Resource Sharing in Hierarchical Synthesis*

Oliver Bringmann and Wolfgang Rosenstiel

Forschungszentrum Informatik an der Universität Karlsruhe (FZI)
Haid-und-Neu-Str. 10-14, 76131 Karlsruhe, Germany
and Universität Tübingen, Sand 13, 72076 Tübingen, Germany

Abstract

This paper presents a new approach to hierarchical high-level synthesis with respect to internal register-transfer structures of complex components. Entire subdesigns can efficiently be used as complex components at a higher hierarchical level of the design. After synthesis, the calculated schedule of each subdesign is added to its register-transfer component model. This enables the sharing of unused subcomponents across different hierarchical levels of the design. Especially, subcomponents of autonomous components, with a separate controller, can also be shared. As a result, the presented methodology offers a high degree of optimization to hierarchically specified designs.

1 Introduction

In modern system design, the specification of complex systems, which can be hierarchically composed of several subsystems is becoming increasingly important. In this context, the complexity of the subsystems, also called components, is increasing as well. Examples for such subsystems are microprocessor cores, application specific functional units (e.g. DCT, FFT), and interface controllers. However, state-of-the-art high-level synthesis systems produce insufficient results in terms of quality of the result and execution time when considering large applications [1].

This paper addresses the problem of optimized integration of already synthesized system specifications as complex register-transfer components, in the further high-level synthesis flow. For this, it is important that the behavioral specification, the synthesized register-transfer structure, and the already determined schedule of the used components are known during further high-level synthesis steps. Among others, the essential features of such components are the visible, hierarchical composed component structure, the usage of a separate controller, and the data-dependent timing. Hence, such components can be autonomous in the entire system. Finally, components should have the capability to share subcomponents with other components across different hierarchical levels, with respect to their visible structures. If necessary, the used components may be modified during synthesis of the enclosing system. This allows an efficient specification of less area consuming hierarchical designs while synthesis time can be reduced.

* This work is partially supported by the DFG.

0-89791-993-9/97 $10.00 © 1997 IEEE

1.1 Related Work

A closer investigation of existing approaches shows, that the terms *hierarchical synthesis* and *complex components* are not used uniformly at algorithmic level. We can identify three different methodologies, which use the term hierarchical synthesis:

1. Data-flow graph clustering or partitioning methods followed by the synthesis of the clusters and partly of the clustered data-flow graph.
2. Using already synthesized systems as components, but without regard to internal component structures (“*black-box reuse*”).
3. Using already synthesized systems as components with the possibility of sharing subcomponents with regard to internal component structures (“*white-box reuse*”).

One of the first clustering methods involves collecting all operations with a high similarity measure into one cluster [2]. The allocation and assignment of functional units, registers, and multiplexers are determined separately for each cluster. The scheduling algorithm considers the whole data-flow graph and takes specific cluster restrictions into account. An extension to this approach can be found in APARTY (System Architect’s Workbench) [3], in which the user can choose a partitioning strategy like control, data, procedure-call, procedure-data and operation clustering. Another clustering approach extracts regular structures from the data-flow graph in order to create an additional level of hierarchy [4]. Then, the same scheduling algorithm is used to synthesize each cluster, and subsequently the clustered data-flow graph. The CATHEDRAL-III system [5] uses another clustering approach for synthesis. In addition to the other approaches, clusters with similar functional units are

merged after the clustering step in order to build complex data-path elements and to increase the cluster size [6]. A larger cluster size allows an enlarged design space exploration as opposed to the local decisions of other clustering techniques. A further data-flow graph partitioning technique proceeds scheduling in a bottom-up traversal of the loop/ subroutine hierarchy. One modern example is the *Synopsys Behavioral Compiler* [7].

The first methodology perform scheduling in a bottom-up traversal of the cluster or the given loop/subroutine hierarchy. The second mentioned approach for hierarchical synthesis allows the usage of already synthesized components for further high-level synthesis tasks without regarding specific component structures. The register-transfer library used in the Hebe system [8], which is integrated in the Olympus synthesis system, can contain any component that is specified in HardwareC. A similar approach is integrated in AMICAL [9], with the difference that a proprietary component intermediate format is used for synthesis. However, all previous mentioned approaches perform component reuse and resource sharing just at one hierarchical level of the design, without respect to its component structures ("black-box reuse").

Our approach presented in this paper offers a technique of hierarchical synthesis according to the methodology under 3 and supports "white-box reuse". Figure 1 illustrates the proposed hierarchical synthesis technique. The entire system is synthesized in a bottom-up traversal of the hierarchy. Each symbol represents a subdesign which is saved in the component library after synthesis. The component model of the library includes the VHDL behavioral description, the RT structure, and the calculated schedule. The enhanced component model is a prerequisite to perform subcomponent sharing of autonomous components. Both VHDL models are the basis for supporting the simulation of the entire system at algorithmic level as well as RT level.

