows how to derive the dominated column conditions, this extension should not require much effort.

Implementation The implementation is left completely to the control of Xilinx design tools. As the accelerator does consist of very regular structures, an FPGA implementation could be certainly improved in terms of speed and area by making use of the this regularity and doing placement of components using domain specific knowledge.

A major problem when using the accelerator for smaller or medium sized problems is the rather *long synthesis* time. Since most of the blocks are either fixed or built up very regularly, they could be pre-synthesized. Due to the simplicity of the design, synthesis could be omitted completely, by directly writing net-lists that can be used for placement and routing.

Configuration Virtex FPGAs have a feature called partial reconfiguration, which allows parts of the circuit to be reprogrammed at runtime, i.e. while the device is running. A Java class library called *JBits* available from Xilinx can be used to gain very low-level access to the FPGA resources. Using JBits, configurations could be created without any synthesis by directly programming LUTs, Multiplexers, RAM and routing. Since JBits allows to generate bit-streams for partial reconfiguration, incremental synthesis using JBits and partial reconfiguration could be combined to greatly reduce the time from transforming a problem description into a runnable FPGA circuit and therefore making applicable the accelerator even for smaller problems.

Code Generation The tool for generating instance specific VHDL code could be extended to support any combination of architectural features for the accelerator. Up to now, only given combinations, i.e. CE, CEDC and CEDCES are supported.

Currently, no preprocessing on the problem description is done. The order in which variables are assigned to the statemachines has an influence

on how fast a solution is found. Heuristics could be used to find a good arrangement for the variables.

Tool-flow At the moment the tool-flow is not automated. For the ease of benchmarking and integrating the solver in a real application, the procedure of going from circuit generation via synthesis to the read-back of the results should be fully automated.

Partitioning of the problem Minimum covering problems can be easily split up into several smaller subproblems. In principal this is done at every branch step in the exact_cover algorithm, but the resulting subproblems are solved sequentially instead of parallel. As todays FPGAs are very large and an accelerator for a given problem may use only few of the FPGA resources, several instances of the architecture may be implemented concurrently, each of them searching for a solution in a subtree of the original problem.

9 References

References

[ADS00] 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.

[AdSS99] M. Abramovici, J.T. de Sousa, and D. Saab. A massively-parallel easily-scalable satisfiability solver using reconfigurable hardware. In *1999 Design Automation Conference (DAC)*, pages 684–690. IEEE, Piscataway, NJ, USA, 1999.

[AS97] Miron Abramovici and Daniel Saab. Satisfiability on Reconfigurable Hardware. In *7th International Workshop on Field-programmable Logic and Applications (FPL'97)*, pages 448–456. Springer-Verlag, Berlin, Germany, 1997.

[BFA96] 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 Filterung Using Reconfigurable Logic*, volume 2914, pages 225–236, 1996.

[DM94] Giovanni De Micheli. *Synthesis and Optimization of Digital Circuits*. McGrawHill, 1994.

[DMP99] Andreas Dandalis, Alessandro Mei, and Viktor K. Prasanna. Domain Specific Mapping for Solving Graph Problems on Reconfigurable Devices. In *Reconfigurable Architecures Workshop (RAW'99)*, 1999.

[DP60] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, (7):201–215, 1960.

[GPFW97] Jun Gu, Paul W. Purdom, John Franco, and Benjamin W. Wah. Algorithms for the Satisfiability (SAT) Problem: A Survey. In *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, volume 35: Satisfiability (SAT) Problem, pages 19–151, 1997.

[HM97] 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-Verlag, Berlin, Germany, 1997.

[HS00] Holger H. Hoos and Thomas Stützle. Local Search Algorithms for SAT: An Empirical Evaluation. *Journal of Automated*, 24:421–481, 2000.

[MP99] Oskar Mencer and Marco Platzner. Dynamic Circuit Generation for Boolean Satisfiability in an Object-Oriented Environment. In *32th Hawaiian International Conference on System Sciences HICSS-32*, January 1999.

[Pla00] Marco Platzner. Reconfigurable accelerators for combinatorial problems. *IEEE Computer*, 33(4):58–60, April 2000.

