System-Level Synthesis Using Evolutionary Algorithms

Tobias Blickle Jürgen Teich
Lothar Thiele
Computer Engineering and Communication Networks Lab (TIK)
Swiss Federal Institute of Technology (ETH)
Gloriastrasse 35, 8092 Zurich
Switzerland

TIK-Report
Nr. 16, April 1996

Contents

1 Introduction **1**
- 1.1 Optimization Methodology . . . 1
- 1.2 Existing Approaches to System-Level Synthesis . . . 2
- 1.3 Overview . . . 4

2 Modeling Algorithms and Architectures **7**
- 2.1 Specification . . . 7
- 2.2 System Synthesis . . . 11
- 2.3 The Task of System Synthesis . . . 14
- 2.4 Examples of Model Refinement . . . 15
  - 2.4.1 Model for Resource Sharing . . . 15
  - 2.4.2 Model for Chip Boundaries . . . 17

3 Optimization Method **20**
- 3.1 Principle of Evolutionary Algorithms . . . 20
- 3.2 Coding of Implementations . . . 21
  - 3.2.1 The Function node_allocation() . . . 23
  - 3.2.2 The Function binding() . . . 25
  - 3.2.3 The Function update_allocation() . . . 28
- 3.3 Scheduling . . . 28
- 3.4 Cost Evaluation: The Choice of the Fitness Function . . . 29
- 3.5 Parameters of the Evolutionary Algorithm . . . 30

4 Case Studies **31**
- 4.1 Video Codec . . . 31
  - 4.1.1 Problem Specification . . . 31
  - 4.1.2 Results . . . 33
- 4.2 PDE-Integrator . . . 35
  - 4.2.1 Problem Specification . . . 36
  - 4.2.2 Results . . . 36

5 Summary and Conclusions **39**

A Proof of Theorem 2.2.1 **41**

Abstract

In this paper, we consider *system- level synthesis* as the problem of optimally mapping an algorithm-level specification onto a heterogeneous hardware/software architecture. This problem requires (1) the *selection of the architecture* (allocation) including general purpose and dedicated processors, ASICs, buses and memories, (2) the *mapping* of the algorithm onto the selected architecture in space (binding) and time (scheduling) and (3) the design space exploration with the goal to find a set of implementations that satisfy a number of constraints on cost and performance. Existing methodologies often consider a fixed architecture, perform the binding only, do not reflect the tight interdependency between binding and scheduling, require long run-times preventing design space exploration or yield only one implementation with optimal cost. Here, a model is introduced that handles all mentioned requirements and allows the task of system-synthesis to be specified as an optimization problem. Furthermore, the application of an Evolutionary Algorithm to solve this problem is described.

Chapter 1

Introduction

The enormous progress in VLSI and CAD technology to support automated logic and high level synthesis throughout this decade has helped design engineers to shorten the time-to-market of new products drastically. As a consequence, more complex designs can be developed in shorter time. The time saved may be used to investigate different implementations using automated synthesis tools what is frequently called *design space exploration*. Going hand in hand, one can recognize a shift in the interest in the realm of CAD research to climb one level higher in the abstraction hierarchy by investigating the automated synthesis of designs starting at the *system-level*. Although there is no common definition of what system-level synthesis means, the following characterization might fit: System-level synthesis can be described as a mapping from a behavioral description where the functional objects possess the granularity of algorithms, tasks, procedures, or processes onto a structural specification with structural objects being general or special purpose processors, ASICs, buses and memories.

In the search of a sophisticated design methodology, there are many tradeoffs which have to be examined, and the high mutual dependency of different design criteria as well as the uncertainty of the estimated quality of underlying synthesis tools make it hard to explore the design space automatically.

1.1 Optimization Methodology

The proposed optimization methodology treats the problem of optimizing the mapping of an algorithm-level dataflow graph based specification onto a heterogeneous hardware/software architecture. This problem requires (1) the *selection of the architecture* (allocation) among a specified set of possible architectures, (2) the *mapping* of the algorithm onto a selected architecture in space (binding) and time (scheduling), and (3) the design space exploration with the goal to find a set of implementations that satisfy a number of constraints on cost and performance.

The novelty of our approach consists in

- *providing a new system-level algorithm/architecture model* for heterogeneous hardware/software systems in which the underlying architecture is not fixed a priori but a set of architectures that should be investigated can be specified using a graph-theoretic framework,

- *formulating a new formal definition for system-level synthesis* including steps (1)-(3),

- *applying Evolutionary Algorithms to system-level synthesis* and showing that they can perform steps (1)-(3) in a single optimization run.

Furthermore, we show that Evolutionary Algorithms are a good candidate for system-level synthesis because they a) iteratively improve a *population* (set) of implementations, b) they do not require the quality (cost) function to be linear (e.g., area-time product), c) they do not suffer from long run-times if the quality of an implementation (*fitness function*) can be computed efficiently, and d) they are known to work "well" on problems with large and non-convex search spaces.

Below, we give a short summary of existing approaches to system-level synthesis.

1.2 Existing Approaches to System-Level Synthesis

There exist already many different approaches to system-level synthesis that may be classified according to their class of input specifications:

- *Control-dominant specification:* Methods in this class consider control-dominated specifications, e.g., C-code [Hardt and Camposano, 1995] and extensions thereof [Ernst *et al.*, 1994] or finite-state-machine based specifications, e.g., [Ismail *et al.*, 1994]. In the mapping phase, (static) scheduling cannot be done due to non-determinism of execution times. Here, estimation mainly depends on profiling and simulation.

- *Dataflow-dominant specification:* Methodologies in this class consider dataflow specifications, e.g. dataflow graphs, e.g. [Kalavade and Lee, 1994], precedence- and taskgraphs or parallel languages, e.g. UNITY [Barros and Rosenstiel, 1992]. The approach in [Gupta and Micheli, 1992] belongs to the same category, however, with extensions to allow for operations with non-deterministic delays.

Tightly coupled with the class of input specifications is the scope of *target architectures*:

- *Dedicated control & data path in VLSI:* [McFarland, 1986], [Lagnese and Thomas, 1991], [Vahid and Gajski, 1992] are approaches to partition a functional specification for high-level synthesis. The target architecture is in most cases a dedicated hardware architecture including a control path and a data path.

- *Multi-chip dedicated VLSI architecture:* Some methodologies do focus on multichip VLSI solutions, e.g., [Küçükçakar and Parker, 1995]. All functionality is mapped onto a dedicated multi-chip architecture including buses.

- *Hardware/Software Architectures:* In the realm of hardware/software architectures, most approaches consider the target architecture to be fixed, e.g., [Kalavade and Lee, 1994] (one processor and custom hardware communicating via memory-mapped I/O), [Ernst *et al.*, 1994] (one RISC processor and one or more given custom blocks and predefined HW-modules that communicate mutually exclusive by memory coupling using a single CSP type protocol), or [Gupta and Micheli, 1992] (one programmable component and multiple hardware modules communicating with each other using one system bus, the processor being the master).

Finally, different approaches can be classified according to their optimization model and procedure: Most of these mentioned methodologies consider system-level synthesis as a *partitioning* problem. Partitioning techniques have been applied for many different problems in hardware synthesis, e.g. [Camposano and Brayton, 1987], [McFarland, 1986], [Lagnese and Thomas, 1991], [Vahid and Gajski, 1992]. There, the goal is in most cases to meet chip capacity and time constraints. Most of these approaches present a clustering-based approach. [Küçükçakar and Parker, 1995] considers multichip modules, however no programmable components are allowed. In some clustering based approaches like e.g., [Barros and Rosenstiel, 1992], only the space mapping (binding) is considered.

Finally, optimization methods can be classified into

- *exact methods:* These methods include enumerative search techniques and approaches based on integer linear programming. These approaches suffer from long run-times and are only worth investigating if the estimation of costs and performance can be proven to be highly accurate, and the number of objects is rather small.

- *heuristics:* Gupta in [Gupta and Micheli, 1992] present a heuristic for hardware/software partitioning where all functionality is initially mapped to hardware with the goal to move as much functionality as possible into the processor such that performance constraints remain guaranteed. Henkel et al. [Ernst *et al.*, 1994] present an approach that starts with an initial partition in software and moves functionality to hardware until timing constraints are met. They use simulated annealing in an outer optimization loop and, adapt estimated parameters in an inner loop.

1.3 Overview

Figure 1.1 gives an overview of our optimization methodology.