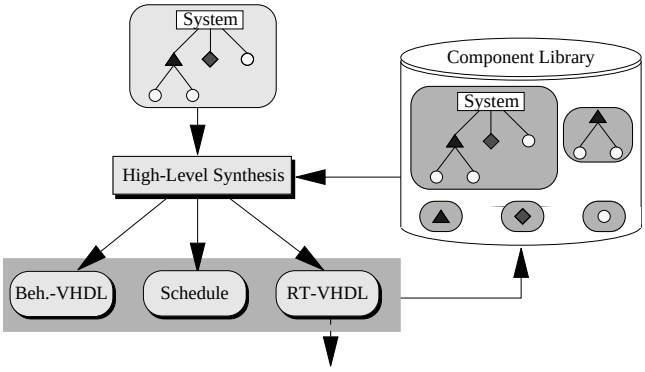


Figure 1. Hierarchical Synthesis with Component Model

Another requirement for hierarchical synthesis is a visible and changeable structure of complex components, which can be composed hierarchically from subcomponents. Furthermore, a behavioral simulation of the entire system, including the chosen register-transfer components, should be supported.

A closer investigation of the existing approaches shows that specific libraries developed for the respective high-level synthesis systems and technology-oriented libraries can be distinguished. Specific libraries are used in the high-level synthesis tools System Architect's Workbench (SAW) [3], Synopsys Behavioral Compiler [7], Hebe [8], and NEAT [10]. The libraries differ in the complexity of their components, but neither consideration of specific component structures nor behavioral simulation at algorithmic level, including the used register-transfer components, are supported. Only some recently introduced approaches consider an enhanced component model. OSCAR [11] and ISE [12] represent complex component as behavior templates in order to match multiple operations by a single component. Additionally, in [12] components may contain multiple functional outputs. In contrast to the other systems mentioned, CATHEDRAL-III [5] uses a constructive approach. Complex data-path elements are constructed from primitive operators, which are mapped to primitive library components, or to hardware building blocks of a module generator. Reusing complex components as primitive operators or complex data-path elements is not possible. A similar technique is used in the pre-synthesis system ACE [13]. Their component models are more abstract, but the system only provides some architectural transformations, like component merging, in order to increase the potential of resource sharing within components.

GENUS [14] is a generic, technology oriented register-transfer library used by the high-level synthesis system BdA, formerly VSS. GENUS automatically generates a component for an operation from elementary function units. But it is not possible to hierarchically combine elementary components to build complex components. Hence, several operations of a given behavioral specification can not be assigned to such complex components. Simulation models in VHDL can be generated for functional simulation as well as timing simulation.

Hierarchical synthesis in terms of reusing complete designs as components at a higher hierarchical level is either performed without regarding specific component structures or infeasible due to the lack of flexible component features. Therefore, the component concept must be extended and the synthesis must be enhanced, in order to handle complex components. Components should especially be able to be reconfigured or partially re-synthesized so that their subcomponents can be shared with other components. In contrast to black-box reuse with direct assignment to the corresponding DFG operation, the degree of optimization is increased.

This paper is organized as follows: Section 2 describes

the basic concepts of hierarchical synthesis. Section 3 addresses the hierarchical resource sharing problem of autonomous, parallel working components. Some examples, including experimental results, are presented in section 4. Finally, this paper concludes with a summary in section 5.

2 A Concept for Hierarchical Synthesis

In this section, we explain in more detail our hierarchical synthesis concept. First, the identification of complex components is described. Then, the most important task in hierarchical synthesis, the sharing of subcomponents across different levels of hierarchy, is outlined.

2.1 Identification of Complex Components

The first task that has to be solved during hierarchical synthesis is the identification of complex components in the control-data flow graph, distinguishing *direct component instantiation* and *component matching*. Direct component instantiation denotes a user specified component instantiation in the algorithmic specification. In this case, the user may preset the allocation of a complex component by invoking the corresponding procedure in the specification. The term component matching denotes the matching of component and system data-flow subgraphs, in order to identify suitable optimized complex components for the design. The basis of component matching is the component behavioral specification, which can easily be transferred into a control-data flow graph. Thus, the matching problem needs to be solved for both the data-flow subgraphs and the control-flow subgraphs of the specification.

Figure 2 illustrates both mentioned possibilities of complex component identification with three levels of hierarchy. In the first step, the mult-add subcomponent has been identified as a part of component f_3. Component f_3 has been instantiated directly by node f_3 of the overall system and can also be identified as a part of the overall system.

