Automatic Structuring and Optimization of Hierarchical Designs

Heinz-Josef Eikerling, Wolfgang Rosenstiel
Universität Tübingen
Technische Informatik
Sand 13
D-72 076 Tübingen, Germany
{eikerlin,rosenstiel}@peanuts.informatik.uni-tuebingen.de

Abstract

In this paper an approach for the optimization of digital synchronous designs is described. The optimization is done for smaller components which are the result of a partitioning process. The actual optimization is done on a graph which reflects the communication structure between the modules. Sequential don't care conditions are extracted and used for sequential optimization. As experimental results show, the robustness of the subsequent logic synthesis methods can be increased while achieving a significant gain in cost and power consumption. This is shown by applying the described methods to a set of benchmarks obtained from high-level synthesis.

1 Introduction

State-of-the-art synthesis systems are concerned with the optimization of large, digital designs which are produced by higher level synthesis tools. Normally, a hierarchical, structural description at the RT-level is given which is described in an HDL. The description is passed to logic synthesis and stepping through other activities the design becomes manufacturable.

The outcome of the higher level synthesis tasks is a structural (RTL) description which consists of a set of connected entities. For instance, on the top level the RTL description can be roughly divided into two parts, the *controller* and the *datapath*. Both parts are connected by status and command lines. Since especially in the presence of arithmetic operators or if a large number of entities is encountered, appropriate partitioning methods for the datapath have to be devised. At the boundaries of the parts of the decomposed datapath a certain potential for optimization can be expected. Boundary optimization in current synthesis systems is reduced to the propagation of constants from one entity to the next. The exploitation of these don't care (DC) conditions at the hierarchy boundary of the datapaths parts is limited.

If we incorporate the controller into the optimization procedure, the extraction and treatment of sequential DCs. This is not directly applicable to large systems of interacting components due to complexity reasons comes into mind. However, the conditions provide an additional option for sequential optimization of the generated parts.

This paper deals with the automatic optimization of this kind of systems. We propose a two-stage approach. First, the complexity of the problem is reduced by applying a partitioning strategy which exploits the hierarchy of operators and the regularity of repetitive interconnection patterns in the datapath. Second, we use symbolic methods for DC extraction (co-simulation) which can be used by the subsequent logic optimization task.

1.1 Previous and Related Work

As pointed out above, the task consists of partitioning a digital synchronous system into a set of blocks and subsequently optimizing this set of interacting components.

Apart from the technical side which is extensively summarized in [9], partitioning of RT-level descriptions has mainly focused on partitioning of combinational logic. *Camposano* [3] proposed a method to minimize the number of blocks for the subsequent logic synthesis task. For the actual partitioning (clustering) a similarity function is used which is defined for low-level (atomic) operators. Hierarchical building blocks and sequential dependencies among the connected modules are not considered.

The decomposed structure can be regarded as a set of interacting finite state machines. The problem of optimizing these systems is frequently referred to as hierarchical sequential synthesis. *Watanabe* [12] and *Damiani* [5] independently proposed schemes for the optimization of interacting components modeling one component as a non-deterministic finite state machine. However, both techniques are limited to the optimization of state machines with up to 100 states only. Moreover, they are not applicable to datapath components because datapaths components (e.g. registers with load enable) constitute rather dense transition systems which hardly can be handled by explicit methods. Therefore, we propose to use subsets of the entire DC set which can be computed efficiently using a co-simulation approach.

1.2 Outline

The paper is organized as follows. First, we describe the hierarchical partitioning algorithm which is dedicated to the datapath logic. For reasons of efficiency, the partitioning step is not carried out on the netlist directly. Instead, a weighted graph model is employed. Second, we describe how to incorporate the controller into the optimization

EURO-DAC '96 with EURO-VHDL '96
0-89791-848-7/96 $4.00 ©1996 IEEE

process. Finally, we give an overview of the actual implementation and present some experimental results.