The specification consists of a dataflow graph based problem graph (see Fig. 1.1a), an architectural model (see Fig. 1.1c), user-defined mapping constraints (see Fig. 1.1b) and an optimization procedure (see Fig. 1.1d) using an evolutionary algorithm. An evolutionary algorithm works on *populations of individuals* J_i, $i = 1, \cdots, N$ where N is called *size of the population*, and each individual codes an implementation of the problem graph including an architecture and a mapping of nodes in the problem graph in space (binding) to that architecture. The evolutionary algorithm consists of an optimization loop (see Fig. 1.1d) that applies the principles of *reproduction*, *crossover* and *mutation* to the strings that code implementations. The purpose is to iteratively find better populations: Each individual in an actual population P_k is ranked by evaluation of a *fitness function* that gives a measure how good an implementation is in terms of cost and performance, etc. The evolutionary algorithm terminates after a certain number k_{max} of generated populations and outputs those implementations with the best fitness values.

Example 1.3.1 *Fig. 1.1e shows some individuals out of a population P_k of implementations. They correspond to the problem graph in Fig. 1.1a including the binding of functional nodes to functional resources (RISC, Hardware Modules 1 and 2) and the binding of communication nodes (shaded nodes) to bus resources (shard bus, point-to-point bus). The scheduling of nodes inside the functional resources is not shown. Note that each individual of a population may code a different architecture.*

The complete procedure implicitly performs an *exploration of the design space*: it is likely that in the end, several optimal implementations are obtained with possibly completely different architectural features. As for scheduling good heuristics are available, the individuals code allocation and binding only. The scheduling is considered to be part of the fitness evaluation of an individual.

First, the new specification model including the problem graph and the architecture graph will be formally defined. It is explained how to model costs, delays, bus- and pin- constraints, how to model single-chip systems, multiple chip solutions, communication, and resource sharing (Section 2). The tasks of the Evolutionary Algorithm including the definition of good fitness functions for evaluating the quality of an implementation are outlined in section 3. The

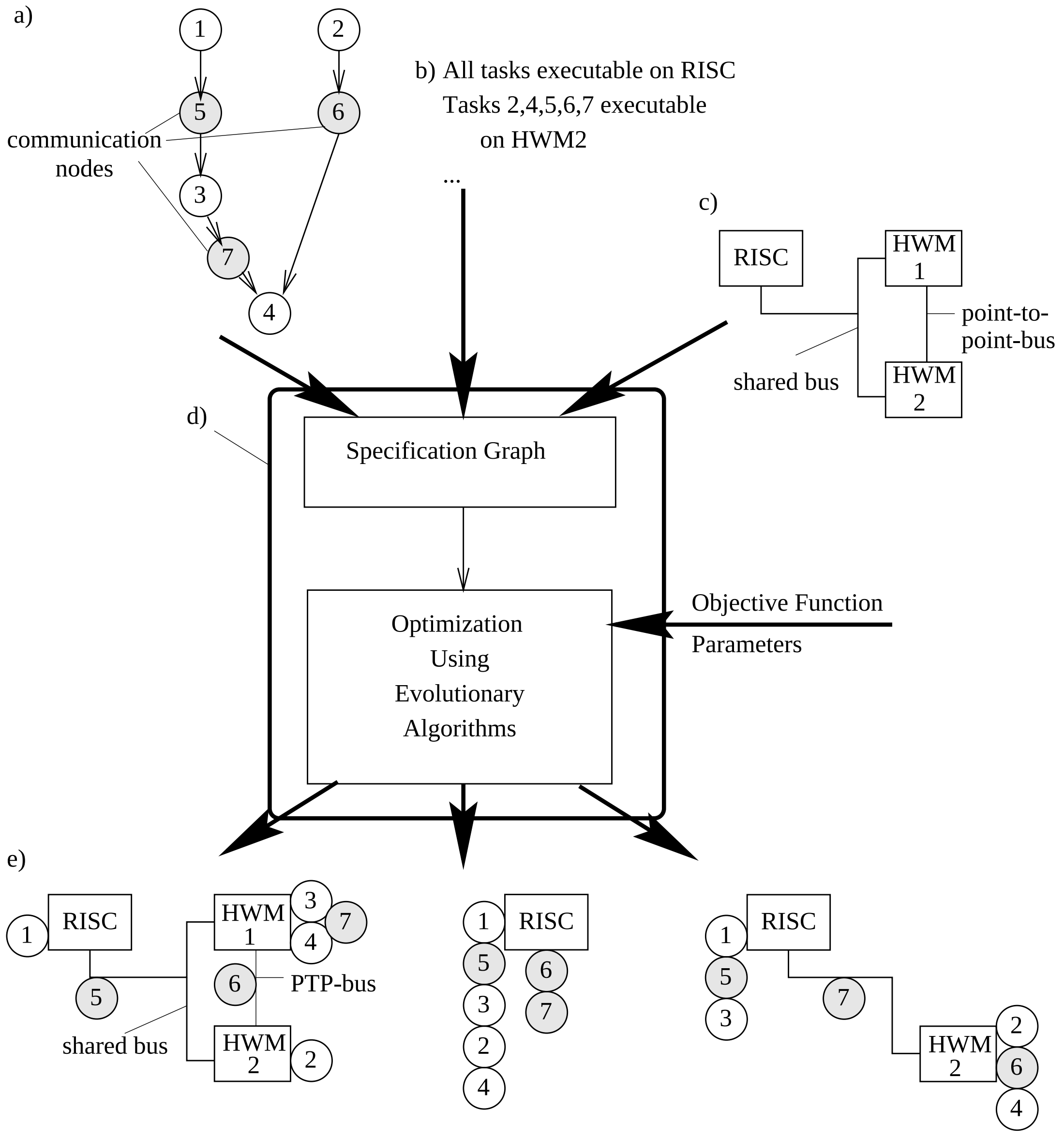

Figure 1.1: Overview of the proposed optimization methodology: a) dataflow based algorithm specification, b) mapping constraints, c) architecture template, d) optimization methodology, e) implementations

example of a video codec is used in section 4 as a case study to explain our optimization methodology and to show its performance. Also, a benchmark taken from high-level synthesis is used as an example to demonstrate the applicability of the methodology to high-level synthesis as a special case. This example also shows that several different optimal implementations can be obtained during one optimization run (design space exploration). Finally, some conclusions for further research are given in section 5.

Chapter 2

Modeling Algorithms and Architectures

2.1 Specification

The specification model consists of three main components:

- The algorithm that should be mapped on an architecture as well as the class of possible architectures are described by means of a universal *dependence graph* $G(V, E)$.
- The user-defined mapping constraints between algorithms and architectures are specified in a specification graph $G_S(V_S, E_S)$. Additional parameters which are used for formulating the objective functions and further functional constraints are associated to the components of G_S.
- Associated to nodes and edges of the specification graph are activations which characterize the allocation and binding.

At first, the (well known) concept of a dependence graph is used to describe algorithms as well as architectures on different levels of abstraction.

Definition 2.1.1 (Dependence Graph) *A* dependence graph *is a directed graph $G(V, E)$. V is a finite set of nodes and $E \subseteq (V \times V)$ is a set of edges.*

For example, the dependence graph to model the data flow dependencies of a given algorithm will be termed *problem graph* $G_P = (V_P, E_P)$. Here, V_P contains nodes which model either functional operations or communication operations. The edges in E_P model dependence relations, i.e. define a partial ordering among the operations.

Example 2.1.1 *One can think of a problem graph as the graph obtained from a data flow graph of an algorithm by inserting communication nodes into some*

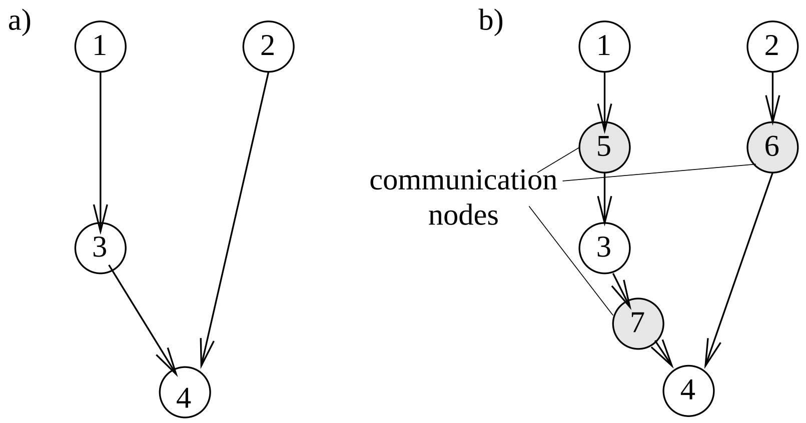

Figure 2.1: A data flow graph (a) and the corresponding problem graph (b)

edges of the data flow graph (see Fig. 2.1). These nodes will be drawn shaded in the examples.