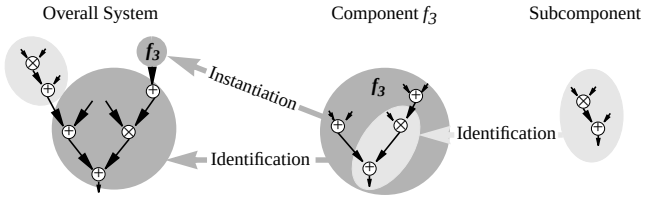


Figure 2. Identification of Complex Components

For the sake of clarity, this example illustrates only the data-flow graph matchings. The task of component identification is to be integrated in allocation as well as scheduling. Thereby, the synthesis system has to decide automatically whether specialized and optimized complex components or several primitive components are allocated. After scheduling, all DFG operations covered by one component are folded into a single complex operation node. Hence, no enhancements are needed in the further synthesis steps. Due to space limitations, this task is beyond the scope of this paper.

2.2 Hierarchical Resource Sharing

The most important task during hierarchical synthesis is the sharing of subcomponents across different hierarchical levels of the design. This aims at reusing an already synthesized design as a complex component at a higher hierarchical level with respect to the internal component structure and the already determined component schedule. The component schedule lists all used subcomponents per clock step. In the case of sharing subcomponents, the component structure is needed in order to estimate the additional required area and to extract the set of allocated subcomponents. All unused subcomponents at one clock step can now be added to the set of allocated and unused components of the overall system.

Figure 3 clarifies the sharing of subcomponents of the example given in figure 2, simplified to components without separate controllers. The starting point is the data-flow graph of the overall system and the schedule of the already synthesized component f_3. Assuming that component f_3 is used for the second part of the system data-flow graph (see figure 2), just one complex component f_3, is needed to cover the overall system. This is true, due to sharing the mult-add subcomponent of component f_3 with the remaining multiply and add operations of the overall system. Note that the allocation of one complex component f_3 implies the allocation of one mult-add and one add component, due to the visible structure of component f_3. The allocation of one mult-add component does not cause a resource conflict at clock step 3 of the system schedule because a pipelined component with an initiation interval of one clock step is assumed.

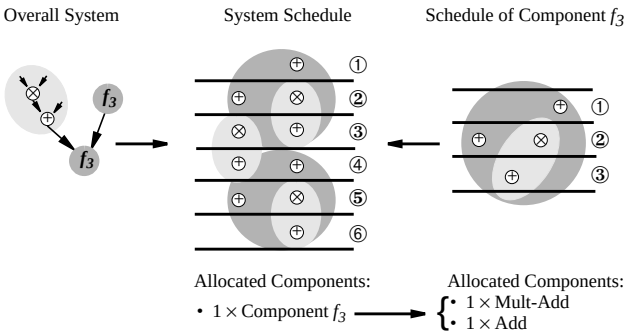


Figure 3. Hierarchical Resource Sharing

In the following, we want to extend the hierarchical resource sharing to autonomous, parallel working components with separate controllers.

3 Hierarchical Resource Sharing of Autonomous Components

Here we will concentrate on hierarchical resource sharing, applicable to autonomous, parallel working components. Especially, the additional needed calculations to perform subcomponent sharing of autonomous components are presented in this section.

3.1 Component Model

The underlying component model is not restricted to specific synthesis limitations. It consists of a flowgraph, representing the functional specification, a hierarchical schedule, the generated RT structure, and the physical component attributes.

Definition 1. A *component* of a design is denoted by the tuple $C := \langle FG, HS, FSMD, P \rangle$, where *FG* is an intermediate flowgraph format, compiled from an algorithmic specification, for instance written in VHDL, *HS* denotes the component schedule, which will be defined next, *FSMD* represents the synthesized RT structure as a finite state machine with datapath, and *P* describes the common physical component attributes, e.g. timing, area, and power consumption.

This data structure can be used for specifying both system under synthesis and instantiated components. In contrast to conventional component models, our model is much more detailed, in order to provide all necessary component information for hierarchical resource sharing. Next, the formal definition of a hierarchical schedule is given.

Definition 2. A *hierarchical schedule* of a component C is denoted by the tuple $HS(C) := \langle V, E, t, OP, IC, C \rangle$, where

- V is a set of nodes representing clock steps, conditional branches, or nested loops, respectively.
- $E \subset L \times V$ is a set of edges with $L := \{v_l \in V : t(v_l) = loop \vee t(v_l) = branch\}$, where
- the function $t(v_l) \in \{operation, loop, branch\}$ denotes the type of a node $v_l \in V$.
- The relation $OP(v_{op}), \forall (v_{op} \in V : t(v_{op}) = operation)$ refers to the scheduled operations of the clock step v_{op}.
- The function $IC(v_l), \forall (v_l \in V : t(v_l) = loop)$ returns the minimal iteration count ic_{min} and the maximal iteration count ic_{max}, where *iteration count* denotes the number of iterations of a loop, which may depends on outer loop iterators.
- The function $C(op), \forall op \in OP(v_{op})$ refers to the instantiated component or component type of an operation *op*.