2 Partitioning

2.1 Design Representation

The input of the partitioning task is a *hierarchical netlist* $N = \{C_1, \ldots, C_n\}$, i.e. a sequence of hypergraphs or *cells* $C_i(V_i, E_i)$ (instances of these cells are referred to as *cell instances* $CI \in V_i$) which are connected by *wires* in the set E_i. Pins are the terminal nodes in V_i. Inputs and outputs of the cells are referred to as *pins* (corresponding instances at the cell instances are called *pin instances*). In the hypergraphs $C_i(V_i, E_i)$ wires are running between pins and pin instances. Each cell except for leaf cells in the hierarchy are composed out of instances of other cells. We maintain a *hierarchy graph* $H(N, D)$ in which a cell C_i is connected to a cell C_j if there is an instance CI with $cell(CI) = C_j$ being instantiated in C_i. Each cell instance contains a pointer $cell(CI) \in \{C_1, \ldots, C_k\}$ to its corresponding cell.

$H(N, D)$ is a dag. At the top-level we distinguish two main cells: the datapath and the controller. We refer to $C_d, d \in \{1, \ldots, n\}$ as the datapath cell. The index d is given by a topological sorting of the nodes in $H(N, D)$. The graph $H(N, D)$ can be modified by expanding cell instances and by combining cell instances to form new cells. Each cell may be composed out of other cell instances constituting to the overall behavior. The controller can be represented either by a symbolic state transition table or by an implementation (netlist). For the datapath we concentrate on a mux-based design, i.e. the routing of data is done via muxes.

2.2 Optimization Objectives

We will concentrate on the following criteria for optimization:

- $Area(C)$ is the area of cell C which is given by summing up the sizes of gates and latches required to implement C. For reasons of simplicity, we do not take wiring area into account.
- $Power(C)$ describes the power consumption of a module C. Notice, that this criterion normally depends on the target technology and the set of input patterns on which the module will operate, i.e. this may vary for each cell instance CI with $cell(CI) = C$. In our application we consider the power dissipation due to switching activity. For this, a CMOS implementation is assumed. We assume static probabilities of 0.5 for an input pin of the cells. In this case, we achieve an additive quantity and can study design alternatives on average sets of input patterns.
- $Comm(C)$ is the communication of the cell C with the environment. By the minimization of the communication also the performance can be increased because interconnections between modules can be kept short.

These quantities are described for cells and cell instances, i.e. for measure $m \in \{Area, Power, Comm\}$ we have $m(CI) = m(cell(CI))$. In the further description we will refer to $m(CI_1, CI_2)$ as the characteristic for the module which results out of the combination of the instances CI_1 and CI_2.

2.3 Problem Formulation

Restructuring or partitioning amounts to finding a partitioning of the datapath into blocks of manageable size without deteriorating the characteristics of the design too much by separating parts of the design. The size of the block is assumed to be an indicator for the complexity of the subsequent logic synthesis step.

Input. A hierarchical netlist $N_d = \{C_d, \ldots, C_n\}$. We have weighting functions $Area: N_d \to \mathbb{R}^+$ for the estimated size and $Seq: N_d \to \mathbb{N}$ for the number of sequential elements of a cell C. For all pairs of cell instances CI_1 and CI_2 in the netlist which are connected by a wire and each measure $m \in \{Area, Power, Comm\}$ we assume a weighting function to be given ($\alpha_m \in [0, 1]$)

$$c(CI_1, CI_2) = \sum_m \alpha_m \cdot \left(1 - \frac{m(CI_1, CI_2)}{m(CI_1) + m(CI_2)}\right)$$

We have functions $bound_A: \mathbb{R}^+ \to \mathbb{R}^+ \cup \{\infty\}$ and $bound_S: \mathbb{N} \to \mathbb{N} \cup \{\infty\}$ to constrain the size of the blocks in terms of area and number of sequential elements.

Output. Let C_d' be the cell which results out of C_d by iteratively expanding cell instances. The output is a disjoint decomposition $\Pi_{C_d'} = \{S_{CI,1}, \ldots, S_{CI,p}\}$ of the set of instances S_{CI} in C_d'.