Now, the architecture including functional resources and buses can also be modeled by a dependence graph termed *architecture graph* $G_A = (V_A, E_A)$. V_A may consist of two subsets containing functional resources (hardware units like adder, multiplier, RISC processor, dedicated processor, ASIC) and communication resources (resources that handle the communication like shared busses or point-to-point connections. An edge $e \in E_A$ models a directed link between resources. All the resources are viewed as *potentially allocatable* components.

Example 2.1.2 *Fig. 2.2a) shows an example of an architecture consisting of three functional resources (RISC, hardware modules HWM1 and HWM2) and two bus resources (one shared bus and one unidirectional point-to-point bus). Fig. 2.2b) shows the corresponding architecture graph G_A.*

In addition, in a next level of abstraction one may define a dependence graph termed *chip graph* $G_C(V_C, E_C)$ whose nodes correspond to integrated circuits and off-chip communication resources.

Example 2.1.3 *Fig. 2.3a) shows an example of a multi-chip architecture consisting of two integrated circuits* chip1 *and* chip2 *and a bi-directional point-to-point bus resource. Fig. 2.3b) shows the corresponding chip graph G_C.*

Note that the above used problem, architecture and chip graph are examples only. Next, it is shown how user-defined mapping constraints can be specified in a graph based model. Moreover, the *specification graph* will also be used to define *binding* and *allocation* formally.

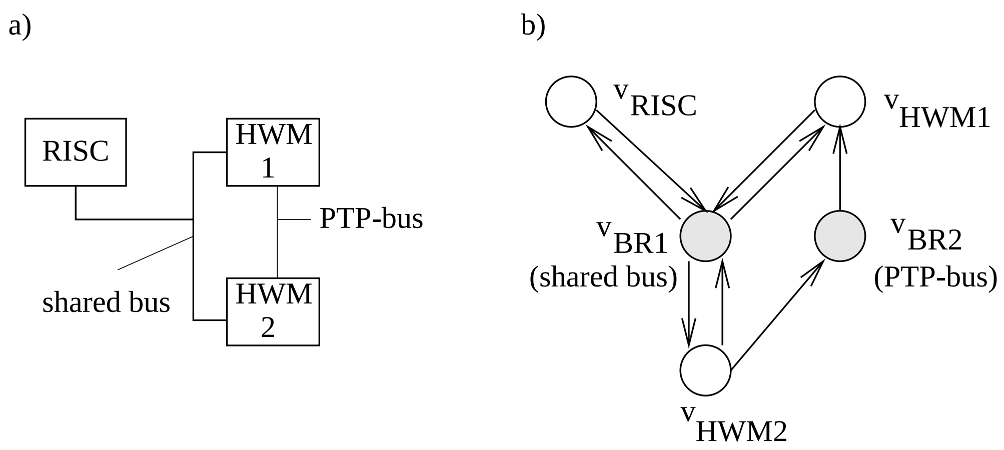

Figure 2.2: An example of an architecture (a), and the corresponding architecture graph G_A (b)

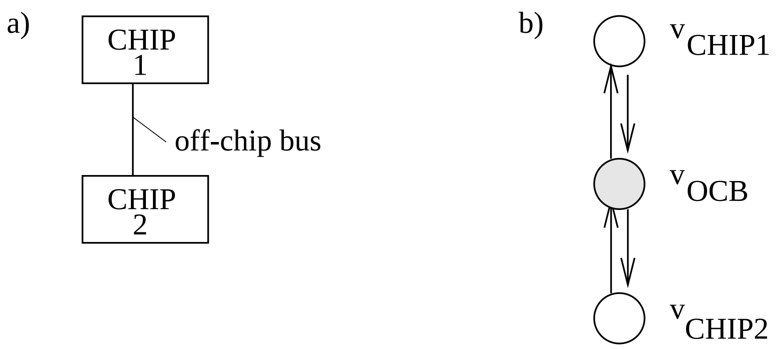

Figure 2.3: An example of a multi-chip architecture (a) and the corresponding chip graph G_C (b).

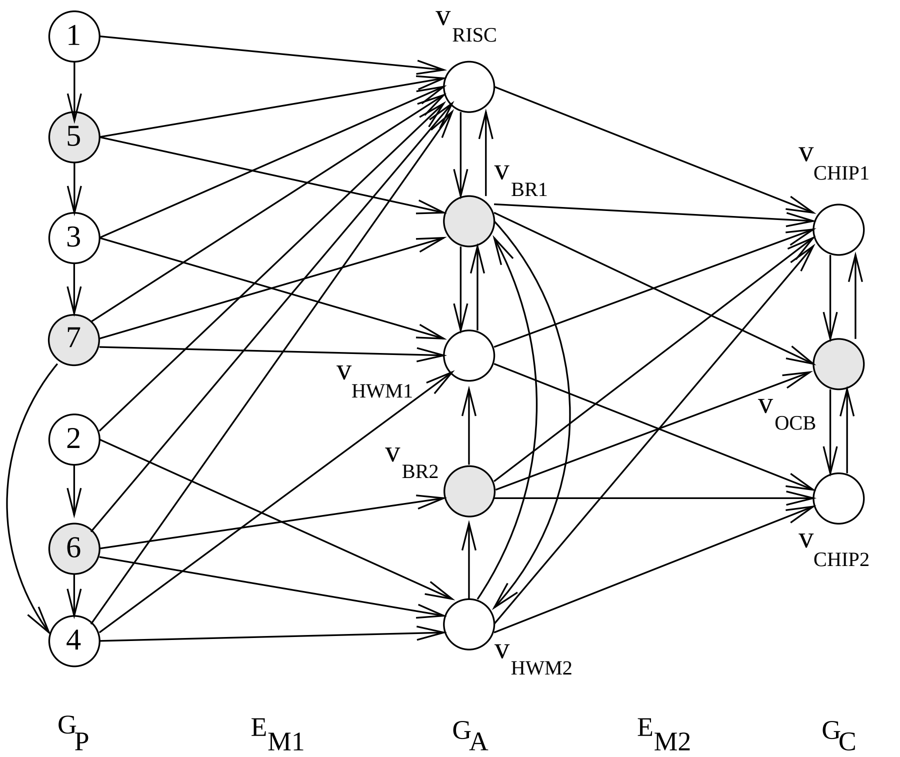

Figure 2.4: An example of a specification graph G_S

Definition 2.1.2 (Specification Graph) *A* specification graph *is a graph $G_S = (V_S, E_S)$ consisting of D dependence graphs $G_i(V_i, E_i)$ for $1 \leq i \leq D$ and a set of* mapping edges E_M. *In particular, $V_S = \bigcup_{i=1}^{D} V_i$, $E_S = \bigcup_{i=1}^{D} E_i \cup E_M$ and $E_M = \bigcup_{i=1}^{D-1} E_{Mi}$, where $E_{Mi} \subseteq V_i \times V_{i+1}$ for $1 \leq i < D$.*

Consequently, the specification graph consists of several layers of dependence graphs and mapping edges which relate the nodes of two neighboring dependence graphs. These layers correspond to levels of abstractions, for example algorithm description (problem graph), architecture description (architecture graph) and system description (chip graph). The edges represent user-defined mapping constraints in the form of a relation: 'can be implemented by'.

Example 2.1.4 *Fig. 2.4 shows an example of a specification graph using the problem graph of example 2.1.1 (left), the architecture graph of example 2.1.2 (middle) and the chip graph of example 2.1.3 (right). The edges between the two subgraphs are the additional edges E_{M1} and E_{M2} that describe all possible mappings. For example, operation v_1 can be executed only on v_{RISC}. Operation v_2 can be executed on v_{RISC} or v_{HWM2}.*

Note that it can be useful to map communication nodes of the problem graph to functional resources: If both predecessor and successor node of a communication node are mapped to the same functional resource, no communication is necessary and the communication is internal. *In this case, the communication can be viewed to be handled by the functional resource.*

Also, communication v_7 can be executed by v_{BR1} or within v_{RISC} or v_{HWM1}. It can also be seen, that the specification allows the RISC *processor, the hardware modules* HWM1, HWM2 *and the communication modules* BR1, BR2 *to be implemented in* CHIP1. *The communication* BR1 *can either be handled by* CHIP1 *or by the off-chip bus* OCB.

This way, the model of a specification graph allows a flexible expression of the expert knowledge about useful architectures and mappings.

In order to describe a concrete mapping, i.e. an *implementation*, the term *activation* of nodes and edges of a specification graph is defined. Based on this, *allocation*, *binding* and *scheduling* will formally be defined in the next section.