The hierarchical schedule $HS(C)$ of a component represents a tree, with the property that only nodes of type *loop* or *branch* can have descendants. The children of one node represent the schedule of this loop, or branch path, respectively. The root node represents the entire process and has an infinite iteration count ($ic_{max} = \infty$). An unbounded iteration count, which can not be statically determined, is denoted by $ic_{max} = u$. In case of a statically determined iteration count, the formulas ic_{min} and ic_{max} are equal. In order to be more general and to avoid the distinction between system and components, the term module is used as a genus.

3.2 Sharing Interval

Resource sharing among different modules is only possible, if the state of the concerning modules can be determined statically. On the condition that some modules may be autonomous in the entire system and using a separate controller, the module state is determined, once the module under synthesis communicates with the other modules. The state of a module in which such a data transfer is initiated, is called *synchronization point* ($v_{sp} \in V$). The reset state is a priori a synchronization point. During list scheduling, a module state can be determined, as soon as the corresponding synchronization point is reached. Resource sharing is now possible until a loop, with an unbounded iteration count ($ic_{max} = u$) is reached and then only during the first ic_{min} iteration of this loop, such a state is called *desynchronization point*. Once the next synchronization point is reached, resource sharing is possible again. Note that one module can have multiple sharing intervals.

Definition 3. The *sharing interval* of a module *M* regarding state v_t is given by the set $SI(M, v_t)$ and denotes the set of states between a synchronization point and a desynchronization point, such that $v_t \in SI(M, v_t)$ holds.

The size of a sharing interval depends on the underlying module model. Modules without a separate controller (type *MT1*) are always statically determined (see figure 3). Hence, all further calculation in this section are not needed for this module type. Modules with a separate controller and data-independent timing (type *MT2*) permit the static calculation of their state. Only modules with a separate controller and unbounded data-dependent delay (type *MT3*) have a restricted sharing interval, which can be calculated as mentioned above. Figure 4 illustrates sharing intervals for the different module types. For the sake of clarity, the hierarchical schedule of modules with a separate controller is represented as a state transition graph of a FSM, and its datapath is not shown. In this case a sharing interval ending at the first transition of the FSM representation with unbounded iteration count. The states *R* and *W* represent read and write communication to other modules and are the synchronization points of the examples. Therefore, module type *MT3* of figure 4 contains two sharing intervals, the first beginning at state *R*, and the second beginning at state *W*.

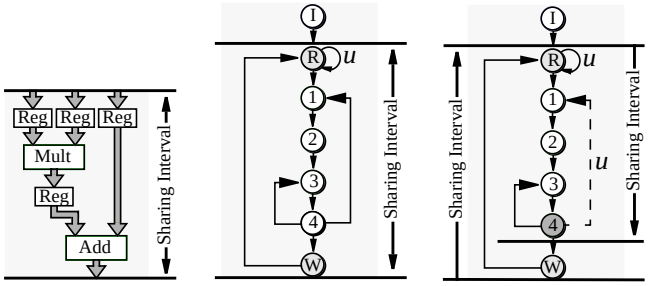


Figure 4. Sharing Intervals of Different Examples

3.3 Clock Cycle Space of a State

In the case of modules with separate controllers, all clock cycles of a module state, depending on their iteration space, have to be calculated, where the *iteration space* of a module is defined by the iteration counts of all loops. Note that loops with loop increment or loop decrement greater than one can be handled similarly, because only the iteration count has to be taken into account. Now, we will define the clock cycle space of a module state.

Definition 4. The *clock cycle space* $CCS(v_t)$ denotes the set of all clock cycles in which the considered module M is in the given state v_t, within the previous calculated sharing interval $SI(M, v_t)$.

The clock cycle space depends directly on its loop nesting. A single loop with ten iterations, for instance, implies that all states of the loop body are reached ten times.

Before the calculation of the clock cycle space is given, three helpful functions are defined. First, the number of clock cycles of a loop body v_l is given by $csteps_{LB}(v_l) = |\{v_c : v_c \in children(v_l) \wedge t(v_c) = operation\}|$, where $children(v_l)$ denotes the set of direct succesors of v_l. Secondly, the set of all subloops of a loop v_l is defined by $subloops(v_l) = \{v_s : v_s \in children(v_l) \wedge t(v_s) = loop\}$. Thirdly, the set $loopH(v_{sp}, v_t) = \{v_s : v_s \in descendant(v_{sp}) \wedge v_s \in ancestor(v_t) \wedge t(v_s) = loop\}$ specifies all subloops of the loop hierarchy $\langle v_{sp}, v_t \rangle$, where the sets $descendant(v_t)$ and $ancestor(v_t)$ represent the transitive closure to the direct successors or predecessors of v_t, respectively. Next, we calculate the *clock cycle space* of a given state v_t:

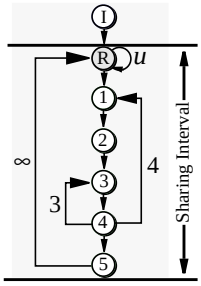
$$CCS(v_t) = csteps_{LB}(v_t) + \sum_{v_l \in loopH(v_{sp}, v_t)} csteps_{LH}(v_l) \cdot x_{v_l},$$

subject to $0 \le x_{v_l} < ic_{max}(v_l), \forall\, v_l \in loopH(v_{sp}, v_t),$

where

$$csteps_{LH}(v_l) = csteps_{LB}(v_l) + \sum_{v_s \in subloops(v_l)} ic_{max}(v_s) \cdot csteps_{LH}(v_s).$$

The function $csteps_{LH}(v_l)$ denotes the number of clock cycles of a given loop hierarchy v_l. Figure 5 illustrates the calculation of the clock cycle space with a simple example. The edge labels denote the iteration count of the corresponding loop. The outermost loop represents the entire process. Note that the innermost loop is specified with an iteration count of three. The clock cycle space is shown for the states 1, 2, 3, 4, and 5.



$CCS(\{1,2\}) := 8 \cdot x_1 + \{1,2\};$ where $0 \le x_1 < 4$

$CCS(\{3,4\}) := 8 \cdot x_1 + 2 \cdot x_2 + \{3,4\};$ where $0 \le x_1 < 4,\ 0 \le x_2 < 3$

$CCS(1) := \{1, 9, 17, 25\}$
$CCS(2) := \{2, 10, 18, 26\}$
$CCS(3) := \{3, 5, 7, 11, 13, 15, 19, 21, 23, 27, 29, 31\}$
$CCS(4) := \{4, 6, 8, 12, 14, 16, 20, 22, 24, 28, 30, 32\}$
$CCS(5) := \{33\}$

Figure 5. Clock Cycle Space $CCS(R,v_t)$ of an Example

3.4 Collision Set

The clock cycles spaces of all modules containing shareable submodules have to be calculated and examined, in order to determine whether submodules can be shared of two or more modules.

Definition 5. Assuming that module M_1 is in state v_t and a submodule s_2 contained in module M_2 should also be used *in* M_1, then the *collision set* $COL(M_1, v_t, M_2, s_2)$ is given by

$$CCS_{M_1}(v_t) \cap \left(\bigcup_{\substack{v_l \in SI(M_2, v_t) \cap \\ used(M_2, s_2)}} CCS_{M_2}(v_l) \right), \text{ where}$$

$used(M_2, s_2)$ denotes all states of module M_2, in which submodule s_2 is in use, within the given sharing interval $SI(M_2, v_t)$, so that $SI(M_1, v_t) \subseteq SI(M_2, v_t)$ holds.

The collision set describes, whether resource conflicts arises by applying resource sharing across different levels of hierarchy of autonomous modules.

Definition 6. A collision set $COL(M_1, v_t, M_2, s_2)$ is called *collision-free*, if and only if $COL(M_1, v_t, M_2, s_2) = \emptyset$ holds.

Generally, a submodule s_j of module M_j can be used in state v_t of module M_i, or can be shared with module M_i, if $COL(M_i, v_t, M_j, s_j)$ is collision-free. In this case, the clock cycle space of M_i covers the clock cycle spaces of M_j in which the submodule s_j is not being used. This is the complementary set of the above defined union of used submodules. It should be noted, that checking, whether a module contains submodules which are able to perform a chosen operation $op \in OP(v_t)$ should be done before calculating the collision set. This is the remaining step of conventional resource sharing techniques.

Figure 6 shows the collision set of state 3 of the module M_1 with an assumed demand of one adder. The module M_2

contains one allocated add component, which is used in the state set {2, 4, 5}. Because the determined collision set is collision-free, the module M_1 may use the add submodule of the module M_2.

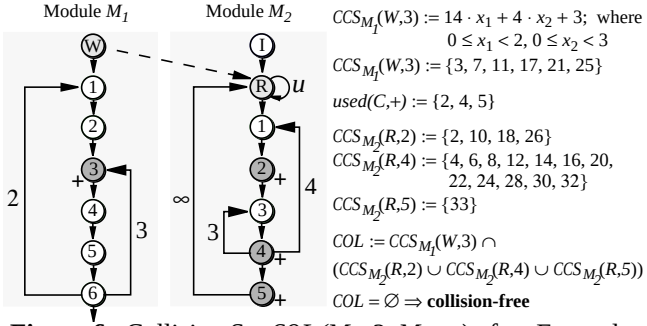


Figure 6. Collision Set $COL(M_1, 3, M_2, +)$ of an Example

3.5 Collision Set Intersection Problem

For the case of loops with a small iteration count, the collision set can be determined by simple set operations. However, this results in an exhaustive enumeration, with an intractable complexity. For example, if two nested loops have been specified, both with an iteration count of 1000, the corresponding clock cycle space consists of one million members. Therefore, we will use the following integer problem solving technique.