Optimize. The objective of optimization is (1) the minimization of the number of partitions p, (2) of the cut size

$$c(\Pi_{C_d'}) = \frac{1}{2} \cdot \sum_{i=1}^{k} \sum_{CI_1, CI_2 \in Cut_i} c(CI_1, CI_2),$$

and (3) of the block constraint penalty function

$$r(\Pi_{C_d'}) = \sum_{i=1}^{p} (bound_A(a)) + \sum_{i=1}^{p} (bound_S(s))$$

Because the number of blocks which have to be determined by the partitioner is not fixed, this type of problem is called *free netlist partitioning problem (FNPP)*. The corresponding threshold variant is a NP-complete problem [7], even for very simple and restricted issues of this problem, i.e. only the flat hierarchy is considered, all node and edge weights are uniform ($c \equiv 1, Area \equiv 1, Seq \equiv 1$) and a static bound function is used for $bound_A$ and $bound_S$:

$$bound_X^{\,|}(x) = \begin{cases} 0 & , \text{if } x \le X \\ \infty & , \text{else.} \end{cases}$$

Therefore, an heuristic approach is justified.

2.4 Partitioning Library

For the partitioning step we need information on how to decrease the cut size by merging two cell instances into one partition. Moreover, we have to provide information about the design characteristics of the modules.

Design Information. For each cell, the design library contains information about the area and power consumption parameterized on the number and type (input, output, both) of the pins and the wordlengths of these pins. For meta-cells (swapping, splitting and merging of bus lines) no design information is being provided since incorporating the corresponding cell instances into any partition only changes the communication cost which can be evaluated statically out of the context of the instance.

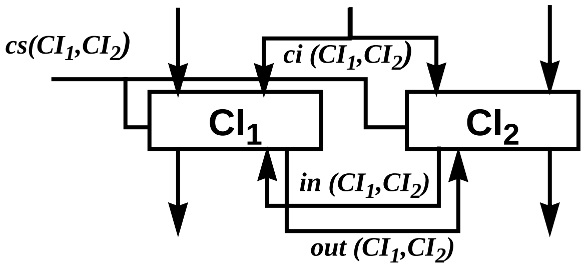

Figure 1. Connection types for cell instances.

The information about the design data of a particular element of the library is stored in a dictionary. The data can be accessed via a string which is uniquely assigned to each cell. This data may heavily depend on the particular script which is used for logic synthesis and on the gate library onto which the RTL description will be mapped. By maintaining a persistent version of the dictionary we achieve flexibility concerning these parameters.

Partitioning Information. The partitioning library gives information on how the characteristics $Area(CI_1, CI_2)$, $Power(CI_1, CI_2)$ and $Comm(CI_1, CI_2)$ decrease if two instances CI_1 and CI_2 are merged.The values are given by considering all possible connection types occurring in the actual datapath implementation.

The information is determined a priori by passing the combined cell instances to logic synthesis running a default script. The connection topology has a big influence on the design characteristics of the merged instance. For keeping the partitioning library as small as possible we proceed by defining some rules for abstracting from the actual shape of the connection topology:

(1) We define an ordering on the set of cells $<$. This is done by analyzing the string which is associated with the cells of the connected instances CI_1 and CI_2.

(2) The set of nets (except for the clock) connecting the instances CI_1 and CI_2 are classified as follows: $ci(CI_1, CI_2)$ is the set of common input nets used by both instances, $in(CI_1, CI_2)$ and $out(CI_1, CI_2)$ describe the set of nets which are connecting the outputs of CI_1 with the inputs of CI_2 and vice versa assuming $cell(CI_1) < cell(CI_2)$. $cs(CI_1, CI_2)$ is the set of common control lines from the controller. The set of all possible connection pattern is shown in figure 1. In figure 2 an example for different connection types leading to different characteristics when merging is shown.