Definition 2.1.3 (Activation) *The* activation *of a specification graph $G_S(V_S, E_S)$ is a function $a : V_S \cup E_S \mapsto \{0, 1\}$ that assigns to each edge $e \in E_S$ and each node $v \in V_S$ the value* 1 *(*activated*) or* 0 *(*not activated*).*

The activation of a node or edge of a dependence graph describes its use. In the examples used so far, all nodes and edges of the problem graph contained in G_S may be necessary, i.e. activated. The determination of a an implementation can be seen as the task of assigning activity values to each node and each edge of the architecture graph and/or chip graph. An activated mapping edge represents the fact that the source node is implemented on the target node.

2.2 System Synthesis

Now, the term of implementation will be formally defined using the main tasks of synthesis, namely *allocation*, *binding*, and *scheduling*.

Definition 2.2.1 (Allocation) *An* allocation *α of a specification graph is the subset of all activated nodes and edges of the dependence graphs, i.e.,*

$$\alpha = \alpha_V \cup \alpha_E$$

$$\alpha_V = \{v \in V_S | a(v) = 1\}$$

$$\alpha_E = \bigcup_{i=1}^{D} \{e \in E_i | a(e) = 1\}$$

Definition 2.2.2 (Binding) *A* binding β *is the subset of all activated mapping edges, i.e.,*

$$\beta = \{e \in E_M | a(e) = 1\}$$

Definition 2.2.3 (Feasible Binding) *Given a specification G_S and an allocation α. A* feasible binding β *is a binding that satisfies*

1. *Each activated edge $e \in \beta$ starts and ends at an activated node, i.e.*

$$\forall e = (v, \tilde{v}) \in \beta \; : \; v, \tilde{v} \in \alpha$$

2. *For each activated node $v \in \alpha_V$ with $v \in V_i$, $1 \leq i < D$ exactly one outgoing edge $e \in V_M$ is activated, i.e.*

$$| \; \{e \in \beta \; | e = (v, \tilde{v}), \tilde{v} \in V_{i+1}\} \; | = 1$$

3. *For each activated edge $e = (v_i, v_j) \in \alpha_E$ with $e \in E_i$, $1 \leq i < D$*

   - *either both operations are mapped onto the same node, i.e.*

$$\tilde{v}_i = \tilde{v}_j \quad \textit{with} \quad (v_i, \tilde{v}_i), (v_j, \tilde{v}_j) \in \beta$$

   - *or there exists an activated edge $\tilde{e} = (\tilde{v}_i, \tilde{v}_j) \in \alpha_E$ with $\tilde{e} \in E_{i+1}$ to handle the communication associated with edge e, i.e.*

$$(\tilde{v}_i, \tilde{v}_j) \in \alpha_E \quad \textit{with} \quad (v_i, \tilde{v}_i), (v_j, \tilde{v}_j) \in \beta$$

The synthesis procedure which will be presented simultaneously determines allocation and binding. On the other hand it is useful to determine the set of feasible allocations and feasible bindings in order to restrict the search space.

Definition 2.2.4 (Feasible Allocation) *A* feasible allocation α *is an allocation that allows at least one feasible binding β.*

But the next theorem shows that the calculation of a feasible binding and therefore the test of an allocation for feasibility is hard. This result will influence the coding of allocation and binding in the evolutionary algorithm.

Theorem 2.2.1 *The determination of a feasible binding is NP-complete.*

Proof: See appendix A. □

Finally, it is necessary to define a *schedule*. Here, we use the execution time delay $delay(v, \beta)$ of an operation associated to node v of a problem graph G_P. In order to be as general as possible at this point, we suppose that the execution time depends on a particular binding β. For example, the time delay of an operation may depend on the resource it is bound to.

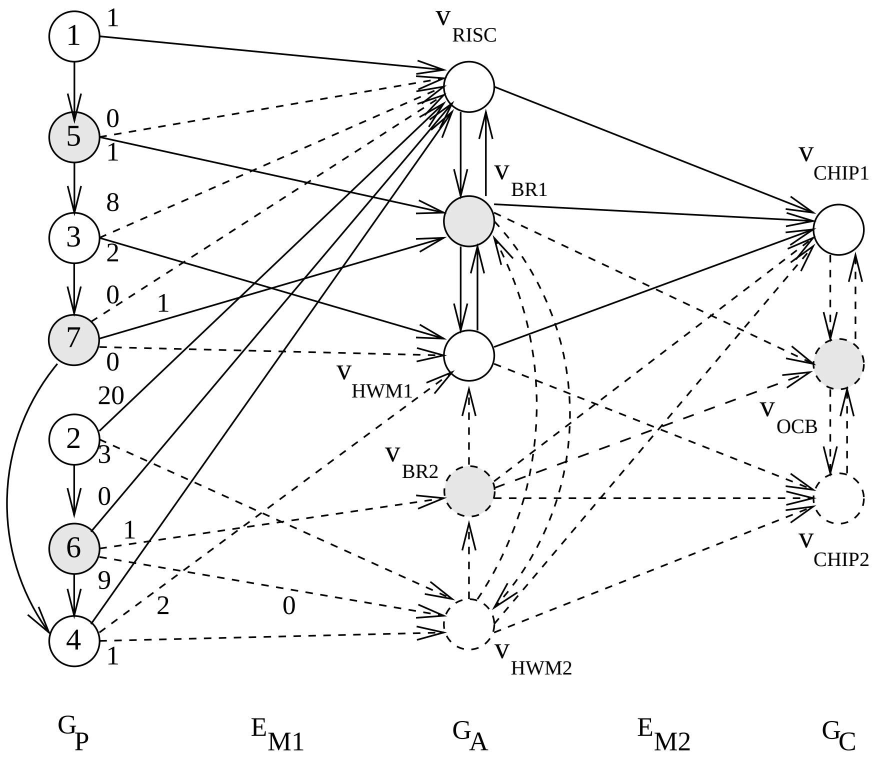

Figure 2.5: An example of an implementation from the specification given in Fig. 2.4.

Definition 2.2.5 (Schedule) *Given specification G_S containing a problem graph $G_1 = G_P$, a feasible binding β, and a function delay which determines the execution time $delay(v, \beta) \in \mathbb{Z}^+$ of a node $v \in V_P$. A* schedule *is a function $\tau : V_P \mapsto \mathbb{Z}^+$ that satisfies for all edges $e = (v_i, v_j) \in E_P$:*

$$\tau(v_j) \geq \tau(v_i) + delay(v_i, \beta)$$

$\tau(v_i)$ may be interpreted as the start time of the operation of node $v_i \in V_P$. For example, the delay values of communication nodes might be considered as the time units necessary to transfer the associated data using the node bound to it. Usually, these values depend not only on the amount of data transferred but also on the capacity of the resource, for example the bus width and the bus transfer rate. Therefore, the delay may depend on the actual binding. The special case of an *internal communication* is described in the next example.

Example 2.2.1 *Consider the case that the delay values of a node $v_i \in V_P$ only depend on the binding of that particular node. Then the delay values can be associated with the edges E_{M1} (see Fig. 2.5). All execution times of all operations* *on different resources are shown. For example, operation v_3 takes 8 time units if executed on the RISC (v_{RISC}) but 2 time units if mapped to the hardware module HWM1. Note that internal communications (like the mapping of v_5 to v_{RISC}) are modeled to take zero time.*

Definition 2.2.6 (Implementation) *Given a specification graph G_S a* (valid) implementation *is a triple (α, β, τ) where α is an allocation, β is a feasible binding, and τ is a schedule.*

Example 2.2.2 *Fig. 2.5 shows an implementation of the specification from Fig. 2.4. The nodes and edges not allocated are shown dotted, as well as the edges $e \in E_M$ that are not activated. The allocation of nodes is $\alpha_V = \{v_{RISC}, v_{HWM1}, v_{BR1}, v_{CHIP1}\}$ and the binding is $\beta = \{(v_1, v_{RISC}),\ (v_2, v_{RISC}),\ (v_3, v_{RISC}),\ (v_4, v_{HWM1}),\ (v_5, v_{RISC}),\ (v_6, v_{BR1}),\ (v_7, v_{BR1}),\ (v_{RISC}, v_{CHIP1}),\ (v_{BR1}, v_{CHIP1}),\ (v_{HWM1}, v_{CHIP1})\}$. This means that all architecture components are bound to CHIP1.*

Note that communication modeled by v_6 can be handled by the functional resource v_{RISC} as both predecessor node (v_2) and successor node (v_4) are mapped to resource v_{RISC}. A schedule is $\tau(v_1) = 0$, $\tau(v_2) = 1$, $\tau(v_3) = 2$, $\tau(v_4) = 21$, $\tau(v_5) = 1$, $\tau(v_6) = 21$, $\tau(v_7) = 4$.