The collision set $COL(M_i, v_t, M_j, s_j)$ can be decomposed into $|SI(M_j, v_t) \cap used(M_j, s_j)|$ subproblems which can be solved independently as follows:

$$CCS_{M_i}(v_t) \cap CCS_{M_j}(v_j),\ \forall\ v_j \in SI(M_j, v_t) \cap used(M_j, s_j).$$

Now we are able to formulate the *collision set intersection problem*.

Definition 7. The *collision set intersection problem* is defined by $CCS_{M_i}(v_t) - CCS_{M_j}(v_j) = 0,\ \forall\ v_j \in SI(M_j, v_t) \cap used(M_j, s_j)$, subject to $0 \le x_{v_l} < ic_{max}(v_l), \forall\ v_l \in loopH(v_{sp}, v_t)$. Due to the linearity of CCS, the equation has the structure:

$$(A_{M_i} - A_{M_j}) \cdot \boldsymbol{x} = b_{M_i} - b_{M_j},$$

subject to $\boldsymbol{x} < ic_{max}(v_l), \boldsymbol{x} \in \mathbb{N}^d$, *with* $d = |loopH(v_{sp}, v_t)|$.

Now, we are interested in the question, whether an integer solution of the equation exists, which satisfies the constraints $0 \le \boldsymbol{x} < ic_{max}(v_l)$. This problem can be solved in four steps. First, an unimodular matrix U and an echelon matrix D are to be found, such that $A \cdot U = D$ holds [16], where $A = A_{M_i} - A_{M_j}$. The matrix U describes the matrix transformations being applied to generate integer solutions. Second, the existence of an integer solution of the unrestricted equation can be determined by applying the gcd test to $D \cdot \boldsymbol{s} = b$, where $b = b_{M_i} - b_{M_j}$ [16]. Thirdly, if the gcd test is successful, the equation $\boldsymbol{x} = U \cdot \boldsymbol{s}$ describes all integer solutions and can be inserted in the constraint system: $0 \le U \cdot \boldsymbol{s} < ic_{max}(v_l)$, $\boldsymbol{x} \in \mathbb{N}^d$. Finally, the existence of a feasible integer solution in the remaining polyhedron can be tested by applying the extended integer Fourier-Motzkin projection [15], [17], which is recommendable due to the simple constraint system. Consequently, if no integer solution is found, the collision set CCS is collision-free and thus, resource sharing is possible.

In general, the extended Fourier-Motzkin projection has an exponential worst-case complexity depending on the number of variables. Nevertheless, the computation time for synthesis can be kept low, because the number of variables is limited by the maximum levels of nested loops. For real problems, the nesting loop level is smaller than five. Furthermore, the algorithm can stop once a collision is detected. Most of the collisions can already be detected by the very efficient gcd test. Due to the very simple constraint system, the average complexity of the algorithm can be further reduced. Their influence on the entire synthesis time is less than one percent in all tested cases.

3.6 Conditional Branches

If the specification contains conditional branches ($\exists\ v_l \in V : t(v_l) = branch$), occurring when using *if* or *case* constructs, the calculation of the collision set must be extended. In this context, two cases have to be distinguished. First, if the branch condition depends on outer loops and can be predicted, then disjunctive clock cycle spaces has to be calculated independently for each alternative path. Accordingly, the collision set intersection problem has to be solved for all disjunctive clock cycle spaces. Second, in case of data dependent unbounded conditions, a submodule may be shared in a given state, only if the submodule are unused in all alternative paths. Hence, the set of used submodules, needed to determine the collision set COL, is calculated by the union of the *used* sets of all alternative paths.

4 Implementation into CADDY-II

This section outlines the integration of hierarchical resource sharing into the high-level synthesis system CADDY-II. The underlying synthesis algorithms are only summarized here. A more detailed description can be found in [18], [19], [20] and [21]. The main synthesis steps performed in the CADDY-II system are *data-flow analysis*, *allocation*, *scheduling*, *binding*, *data-path generation* and *controller generation*. During allocation a suitable component set is selected. Tasks of scheduling are the assignment of operations to clock steps and to component types under resource constraints. The mapping of operations to component instances and the allocation of registers are the tasks of the binding. The presented hierarchical resource sharing approach has been integrated within scheduling and binding.

List-scheduling is used as scheduling algorithm, driven

by a global estimation function, which is based on the probabilities of scheduling DFG operations to control steps. In each step of the list-scheduling algorithm, all ready operations are mapped to unused components, which are able to execute the corresponding operation. If an already allocated autonomous module contains a subcomponent, which can perform an operation out of the ready set, the collision set with respect to the currently scheduled state is calculated. Since the current loop is not necessarily completely scheduled, the remaining delay is expressed by additional parameters, therefore the collision set has to be calculated with respect to this parameters. The determined parametric solutions are used as scheduling constraints of the outer loops. During scheduling of the outer loops the algorithm decides, whether the constraints for sharing subcomponents of autonomous modules can be fulfilled, suitably. During binding, the collision set is calculated again, but with fully determined variables in order to perform the mapping of an operation to a specific component or subcomponent instance.