Normally, due to the high regularity of the generated structure, not all connection patterns will occur which significantly reduces the storage requirements for maintaining the partitioning information. For instance, the output of a functional unit and the selector channel of a multiplexor or decoder and components with different wordlengths will never be connected. Similar to the access of cell characteristics we also define a dictionary for the partitioning data. A connection of two instances is mapped to a string which consists of the strings of the corresponding cells and cardinality of the net types $ci(CI_1, CI_2)$, $cs(CI_1, CI_2)$, $in(CI_1, CI_2)$ and $out(CI_1, CI_2)$.

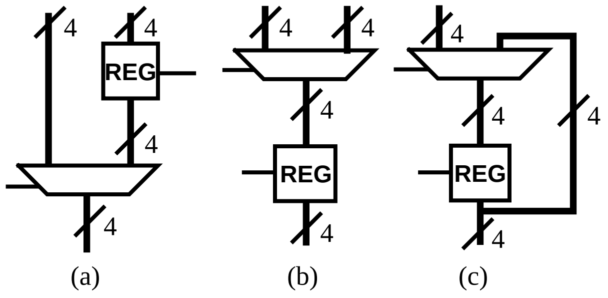

Figure 2. Connection types for a multiplexor/register with enable combination. All combinations are distinct compared to all other instances. Connection types are [0,0,4,0] in (a), [0,0,0,4] in (b) and [0,4,0,4] in (c). The corresponding relative areas are 82 (a), 79 (b) and 62 (c).

This abstraction becomes clear if we notice that most connections run between multiplexors and registers or multiplexers and functional units. For instance, it does not make a distinction to what particular mux-input a register output is being connected. For the representation of both, the design and partitioning data base, randomized search trees have been used which give fast access ($O(1)$) and update times ($O(\log n)$ where n is the number of entries).

2.5 Partitioning Algorithm

Due to the reasons pointed out, we propose a heuristic approach for the solution of the free partitioning problem.

2.5.1 Restructuring of the Datapath

In some cases it is useful to replicate cell instances if the decrease concerning one measure is high compared to the increase in the other criteria. There exists a trade-off because excessive replication of instances will also increase the input size. In our methodology, replication is done for all cell instances representing constants because this does not influence the cost for wiring, the area and/or the power consumption. However, these instances significantly contribute to the sequential DC set (set of values observable at sequential elements).

In the first stage the sizes of the instances CI in the datapath C_d are estimated by considering the information in the design dictionary. The contribution to the total area is $Area(CI)$. However, sometimes the cells are too big to be fed into logic synthesis directly (e.g. if a 32-bit signed integer array multiplier has been used). If this happens, we need to generate a proper area estimate which is done by analyzing the hierarchy tree $H(N, D)$. For this, we recursively compute an estimate for the cell instances in $cell(CI)$. Let the estimated quantity be $Area^{est}(CI)$. If $Area^{est}(CI) > A$

is true we cannot expect to achieve a partitioning which is correct with respect to the block size constraint. We expand one level of the hierarchy in CI and examine the expanded instances $CI_1', \ldots, CI_e'$. The expansion step is carried out for all cells for which $Area^{est}(CI_j') > A$ is true. The result is the cell C_d'.

2.5.2 Graph Construction

The partitioning is carried out on an intermediate model which we refer to as the *connectivity graph*. For each instance CI in C_d' a node v is inserted. A weight $w_{node}(v) = \langle Area(CI), Seq(CI) \rangle$ is assigned to each node which describes the approximated size of the vertex in the top-level datapath cell. Two nodes are connected via an edge in the connectivity graph if the corresponding instances are connected in the netlist. The weight of an edge is given by $w_{edge}(v_1, v_2) = c(CI_1, CI_2)$. We will denote the connectivity graph by G_c.

2.5.3 Graph Partitioning

For partitioning G_c we have implemented a problem-specific genetic algorithm (GA) similar to the one described in [1]. Due to lack of space we can only present the essentials. The population is encoded by sorting the set (a) of edges and (b) of nodes in G_c. We assign a solution to each member of the population (chromosome) by passing it to a fast problem specific heuristics. Depending on the type of sorting of the objects (nodes or edges) we can select among a node oriented and a edge oriented method. A gene represents the value, which is associated with the object leading to the order. The edge oriented method works as follows: initially, all nodes form a cluster. Run through all edges $e = \{v_i, v_j\}$ in the given order. Merge the blocks $\Pi(v_i)$ and $\Pi(v_j)$ if the resulting size and number of sequential elements of the new block is acceptable. The node oriented method works similarly.