2.3 The Task of System Synthesis

With the model introduced above the task of system synthesis can be formulated an optimization problem.

Definition 2.3.1 (System Synthesis) *The task of* system synthesis *is the following optimization problem:*

minimize $f(\alpha, \beta, \tau)$,
subject to

> *α is a feasible allocation,*
> *β is a feasible binding,*
> *τ is a schedule, and*
> $g_i(\alpha, \beta, \tau) \geq 0,\ i \in \{1, \ldots, q\}$.

The constraints on α, β and τ define the set of valid implementations. Additionally, there are function $g_i, i = 1, \ldots, q$, that together with the objective function f describes the optimization goal.

Example 2.3.1 *The specification graph G_S may consist only of a problem graph G_P and an architecture graph G_A. Consider the task of latency minimization* *under resource constraints, i.e., an implementation is searched that is as fast as possible but does not exceed certain cost $MAXCOST$. To this end, a function $cost : V_A \mapsto \mathbb{Z}^+$ is given which describes the $cost(\tilde{v})$ that arises if resource $\tilde{v} \in V_A$ is realized, i.e., $\tilde{v} \in \alpha$. The limit in costs is expressed in a constraint $g_1(\alpha, \beta, \tau) = MAXCOST - \sum_{\tilde{v} \in \alpha} cost(\tilde{v})$. The corresponding objective function may be $f(\alpha, \beta, \tau) = max\{\tau(v) + delay(v, \beta)) \mid v \in V_P\}$.*

The objective function may be arbitrary complex and reflect the specific optimization goal. Likewise, the additional constraints g_i can be used to reduce the number of potential implementations. In the next section some examples of refinements are introduced.

2.4 Examples of Model Refinement

So far, we introduced our basic ideas of modeling the task of system synthesis. This model has been shown to be concise, useful, and universal as it can handle a broad range of mapping problems.

In the following several examples of model refinements are discussed that may be necessary to model a certain design style or a particular design goal. These refinements will consider the modeling of program memory, resource sharing and chip boundaries.

2.4.1 Model for Resource Sharing

A simple cost model was given in Example 2.3.1 that only considers the cost of realizing a certain hardware module. In the following a more detailed model is introduced that takes program size or chip area into account. For simplicity it is assumed that a a specification consists only of a problem graph G_P and an architecture graph G_A.

Definition 2.4.1 (Cost) *The function $c_b : V_A \mapsto \mathbb{Z}^+$ describes the basic cost $c_b(\tilde{v})$ that arises if resource $\tilde{v} \in V_A$ is realized.*

The following definition reflects additional costs that occur if more than one functionality is mapped to the same resource. These costs can for example model program memory or chip area that is specific for each different task.

Definition 2.4.2 (Addcost) *The function $c_a : E_M \mapsto \mathbb{Z}^+$ describes for each edge $e = (v, \tilde{v}) \in E_M$ the additional cost of implementing node $v \in V_P$ on resource $\tilde{v} \in V_A$.*

However, there are often different tasks that can share the same program memory or share the same hardware area. In this case summing up additional costs is not appropriate. As a remedy, so-called *types* are defined.

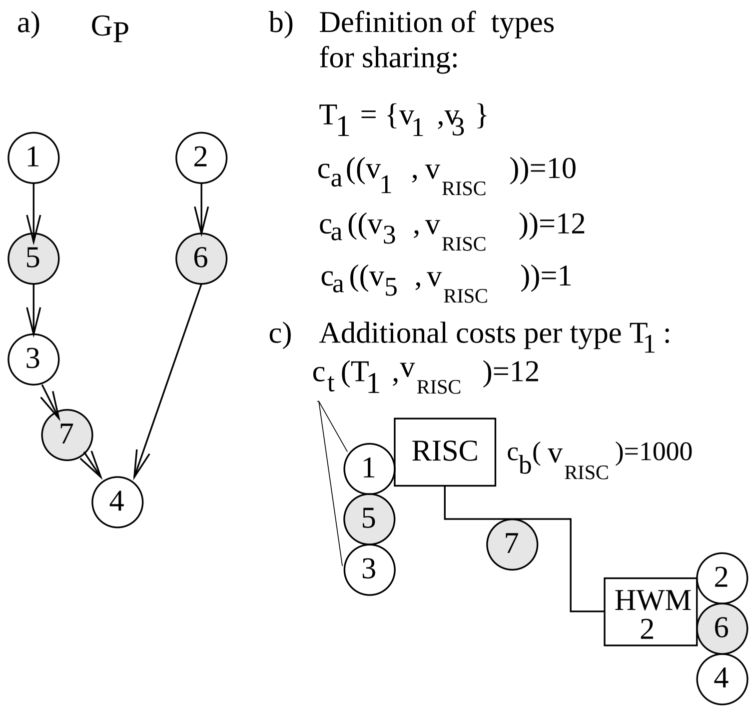

Figure 2.6: Modeling of resource sharing (program memory or hardware modules) by definition of types for sharing.

Definition 2.4.3 (Types) *The node set V_P is partitioned into* types $T \in \mathcal{T}$*, i.e., each node has exactly one type from the set of types.*

The additional costs of all nodes of the same type T mapped onto the same resource $\tilde{v} \in V_A$ cause only additional costs

$$c_t(T, \tilde{v}) = \max\{c_a(e) | v \in T : e = (v, \tilde{v}) \in \beta\}$$

With this definitions a more realistic modeling of the cost of an implementation is possible. In particular, the cost $c(\alpha, \beta)$ of an implementation is specified by

$$c(\alpha, \beta) = \sum_{\tilde{v} \in \alpha \cap V_A} c_h(\tilde{v}, \beta)$$

where $c_h(\tilde{v}, \beta)$ specifies the cost for each allocated hardware component, i.e.,

$$c_h(\tilde{v}, \beta) = c_b(\tilde{v}) + \sum_{T \in \mathcal{T} : (v, \tilde{v}) \in \beta \wedge v \in T} c_t(T, \tilde{v})$$

This way, a constraint on a resource $v \in \alpha \cap V_A$ can be specified that limits the maximum cost for this resource to $MAXCOST_v$, and thereby limits the maximum program size or chip area:

$$g_v = MAXCOST_v - c_h(\tilde{v}, \beta)$$

Similar, a constraint on the total cost of the implementation can be expressed by

$$g = MAXCOST - c(\alpha, \beta)$$

Example 2.4.1 *Fig. 2.6 demonstrates the use of function types to model costs of shared resources. One function type is defined in Fig. 2.6b). $T_1 = \{v_1, v_3\}$, all other nodes belong to a dedicated function type, i.e., $\mathcal{T} = \{T_1, \{v_2\}, \{v_4\}, \{v_5\}, \{v_6\}, \{v_7\}\}$. As an example the calculation of the cost of the resource RISC ($c_1(v_{RISC}, \beta)$) is carried out for the particular binding β given by the implementation in Fig. 2.6c).*

Let the additional costs for implementing node v_1 on the RISC resource be $c_a((v_1, v_{RISC})) = 10$ and that of v_3 on the RISC resource be $c_a((v_3, v_{RISC})) = 12$. As both nodes are implemented by the same resource the addition cost are reflected by the type costs, i.e.,

$$\begin{aligned} c_t(T_1, v_{RISC}) &= \max\{c_a((v_1, v_{RISC})), c_a((v_3, v_{RISC}))\} \\ &= \max\{10, 12\} = 12 \end{aligned}$$

The additional cost of implementing operation v_5 is $c_a(v_5) = 1$. As there is no other node that belongs to the same type as v_5 the type cost calculate to

$$c_t(\{v_5\}, v_{RISC}) = \max\{c_a((v_5, v_{RISC}))\} = \max\{1\} = 1$$

Furthermore, assume the basic cost for realizing RISC are $c_b(v_{RISC}) = 1000$ then the total cost of resource $RISC$ compute to

$$\begin{aligned} c_1(v_{RISC}) &= c_b(v_{RISC}) + \sum_{T \in \{T_1, \{v_5\}\}} c_t(T, v_{RISC}) \\ &= c_b(v_{RISC}) + c_t(T_1, v_{RISC}) + c_t(\{v_5\}, v_{RISC}) \\ &= 1000 + 12 + 1 = 1013 \end{aligned}$$

2.4.2 Model for Chip Boundaries

Module and bus costs strongly depend on whether a system is implemented on a single chip or on a board using multiple chips. Costs and delay values depend on the selection of a layout macro cell in case of single-chip design or on the package and die specification in case of a multiple-chip design.