5 Experimental Results

Experimental results of our approach on several designs, including some benchmark circuits, are given in this section. First, just for reasons of comparability the results for the fifth order elliptic wave filter benchmark are given. This benchmark demonstrates two essential features of our hierarchical synthesis approach: First, recognition of complex component structures in the system data-flow graph, and second, the possibility of sharing subcomponents. Table 1 shows some results for different allocations with and without subcomponent sharing, and compares this with the traditional component model. In this table, a component with a data initiation interval of one clock cycle and an execution time of two clock cycles is specified by the notation '1 : 2', for instance.

Table 1. 5th Order Elliptical Wave Filter

with complex components						without complex components			
resources			clock steps (cs)			resources			clock steps (cs)
+	MAC	MAC	with sharing	without sharing	gain	+	*	*	
1 : 1	1 : 2	1 : 3				1 : 1	1 : 1	1 : 2	
1	1	0	16	20	4	2	1	0	16
2	1	0	15	15	1	3	1	0	15
1	2	0	14	19	5	3	2	0	14
1	0	1	18	21	3	2	0	1	19
1	0	2	17	21	4	3	0	2	17

As a result, we get the performance improvements listed in column entitled "gain". In this example the, speed-up of the design is up to 5 clock cycles with equal hardware costs. This is because the multiply-accumulate component may share the internal adder, if an additional adder is needed. In contrast, if subcomponent sharing is not supported, a resource conflict can only be solved by adding a further clock step. The CPU time for the filter on a Sun SPARC 20 was less than 2 seconds for a fixed set of allocated components and less than 12 seconds for an enlarged design space exploration by synthesizing different sets of automatically allocated components.

Second, we will present the results of the FDCT benchmark, shown in table 2. The given component costs are taken from [11] and amount 10 units for using an adder, 20 units for using an multiplier, and 25 units for using a multiply-accumulate unit. The column entitled "costs" is filled with the area-time product as cost function. Applying subcomponent sharing, a speedup of up to 8 clock cycles can be achieved. In comparison to non-encapsulated components, the performance results of the synthesized circuits are equal in most of the determined cases, while the area costs can be reduced. Particularly, the optimal circuit in relation to the area delay ratio is synthesized using two multiply-accumulate units and two adders. The CPU time for the FDCT benchmark was less than 4 seconds for a fixed set of allocated components and less than 16 seconds for an enlarged design space exploration.

Table 2. Fast Discrete Cosine Transformation

with complex components							without complex components			
resources				clock steps (cs)			resources			clock steps (cs)
+ / –	*	MAC		with sharing	without sharing	gain	+ / –	*		
10	20	25	costs				10	20	costs	
1	0	3	765	9	14	5	4	3	900	9
1	0	2	720	12	18	6	3	2	770	11
1	1	2	990	11	19	8	3	3	810	9
2	0	2	700	11	16	5	4	2	880	11
2	1	2	900	10	13	3	4	3	900	9
2	0	3	855	9	13	4	5	3	990	9
2	0	1	810	18	20	2	3	1	900	18
1	1	1	715	13	21	8	2	2	780	13
2	2	1	850	10	13	3	3	3	810	9
2	1	1	715	11	13	2	3	2	770	11

Finally, the results of the simulated annealing processor taken from [22] are presented. At first, the needed floating-point components and then the overall simulated annealing algorithm have been specified. The high-level synthesis system CADDY-II maps all floating-point operations to the previous designed components and synthesizes the entire system hierarchically with respect to the used floating-point components. All instantiated subcomponents of the floating-point components are now ready to be used as shared components within the entire simulated annealing design. The floating-point multiplier for instance, consists of an integer multiplier, an integer adder, and a barrel shifter. These components can be used additionally for other integer arithmetic operations of the whole design. As a result, all specified arithmetic operations could be covered by the subcomponents of the floating-point units. Table 3 lists the synthe-

sized results for a hierarchical and a inline-expanded description. By using encapsulated components, the controller size of the simulated annealing processor could be reduced to 46% of the controller size of the inline-expanded description by an equal performance. The CLB count is related to the Xilinx XC4000 family and is given to get more detailed information. The CPU time could be reduced from 20 to 7 seconds when using the hierarchical description.

Table 3. Simulated Annealing Processor

Components		Hierarchical Description	Inline-Expanded Description	Area Reduction
Datapath:	Multiplier	1	1	0 %
	Add/Sub	2	2	0 %
	Barrelshifter	1	1	0 %
	Leading_Zero	1	1	0 %
	Comparator	2	3	33 %
	Multiplexer	19	27	30 %
	Register	11	12	8 %
Controller:	Gate Equivalents	158	307	49 %
	CLBs	56	104	46 %
	States	11	26	58 %

In summary, the presented results show several advantages of the hierarchical synthesis method regarding complex component structures. When using subcomponent sharing, the synthesized circuits need less clock cycles under resource constraints, and are less area consuming under timing constraints. Furthermore, the run-time of the synthesis algorithm could be decreased compared to non-hierarchical approaches.