Different results can be achieved by perturbing the set of edges and nodes. This perturbation is done during the computation of the initial population and during runtime by applying the genetic operators (point mutation and one-site crossover). The GA works as follows: first, the initial population is determined. As long as the stopping criterion is not fulfilled, the chromosomes of the current population are scanned and cost and fitness values are computed. The best two chromosomes are always copied into the new population. During a reproduction step, two chromosomes are selected based on their fitness, the cross-over operator is applied with probability $Prob_c \in [0, 1]$ and the result is copied into the new population, unless both populations reach the same size. After applying the cross-over operator the mutation operator is applied with probability $Prob_m \in [0, 1]$ to each gene in the entire new population.

We have used the operator

$$\Phi(\Pi_1, \Pi_2) = \sum_{M \in \{cut, p, r\}} \kappa_M \cdot M_{rel}(\Pi_1, \Pi_2)$$

for comparing different partitionings Π_1, Π_2 of G_c. The *relative cut size* in the formula is given by

$$cut_{rel}(\Pi_1, \Pi_2) = \frac{cut(\Pi_1) - cut(\Pi_2)}{cut(\Pi_1) + cut(\Pi_2)}$$

if $cut(\Pi_1) + cut(\Pi_2) \neq 0$ and 0 otherwise. Similarly the relative values p and r can be computed.

Depending on the characteristics of G_c we can either choose the node or the edge oriented method. Since this is mostly the case and the runtime of the method is reasonable, especially when compared to the subsequent logic synthesis task, we propose to run both methods and pick the best result. κ_M is set to 100% for all measures. With our implementation, the best results were achieved by setting $Prob_m = 0,001$, $Prob_c = 0,6$ and the population size and number of generations to 100. The function

$$bound_X^{\prime}(x) = \begin{cases} 0 & , if\ x \leq X \\ e^{2(a-A)/A} - 1 & , else. \end{cases}$$

is used for bounding the size and the number of sequential elements in each partition.

3 Sequential Optimization

3.1 Don’t Care Extraction

The outcome of the prior partitioning task is a functional decomposition of the datapath which models a *deterministic, synchronous system*. The generated logic blocks can be directly transferred to logic synthesis. Better results can be achieved by considering the sequential DCs arising from unreachable and equivalent states. However, even the application of symbolic techniques [2] which were rather successful for datapath components fails if complex components (e.g. multipliers) are used. Even data abstraction [4] does not help in this case, since we have to specify the DC conditions at the Boolean level.

3.2 Symbolic Co-simulation of the Controller

The gain of sequential DCs arising from pure datapath portions is limited since sequential components (e.g. registers) will normally store all data available at the primary inputs. Therefore, all states will be reachable. Moreover, the decision upon how to process data depending on the value of a particular variable is made in the controller. Therefore, considering the datapath components separately equivalent state pairs are seldom to occur unless sophisticated feedback between branch arbitration logic (e.g. comparator) and data routing (e.g. multiplexor) is partly implemented in the datapath. Normally, the controller constitutes the arbitration logic. Therefore, for sequential DC generation the controller needs to be involved in the optimization procedure.

The DC generation proceeds as follow. Assume, we have computed a p-way partitioning of the datapath. First, the controller is tentatively duplicated p times and is added to all generated partitions as shown in figure 4. By partitioning the datapath also the status and command lines can be partitioned into disjoint sets of control and status signals. The local controller can now be minimized with respect to these reduced sets of signals.

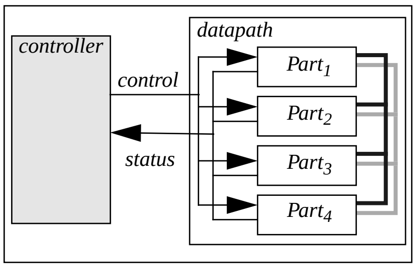

Figure 3. Result of partitioning.