The selection of a chipset can easily obtained by introducing a dedicated level of abstraction, the chip graph G_C (as in example 2.1.3). It is thereby assumed that a chip is only a container for all resources that are mapped to it and that all these resources operate concurrently. Similarly an off-chip bus (as in example 2.1.3) is only a bundle of wires. The possible sequentialization of operations or communications is done by the mapping of operations in G_P to resources in

G_A. Therefore, the specification graph G_S consists of a problem graph G_P, an architecture graph G_A and a chip graph G_C.

As an example modeling the number of pins is carried out in order to specify an upper bound $PINLIMIT$ for a chip.

The number of pins necessary can be split into a fixed part pin_c (e.g. for power supply) and a part that depends on the number of off-chip communications. Let $V_B(\bar{v}, \beta)$ be the set of resources $\tilde{v} \in V_A$ that handle a communication of chip $\bar{v} \in V_C$ to the outside, i.e.,

$$\begin{aligned} V_B(\bar{v}, \beta) = \{\tilde{u} \in V_A| \quad & \exists \tilde{v} \in V_A, \bar{u} \in V_C \text{ with} \\ & (\tilde{u}, \bar{u}), (\tilde{v}, \bar{v}) \in \beta \wedge ((\bar{u}, \bar{v}), (\tilde{u}, \tilde{v}) \in \alpha \vee (\bar{v}, \bar{u}), (\tilde{v}, \tilde{u}) \in \alpha)\} \end{aligned}$$

Assume furthermore a function $width : V_A \mapsto \mathbb{Z}^+$ that assigns to each node of the architecture graph the width of a bus that is modeled by this node. Then the number of pins required for a chip $\bar{v}$ is given by

$$pin(\bar{v}) = pin_c(\bar{v}) + \sum_{\tilde{v} \in V_B(\bar{v})} width(\tilde{v})$$

The limitation of pins to $PINLIMIT_{\bar{v}}$ of a chip $\bar{v}$ can now be expressed by the following constraint:

$$g = PINLIMIT_{\bar{v}} - pin(\bar{v})$$

Example 2.4.2 *Consider the specification of example 2.1.4. The width for the bus resources may be* $width(v_{BR1}) = width(v_{BR2}) = 16$*, the number of the fixed pins of CHIP1* $pin_c = 2$*. Furthermore, assume a constraint that limits the number of pins of CHIP1 to 20, i.e.,* $g = 20 - pin_{v_{CHIP1}}$.

Consider now the implementation shown in Fig. 2.7a) with binding β_a*. For sake of clarity, only the architecture graph* G_A *and the chip graph* G_C *are shown and the problem graph* G_P *is omitted. The set of bus resources that handle communication to the outside of CHIP1 is* $V_B(v_{CHIP1}, \beta_a) = \{v_{BR1}, v_{BR2}\}$*. The pins of CHIP1 then calculate to*

$$\begin{aligned} pin(v_{CHIP1}) &= pin_c(v_{CHIP1}) + \sum_{\tilde{v} \in V_B(v_{CHIP1}, \beta_a)} width(\tilde{v}) \\ &= 2 + width(v_{BR1}) + width(v_{BR2}) = 2 + 16 + 16 = 34 \end{aligned}$$

Hence, the implementation in Fig. 2.7a) violates the constraint as $g = 20 - pin(v_{CHIP1}) = -14 < 0$.

On the other hand, the implementation in Fig. 2.7b) with binding β_b *does satisfy this constraint. As* $V_B(v_{CHIP1}, \beta_b) = \{v_{BR2}\}$ *the pins of CHIP1 calculate to*

$$\begin{aligned} pin(v_{CHIP1}) &= pin_c(v_{CHIP1}) + \sum_{\tilde{v} \in V_B(v_{CHIP1}, \beta_b)} width(\tilde{v}) \\ &= 2 + width(v_{BR2}) = 2 + 16 = 18, \end{aligned}$$

yielding $g = 20 - 18 = 2 \geq 0$.

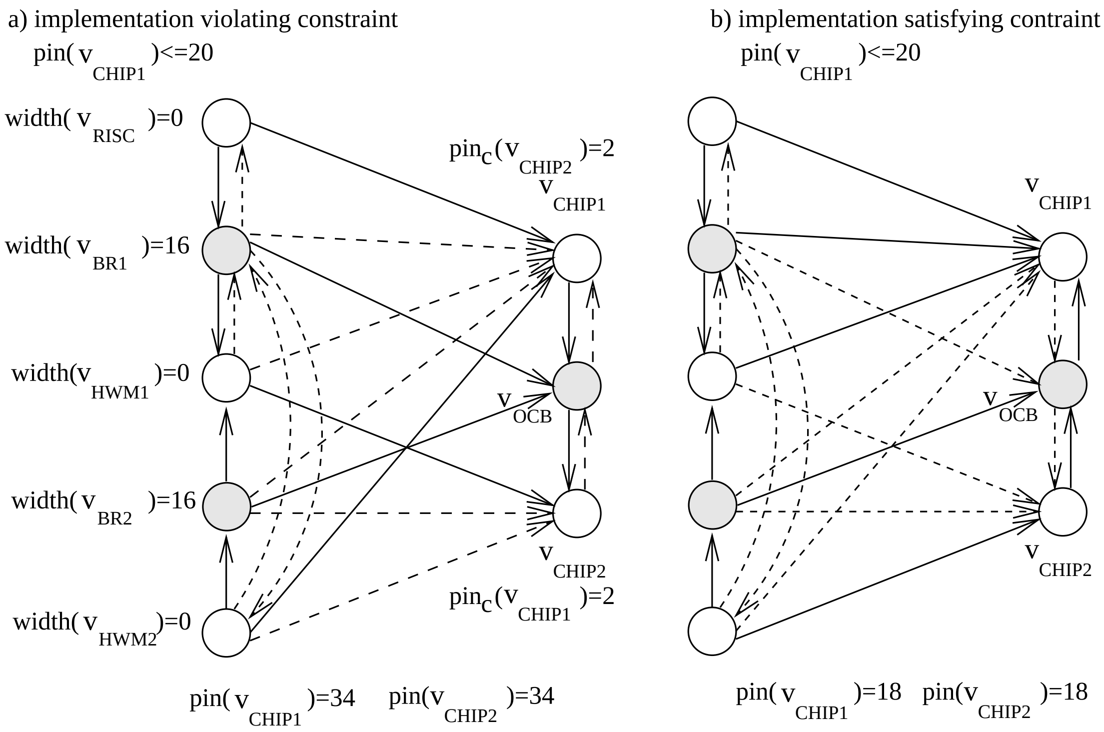

Figure 2.7: Two implementations from the specification of Fig. 2.4: a) violating the pinlimit constraint $pin(v_{CHIP1}) \leq 20$ and b) satisfying the pinlimit constraint $pin(v_{CHIP1}) \leq 20$.

Chapter 3

Optimization Method

In this section the application of an Evolutionary Algorithm is described to solve the problem of system synthesis as specified by Definition 2.3.1. The Evolutionary Algorithm is responsible for the determination of the allocation and of the binding. This is due to the fact that search space for these tasks is large and discrete and the determination of a feasible binding is NP-complete (see theorem 2.2.1). Hence, for reasonable problem sizes exact methods are intractable and EAs are known to work well on such problems. Furthermore, the inherent parallelism of EAs allows a design space exploration during a single optimization run. As there are good heuristics available for the scheduling problem it is not necessary to load the EA with this task. This is why, the schedule for the fixed architecture obtained by the EA is computed by a standard heuristic scheduler.

This devision of work is depicted in Fig. 3.1.

In the remainder of this section first the basic principles of Evolutionary Algorithms are explained, then the coding of implementations as individuals is described. After that, the structure of the fitness function F will be introduced.

3.1 Principle of Evolutionary Algorithms

The Evolutionary Algorithm (EA) is a probabilistic optimization method based on the model of natural evolution. It is characterized by the fact that a number N of potential solutions (called *individuals* $J_i \in \mathbf{J}$, where $\mathbf{J}$ represents the space of all possible individuals) of the optimization problem simultaneously sample the search space. This *population* $P = \{J_1, J_2, ..., J_N\}$ is modified according to the natural evolutionary process: after initialization, selection and recombination are executed in a loop for a fixed number of iterations. Each run of the loop is called a generation and P_k denotes the population at generation k.