6 Summary and Conclusion

This paper presented a new approach for hierarchical high-level synthesis regarding complex component structures. The presented experimental results encourage further investigations in this area. The advantages of the presented approach are:

- The concept of complex components offers the basis for a hierarchical synthesis methodology with respect to specific component structures, in order to increase the degree of optimization.
- Resource sharing can be performed across different levels of hierarchy of parallel working components, with a separate controller.
- The modification of the components can be done after binding, such that the synthesis time can be kept low.
- Each synthesized submodule can be reused in the same design as a complex register-transfer component.
- Multiple instances of one component have to be synthesized only once.

7 References

[1] R. Bergamaschi: *Productivity Issues in High-Level Design: Are Tools Solving the Real Problems?.* Proceedings of DAC, 1995.

[2] M. McFarland, T. Kowalski: *Incorporating Bottom-Up Design into Hardware Synthesis.* IEEE Transactions on CAD, vol. 9, pp. 938-950, 1990.

[3] D. E. Thomas, E. D. Lagnese, R. A. Walker, J. A. Nestor, J. V. Rajan, R. L. Blackburn: *Algorithmic and Register-Transfer Synthesis: The System Architect's Workbench.* Kluwer, 1990.

[4] D. Sreenivasa Rao, F. Kurdahi: *Hierarchical Design Space Exploration for a Class of Digital Systems.* IEEE Transactions on CAD, vol. 1, pp. 282-295, 1993.

[5] W. Geurts, F. Catthoor, H. De Man: *Quadratic Zero-One Programming-Based Synthesis of Application-Specific Data Paths.* IEEE Transactions on CAD, vol. 14, pp. 1-11, 1995.

[6] W. Geurts, F. Catthoor, H. De Man: *Time Constrained Allocation and Assignment Techniques for High Throughput Signal Processing.* Proceedings of DAC, 1992.

[7] T. Ly, D. Knapp, R. Miller, D. MacMillen: *Scheduling using Behavioral Templates.* Proceedings of DAC, 1995.

[8] D. C. Ku, G. De Micheli: *High-Level Synthesis of ASICs Under Timing and Synchronization Constraints*. Kluwer, 1992.

[9] P. Kission, H. Ding, A. Jerraya: *VHDL Based Design Methodology for Hierarchy and Component Re-Use.* Proceedings of EURO-VHDL, 1995.

[10] A. Timmer, M. Heijligers, L. Stok, J. Jess: *Module Selection and Scheduling using Unrestricted Libraries.* Proceedings of EDAC, 1993.

[11] B. Landwehr, P. Marwedel, R. Dömer: *OSCAR: Optimum Simultaneous Scheduling, Allocation and Resource Binding Based on Integer Programming*, Proceedings of EURO-VHDL, 1994.

[12] R. Ang: *Library Insertion and Reuse of Datapath Components in High-Level Synthesis.* PhD Thesis, University of California, Irvine, 1996.

[13] B. G. Hald, J. Madsen: *A Flexible Representation for High-Level Synthesis.* Proceedings of 2nd Asian Pacific Conference on Hardware Description Languages, 1994.

[14] P. Jha, N. Dutt: *Design Reuse through High-Level Library Mapping.* Proceedings of European Design & Test Conference, 1995.

[15] M. Wolfe: *High Performance Compilers for Parallel Computing.* Addison-Wesley, 1996.

[16] U. Banerjee: *Loop Transformations for Restructuring Compilers: The Foundations.* Kluwer, 1993.

[17] William Pugh: *The Omega Test: a fast and practical integer programming algorithm for dependence analysis.* Proceedings of Supercomputing, 1991.

[18] W. Rosenstiel, H. Krämer: *Scheduling and Assignment in High-Level Synthesis.* In R. Camposano, W. Wolf: High-Level VLSI Synthesis, Kluwer, 1991.

[19] P. Gutberlet, J. Müller, H. Krämer, W. Rosenstiel: *Automatic Module Allocation in High-Level Synthesis.* Proceedings of EURO-DAC, 1992.

[20] P. Gutberlet, H. Krämer, W. Rosenstiel: *CASCH - a Scheduling Algorithm for High-Level Synthesis.* Proceedings of EDAC, 1991.

[21] P. Gutberlet, W. Rosenstiel: *Timing Preserving Interface Transformations for the Synthesis of Behavioral VHDL*. Proceedings of EURO-VHDL, 1994.

[22] B. Eschermann, O. Haberl, O. Bringmann, O. Seitz: COSIMA: *A Self-Testable Simulated Annealing Processor for Universal Cost Functions.* Proceedings of EuroASIC, 1992.