Reachable states. Let n_{cp} and n_{dp} be number of latches (state variables) in the i -th part. The set of states in the partitions are given by $S_{cp} = \mathbb{B}^{n_{cp}}$ (controller) and $S_{dp} = \mathbb{B}^{n_{dp}}$ (datapath). The set of the product states is given by $S = S_{cp} \times S_{dp}$. Let $s = (s_{cp}, s_{dp})$ denote a product state. The set of reachable product states $S_R(s_{cp}, s_{dp})$ of the product machine starting from the reset state can be computed by an efficient symbolic state space traversal [10] based on BDDs. By existentially quantifying out the state variables of the controller in $S_U = S - S_R$, we obtain the set of reachable states of the datapath part

$$S_U(s_{dp}) = \exists(s_{cp} \in S_{cp})(S_U(s_{cp}, s_{dp}))$$

By quantifying out the state variables of the datapath the unreachability DCs of the augmented local controllers can be gained. Assume, that $S_{U,1}(s_{cp}), \ldots, S_{U,p}(s_{cp})$ is the amount of unreachable states which have been computed for the local controllers during the p simulation steps, then the set of unreachable states for the lumped controller is given by intersecting all these sets

$$S_U(s_{dp}) = \prod_{i=1}^{p} S_{U,i}(s_{cp})$$

Equivalent states. Let $E(s, s') = E(s_{cp}, s_{dp}, s'_{cp}, s'_{dp})$ describe the equivalence relation of the product states of a partition. Then

$$E(s_{dp}, s'_{dp}) = \forall(s_{cp}, s'_{cp} \in S_{cp})E(s_{cp}, s_{dp}, s'_{cp}, s'_{dp})$$

is the set of all equivalent state pairs of the datapath. Analogously, the sets $E_1(s_{cp}, s'_{cp}), \ldots, E_p(s_{cp}, s'_{cp})$ can be used to compute an approximation of the equivalent state pairs of the lumped controller which is given by

$$E(s_{cp}, s'_{cp}) = \prod_{i=1}^{p} E_i(s_{cp}, s'_{cp})$$

Combinational Don't Cares. Besides the sequential DCs we can also specify the combinational DCs at the controller outputs by computing the range of the functions bound to the control lines. Since the status lines do not intersect, the external DC set for the controller is given by the disjunction of the ranges of the functions at the outputs of the datapath components.

Sequential Optimization. Before passing the structure to logic synthesis, one can try to use more sophisticated sequential optimization techniques to obtain an optimized implementation. In [6] a synthesis procedure is described which works entirely on the implicit representation. A simple transformation which can be performed quickly is to remove the set of redundant latches.

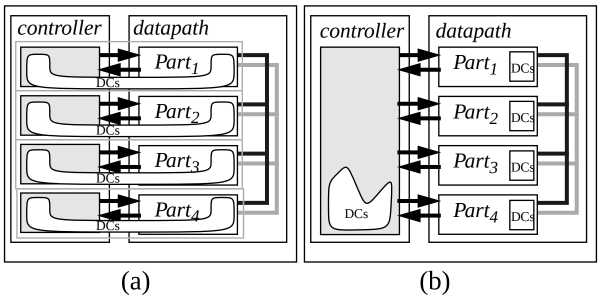

Figure 4. Symbolic co-simulation step for sequential DC exctraction.

4 Implementation and Experimental Results

We have implemented two tools. *REPART* is the partitioning tool which can be executed. Files can be read in BLIF or VHDL format. The partitioning and design information are read from a file. The sequential resynthesis system (*SRS*) and logic synthesis parts are separated processes. For logic synthesis a pipe mechanism has been implemented for fast communication between *REPART* and the logic synthesis system *SIS* [11]. The partitioning library can be generated on demand if no appropriate design data and/or partitioning information is available.