[PM98] Marco Platzner and Giovanni De Micheli. Acceleration of Satisfiability Algorithms by Reconfigurable Hardware. In Reiner W. Hartenstein and Andres Keevallik, editors, *8th International Workshop on Field-Programmable Logic and Applications*, number 1482 in Lecture Notes in Computer Science, pages 69–78, August/September 1998.

[RLHMS98] Azra Rashid, Jason Leonard, and William H. Mangione-Smith. Dynamic Circuit Generation for Solving Specific Problem Instances of Boolean Satisfiability. In *IEEE Symposium on FPGAs for Custom Computing Machines (FCCM'98)*, pages 196–204. IEEE Computer Society, Los Alamitos, CA, USA, 1998.

[SJ73] Earl E. Swartzlander JR. Parallel counters. *IEEE Transactions on Computers*, C-22(11):1021–1024, November 1973.

[SYN99a] T. Suyama, M. Yokoo, and A. Nagoya. Solving satisfiability problems on FPGAs using experimental unit propagation. In *5th International Conference on Principles and Practice of Constraint Programming (CP'99)*, volume 1713 of *Lecture Notes in Computer Science*, pages 434–445. Springer, 1999.

[SYN99b] T. Suyama, M. Yokoo, and A. Nagoya. Solving satisfiability problems on FPGAs using experimental unit propagation heuristic. In *Parallel and Distributed Processing. 11th IPPS/SPDP'99 Workshops held in conjunction with the 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing*, pages 709–711. Springer-Verlag, Berlin, Germany, 1999.

[SYS96] Takayuki Suyama, Makoto Yokoo, and Hiroshi Sawada. Solving Satisfiability Problems on FPGAs. In *6th International Workshop on Field-Programmable Logic and Applications (FPL'96)*, pages 136–145. Springer-Verlag, Berlin, Germany, 1996.

[SYS98] T. Suyama, M. Yokoo, and H. Sawada. Solving satisfiability problems using logic synthesis and reconfigurable hardware. In *31th Hawaiian International Conference on System Sciences (HICSS-31)*, volume 7, pages 179–186. IEEE Computer Society, Los Alamitos, CA, USA, 1998.

[YSLL99] Wong Hiu Yung, Yuen Wing Seung, Kin Hong Lee, and Philip Heng Wai Leong. A Runtime Reconfigurable Implementation of the GSAT Algorithm. In *9th International Workshop on Field Programmable Logic and Applications (FPL'99)*, pages 526–531. Springer-Verlag, Berlin, Germany, 1999.

[YSS96] 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)*, pages 497–509. Springer-Verlag, Berlin, Germany, 1996.

[ZAMM98] Peixin Zhong, Pranav Ashar, Sharad Malik, and Margaret Martonosi. Using reconfigurable computing techniques to accelerate problems in the CAD domain: a case study with Boolean satisfiability. In *35th Design Automation Conference (DAC)*, pages 194–199. IEEE, New York, USA, 1998.

[ZMA00] Peixin Zhong, Margaret Martonosi, and Pranav Ashar. FPGA-based SAT solver architecture with near-zero synthesis and layout overhead. *IEE Proceedings on Computers and Digital Techniques*, 147(3):135–141, May 2000.

[ZMAM98a] Peixin Zhong, Margaret Martonosi, Pranav Ashar, and Sharad Malik. Accelerating Boolean Satisfiability with Configurable Hardware. In *IEEE Symposium on FPGAs for Custom Computing Machines (FCCM)*, pages 186–195. IEEE Computer Society, Los Alamitos, CA, USA, 1998.

[ZMAM98b] Peixin Zhong, Margaret Martonosi, Pranav Ashar, and Sharad Malik. Solving Boolean Satisfiability with Dynamic Hardware Configurations. In *8th International Workshop on Field-Programmable Logic and Applications (FPL'98)*, pages 326–335, 1998.

[ZMAM99] 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.

[ZMMA97] Peixin Zhong, Margaret Martonosi, Sharad Malik, and Pranav Ashar. Implementing Boolean Satisfiability in Configurable Hardware. In *Logic Synthesis Workshop*, May 1997.