The selection operator is intended to improve the average quality of the population by giving individuals of higher quality a higher probability to be copied into the next generation. Selection thereby focuses the search on promising regions in

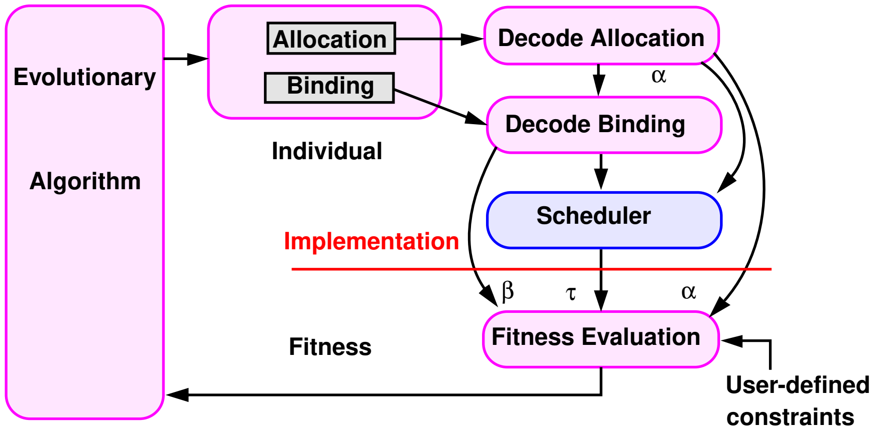

Figure 3.1: The decoding of an individual to an implementation.

the search space. The quality of an individual is measured by a fitness function $F : \mathbf{J} \mapsto \mathbf{R}$. Various selection methods exist (see e.g., [Blickle and Thiele, 1995]).

Recombination changes the genetic material in the population either by crossover or by mutation in order to exploit new points in the search space. Depending on the problem to be solved, various codings for EAs exist, e.g., the individuals are represented by bit strings, vectors of integers or reals, trees, graphs. The choice of coding determines also the recombination operator.

3.2 Coding of Implementations

To obtain a meaningful coding one has to address the question how to handle infeasible allocations and infeasible bindings suggested by the EA. Obviously, if allocations and bindings may be randomly chosen, a lot of them can be infeasible. In general, there are two different methods to handle these invalid implementations:

- Punishing: one can punish these "bad" individuals with a penalty value. This leads to discarding these individuals during the following selection phases. But depending on the specification, a lot of possible allocations and bindings might be infeasible. This would result in a needle-in-the-haystack search and the EA would reveal a very bad performance.

- Repairing: on the other hand, one could "repair" invalid allocations and bindings with some mechanism and incorporate domain knowledge in these repair mechanisms. But as the determination of a feasible allocation or binding is NP-complete, this would result in solving a NP-complete task for every individual to be repaired.

These considerations lead to the following compromise: The randomly generated allocations of the EA are partially repaired using a heuristic. Possible complications detected later on during the calculation of the binding will be considered by a penalty.

A specification graph G_S may consist of D subgraphes $G_i, 1 \leq i \leq D$ corresponding to $D-1$ mapping tasks. In order to handle these mapping tasks the allocation and binding of the levels is done sequentially from level 1 to level $D-1$ (see algorithm 1). In each level three steps are executed:

- First, the allocation of nodes V_{i+1} is decoded from the individual and repaired with a simple heuristic (the function *node_allocation()*),
- next the binding of the edges $e \in E_{Mi}$ is performed (the function *binding()*), and
- finally the allocation is updated in order to eliminate unnecessary nodes $v \in V_{i+1}$ from the allocation and add all necessary edges $e \in E_{i+1}$ to the allocation (the function *update_allocation()*).

One iteration of the loop results in a feasible allocation and binding of the nodes and edges of G_i to the nodes and edges of G_{i+1}. If no feasible binding could be found the whole decoding of the individual is aborted.

In the following the three functions node_allocation(), binding(), and update_allocation() are explained in detail.

```
Algorithm 1: (Decoding)

Input: The individual J consisting of allocations alloc_i,
       repair allocation priority lists L_Ri, binding order lists LOi,
       and binding priority lists L_Bi(v), for all 1 ≤ i < D.
Output: The allocation α and the binding β if both are feasible,
       ({}, {}) if no feasible binding is represented by the individual J

decode(J):
    α ← V_1 ∪ E_1
    β ← {}
    for i ← 1 to D-1 do
            ᾱ ← node_allocation(alloc_i(J), L_Ri(J))
            β̄ ← binding(L_Bi(J), L_Oi(J), ᾱ)
            if β̄ = {}
                    return ({}, {})
            endif
            β ← β ∪ β̄
            α ← α∪ update_allocation(ᾱ, β̄)
    od
    return (α, β)
```

3.2.1 The Function node_allocation()

The allocation of nodes is directly encoded in the chromosome, i.e., for each node $v \in V_{i+1}$ there exists a vector $alloc_i$ that codes the activation of v, i.e., $a(v) = alloc_i[v]$. This simple coding might result in a many infeasible allocations, e.g. 77% of randomly generated allocations of the specification in example 2.1.4 (for the first mapping level from G_P to G_A) lead to infeasible allocations. Due to this fact a simple repair heuristic is applied. This heuristic only adds new nodes $v \in V_{i+1}$ to the allocation and reflects the simplest case of infeasibility that may arise from not-executable functional nodes: Consider the set $V_{Bi} \subseteq V_i$ that contains all nodes that can not be executed, because not a single corresponding resource node is allocated,i.e., $V_{Bi} = \{v \in V_i | \forall \tilde{v} \in V_{i+1} : (v, \tilde{v}) \in E_{Mi} \wedge a(\tilde{v}) = 0\}$. To make the allocation feasible (in this sense) for each $v \in V_{Bi}$, at most one $\tilde{v} \in V_{i+1}$ is added, until feasibility in the sense above is achieved. This way, the number of infeasible allocation for the first mapping level of example 2.1.4 could be reduced to 6.5%.

Algorithm 2: (Allocation)

Input: The allocation $alloc_i$ and repair allocation priority list L_{Ri} of individual J
Output: The allocation α

allocation($alloc_i, L_{Ri}$):
  $\alpha \leftarrow \{\}$
  forall $\tilde{v} \in V_{i+1}$ **do**
    if $(alloc_i[\tilde{v}] = 1)$
      $\alpha \leftarrow \alpha \cup \{\tilde{v}\}$
    endif
  od
  $V_B \leftarrow$ not_bindable_nodes(α)
  $\tilde{v}_r \leftarrow$ first(L_{Ri})
  while $(V_B \neq \{\})$ **do**
    if ($V_B \neq$ not_bindable_nodes($\alpha \cup \{\tilde{v}_r\}$))
      $\alpha \leftarrow \alpha \cup \{\tilde{v}_r\}$
      $V_B \leftarrow$ not_bindable_nodes(α)
    endif
    $\tilde{v}_r \leftarrow$ next(L_{Ri})
  od
  return α

The order in which the new resources are added has a large influence on the resulting allocation. For example, one could be interested in an additional allocation with minimal costs or with maximum performance. As this depends on the optimization goal expressed in the objective function f the order should automatically be adapted. This will be achieved by an *repair allocation priority list* L_{Ri} coded for each mapping level in the individual. In this list, all nodes $v \in V_{i+1}$ are contained and the order in the list determines the order the nodes can be added to the allocation. This list also undergoes genetic operators like crossover and mutation and can therefore be optimized by the Evolutionary Algorithm.

In summary, algorithm 2 is obtained to decode the allocation of nodes.

Example 3.2.1 *Consider the specification of Example 2.1.4. In Fig. 3.2 the allocation information for the first mapping level as stored in the individual is shown on the left. The direct decoding of the allocation string yields the allocation* $\alpha = \{v_{BR1},\ v_{HWM1}\}$. *This allocation is not valid as there exists no allocated resource for executing* $v_1, v_2 \in G_P$. *This allocation is then repaired using the repair allocation priority list.* v_{HWM1} *and* v_{BR1} *are already in the allocation. The allocation of* v_{RISC} *resolves the conflict for both* v_1 *or* v_2*, hence it is allocated.*

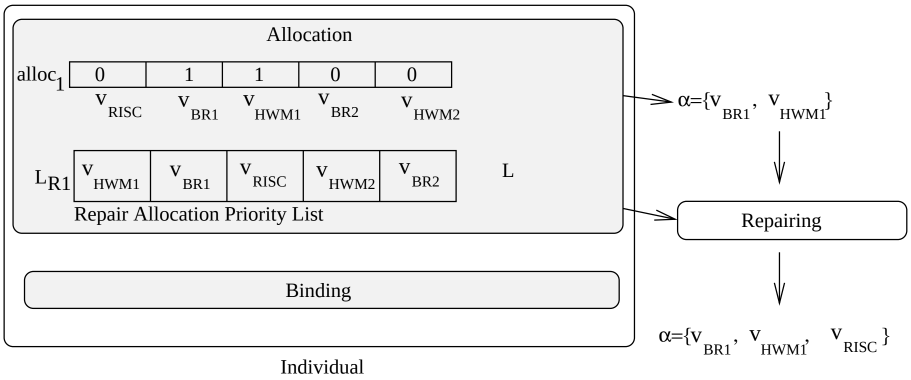

Figure 3.2: The decoding of an allocation.