On this basis a couple of experiments have been carried out. First of all, the partitioning tool has been used for the structuring of datapaths in order to facilitate or even to improve runtime of logic synthesis on datapaths obtained from the high-level synthesis system *PMOSS* [8] which targets a mux-based architecture. The benchmarks are either standard high-level synthesis benchmarks (gcd, ellip, diffeq) or taken from experiments which were carried out in a codesign scenario (fibo, atoi) [8]. During behavioral synthesis, hardware resources are allocated out of the library which is given by a set of C++ generators.

Table 1 shows the characteristics of the datapaths in terms of primary input and output count. The next column shows the area estimate which was gained by summing up the sizes of the optimized instances in the datapath. Note, that this gives an upper bound on the size of the datapath. The next 3 columns give the results for the flat synthesis of the datapath which was impossible to compute for the bigger problem instances. For optimization the rugged script has been used. Next the same data for the partitioned version was given using the same script. We have used bounds of $A = 1000$ and $S = 20$ in $bound_A^{\uparrow}$ and $bound_S^{\uparrow}$. For smaller benchmarks, these values were taken such that a bipartitioning resulted. The time limit for synthesis was set to 1000 CPU seconds. The weights α_m were set to 100%. Finally, we give the results which were obtained after partitioning and sequential optimization. The given runtime data includes time for partitioning, time for sequential optimization and logic synthesis. As can be seen, for most examples the partitioned design optimized by ordinary Boolean techniques comes close to the result for the flat (global) synthesis and it is significantly better than the

	PIs/POs	Est. Area	Synthesis (flat)			Synthesis (partitioned)				Sequential Optimization		
Benchmark	(dp.)	(hier.)	Area / Power / Synth. Time			Area / Power / Synth. Time			p	Area / Power / Synth. Time		
fibo04	11 / 5	436.0	337.0	1142.4	10.5	362.0	1203.7	4.7	2	321.0	1014.3	8.1
fibo08	15 / 9	1003.0	684.0	2293.6	10.7	724.0	2400.7	20.7	2	662.0	2105.1	25.0
fibo16	23 / 17	2003.0	1371.0	4579.1	22.7	1452.0	4794.7	16.6	3	1398.0	4587.3	120.1
fibo32	39 / 33	3740.0	2748.0	9173.8	54.5	2811.0	9305.6	58.6	4	2712.0	9034.3	64.8
gcd04	17 / 6	399.0	352.0	1191.8	7.1	375.0	1402.7	7.0	2	340.0	1063.2	10.3
gcd08	25 / 10	775.0	664.0	2260.2	15.0	685.0	2509.3	13.2	2	644.0	2213.3	17.0
gcd16	41 / 18	1523.0	1283.0	4407.9	44.9	1297.0	5380.0	28.4	2	1158.0	4103.3	35.8
gcd32	73 / 34	3452.0		- time-out -		2487.0	11.18.7	78.0	4	2238.0	10437.7	87.0
atoi04	22 / 11	649.0	471.0	1630.4	7.2	480.0	1654.0	7.3	2	445.0	1362.7	8.0
atoi08	30 / 20	1297.0	925.0	3240.3	20.7	949.0	3678.0	11.2	2	871.0	3043.4	15.4
atoi16	46 / 37	2589.0	1863.0	6396.9	64.7	1995.0	8096.0	16.1	3	1798.0	6086.0	20.6
atoi32	78 / 70	5961.0		- time-out -		4010.0	15798.0	150.4	6	3867.0	13641.3	186.3
ellip04	69 / 33	1558.0		- time-out -		1576.0	5208.0	62.8	2	1515.0	5113.4	78.2
ellip08	101 / 65	2920.0		- time-out -		3044.0	10096.3	126.0	3	3044.0	10101.3	158.0
ellip16	165/129	5840.0		- time-out -		5749.0	18973.7	240.9	6	5696.0	17185.4	267.3
ellip32	293/257	17003.0		- time-out -		11108.0	19867.5	487.6	17	11000.0	39897.5	500.5
diffeq04	44 / 13	1518.0	1301.0	3266.0	82.6	1382.0	3346.0	43.7	2	1250.0	3034.0	98.0
diffeq08	44 / 13	3342.0	2742.0	6669.8	450.6	2701.0	6605.0	105.2	4	2722.0	6699.0	133.0
diffeq16	44 / 13	7789.0		- time-out -		5332.0	11510.7	187.5	8	5196.0	10413.7	283.3
diffeq32	44 / 13	22521.0		- time-out -		11525.0	24164.0	407.6	23	11163.0	23519.0	854.5

Table 1. Results of hierarchical optimization process. All data refers to an implementation using the mcnc.genlib assuming a 20 MHz clock. Power consumption is measured in μW assuming V_{dd} to be 5V. Area refers to grid count of a standard cell implementation. Synthesis time refers to CPU seconds on a Sun Sparc SS20-M712.

result for the hierarchical synthesis (3rd column). The number of generated partitions is shown in the 10th column. For the larger examples, an entire treatment of the datapath is not feasible. Note, that for all designs a significant acceleration of the synthesis step can be achieved. In all cases, the application of the sequential optimization option gives significant gains in cost and power consumption (last 3 columns).

5 Conclusions and Future Work

We have presented a method for the optimization of hierarchical datapaths which exploits the hierarchy of the library cells during the partitioning process. This is because the sizes of the components may vary significantly and therefore instances will have to be partitioned in order to facilitate logic synthesis. Moreover, we have shown how to integrate the controller into the optimization procedure which is used in order to get proper sets of DCs for the generated functional descriptions. The presented approach was mainly dedicated to the implementation of the design as a multi-level circuit, i.e. as an ASIC. However, one could also think of tailoring the presented methods to map the circuit description to FPGAs. Therefore, one will have to encompass the knowledge about how to efficiently combine partitions in order to obtain cost efficient FPGA implementations into the partitioning process.

Acknowledgements

This work has been supported by Deutsche Forschungsgemeinschaft under grant SFB 358, project B3.

References

[1] I. Ahmad and M. Dhodhi. On the m-Way Partitioning Problem. *The Computer Journal*, 38(3):237–244, 1995.

[2] R.E. Bryant. Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.

[3] R. Camposano and R. Brayton. Partitioning before Logic Synthesis. In *Proc. of the International Conference on Computer-Aided Design*, pages 324–326, Santa Clara, CA, November 1987.

[4] F. Corella, M. Langevin, E. Cerny, and et. al. State-Enumeration with Abstract Descriptions of State Machines. In *Proc. of the CHARME'95*, pages 146–159, Frankfurt, 1995. IFIP WG10.5.

[5] M. Damiani. Nondeterministic Finite-State Machines and Sequential Don't Cares. In *Proc. of the European Conference on Design Automation*, pages 192–198, Paris, France, 1994. IEEE.

[6] H.-J. Eikerling, M. Schmidt, and W. Rosenstiel. A Fully Symbolic Framework for Area-minimizing Hardware Resynthesis. In *IFIP Workshop on Logic and Architecture Synthesis*, pages 163–171, INPG, Grenoble, France, December 19-20 1995.

[7] M. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman, 1979.

[8] W. Hardt and W. Rosenstiel. Speed-Up Estimation for HW/SW-Systems. In *CODES/CASHE*, Pittsburgh, PA, March 1996.

[9] Th. Lengauer. *Combinatorial Algorithms for Integrated Circuit Layout*. Teubner-Wiley, 1990.

[10] B. Lin, H. Touati, and A.R. Newton. Don't Care Minimization of Multi-Level Sequential Logic Networks. In *Proc. of the International Conference on Computer-Aided Design*, pages 414–417, Santa Clara, CA, 1990.

[11] E.M. Sentovich, K.J. Singh, C. Moon et al. Sequential Circuit Design Using Synthesis and Optimization. In *Proc. of the International Conference on Computer Design*, pages 328–333, Cambridge, MA, 1992.

[12] Y. Watanabe and R. Brayton. State Minimization of Pseudo Nondeterministic FSM's. In *Proc. of the European Conference on Design Automation*, pages 184–191, Paris, France, 1994.