The rest of the list is then ignored, as no node remains with a conflict.

Note that a different repair allocation priority list may result in a different allocation. If the entries of v_{RISC} and v_{HWM2} are swapped in the list, both v_{HWM2} and v_{RISC} would be allocated as the allocation of v_{HWM2} only solves the conflict for v_2 but not for v_1. This would code a more "lavish" allocation.

3.2.2 The Function binding()

A binding is obtained by activating exactly one edge $e \in E_{Mi}$ adjacent to a allocated node v for each $v \in V_i$. The problem of coding the binding lies in the strong inter-dependence of the binding and the current allocation. As crossover or mutation might change the allocation, a directly encoded binding could be meaningless for a different allocation. Hence, a coding of the binding is of interest that can be interpreted *independently* of the allocation. This is achieved in the following way:

For each node $v \in V_i$ a list is coded as allele that contains all adjacent edges $e \in E_{Mi}$ of v. This list is seen as a priority list and the first edge e_k with $e_k = (v, \tilde{v})$ that gives a feasible binding is included in the binding, i.e., $a(e_k) := 1$. The test of feasibility is directly related to the definition of a feasible binding (Definition 2.2.3). Details are given in the two algorithms. Note that it is possible, that no feasible binding is specified by the individual. In this case, β is the empty set, no schedule can be computed, and the individual will be given a penalty value as fitness value.

Algorithm 3: (Binding)

Input: The binding priority lists $L_{Bi}(v)\ \forall v \in V_i$, the binding order list L_{Oi}, and the allocation α of an individual J.
Output: The binding β, or $\{\}$ if no feasible binding was decoded.

```
binding(L_Bi, L_Oi, α_i):
    β ← {}
    forall_listelements u ∈ L_Oi ∩ α do
        e' ←nil
        forall_listelements e ∈ L_B(u) ∩ α do
            if (is_valid_binding(e, β, α ))
                e' ← e
                break
            endif
        od
        if (e' = nil)
            return {}
        else
            β ← β ∪ {e'}
        endif
    od
    return β
```

Example 3.2.2 *Fig. 3.3 shows an example of a binding as it is coded in the individual for the first mapping level of the specification from example 2.1.4 (Fig. 2.4). The binding order specified by the list* L_{O1} *is* v_1, v_4, v_2, v_3, v_6, v_5, v_7. *The binding priority lists for all nodes are also shown. For example, the priority list for node* v_6 *implies to bind* v_6 *to the resource BR2 (point-to-point bus). If this is not possible, it should be bound to HWM2 and if this is also not possible it should be bound to BR1 (shared bus). As the allocation from example 3.2.1 does not contain HWM2 and BR2, this node is finally bound to BR1.*

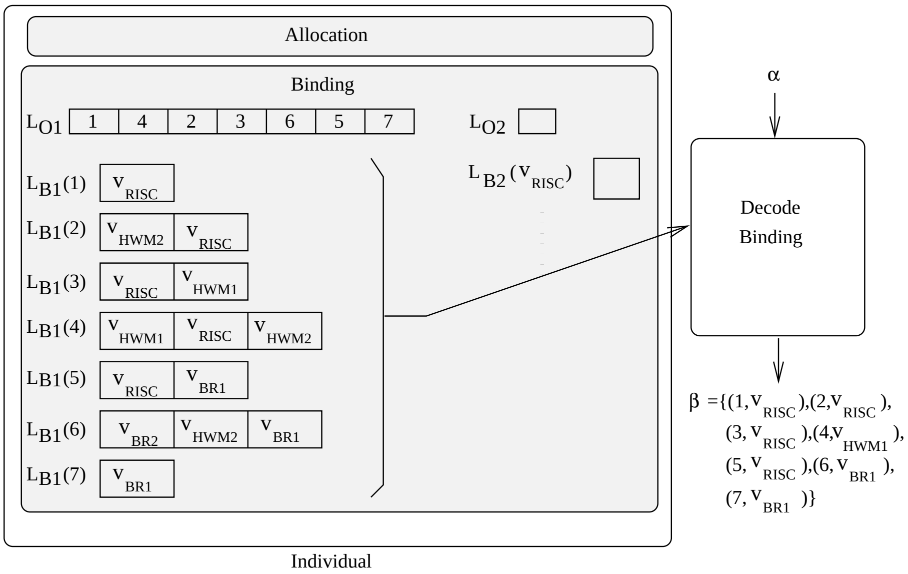

Figure 3.3: An example of the coding of a binding and the resulting binding using the allocation of Fig. 3.2. The obtained architecture is the one depicted in Fig. 2.5 of Example 2.2.2.

Algorithm 4: (Check for Valid Binding)

Input: The edge $e = (v, \tilde{v}) \in V_{Mi}$,
the already computed binding β and allocation α
Output: **true** if e can be included to the binding **false** else.

is_valid_binding($e = (v, \tilde{v}), \beta, \alpha$):
forall $\hat{e} = (w, v) \in E_i \cap \alpha$ **do**
if $((w, \tilde{w}) \in \beta \wedge \tilde{v} \neq \tilde{w} \wedge (\tilde{w}, \tilde{v}) \notin E_{i+1})$
return false
endif
od
forall $\hat{e} = (v, w) \in E_i \cap \alpha$ **do**
if $((w, \tilde{w}) \in \beta \wedge \tilde{v} \neq \tilde{w} \wedge (\tilde{v}, \tilde{w}) \notin E_{i+1})$
return false
endif
od
return true

3.2.3 The Function update_allocation()

In this function nodes of the allocation that are not used will be removed from the allocation. Furthermore, all edges $e \in E_{i+1}$ are added to the allocation that are necessary to obtain a feasible binding.

Algorithm 5: (Update Allocation)

Input: The current allocation α and binding β
of the mapping level

Output: The update allocation, i.e., the allocation where unnecessary nodes are removed from an necessary edges are added to the allocation.

update_allocation(α, β):
  forall $\tilde{v} \in V_{i+1} \cap \alpha$ **do**
    if $\nexists v \in V_i : (v, \tilde{v}) \in \beta$
      $\alpha \leftarrow \alpha \setminus \{\tilde{v}\}$
    endif
  od
  forall $e = (v, u) \in E_i \cap \alpha$ **do**
    $\tilde{e} \leftarrow (\tilde{u}, \tilde{v})$ with $(v, \tilde{v}), (u, \tilde{u}) \in \beta$
    $\alpha \leftarrow \alpha \cup \{\tilde{e}\}$
  od
  return α

Example 3.2.3 *Consider the allocation and binding of example 3.2.1 and 3.2.2, respectively. No node $\tilde{v} \in V_A$ is removed from the allocation, as no unnecessary nodes have been allocated. The computing of the edges $\tilde{v} \in E_A$ leads to the following set that is added to the allocation: $\alpha_E = \{(v_{RISC}, v_{BR1}), (v_{BR1}, v_{RISC}), (v_{HWM1}, v_{BR1}), (v_{BR1}, v_{HWM1})\}$. For example the edge $e = (v_3, v_7) \in E_P$ results in the allocation of edge $(v_{BR1}, v_{RISC}) \in E_A$ as node v_3 is binded to resource v_{RISC} and node v_7 is binded to resource v_{BR1}.*

These algorithms describe in detail the transformation of an individual into an allocation α and a binding β. This specific architecture is used as input to the scheduler to obtain the complete implementation.

3.3 Scheduling

The scheduler chosen here is a list scheduler, see, e.g., [DeMicheli, 1994] that schedules the nodes of G_P and obtains the execution delays and resource bindings for each node from the EA as described above. The schedulers task is to perform latency minimization under resource constraints. We chose the mobility of a node as the priority function for the list scheduler using ASAP- (as soon as possible) and ALAP- (as late as possible) schedules with latency bound $MAXLATENCY$. The scheduler returns the latency of the schedule and the start times of all nodes. Then, the complete fitness function can be evaluated for each individual.

3.4 Cost Evaluation: The Choice of the Fitness Function

In the preceding paragraph we explained how one individual codes a particular implementation. No statement was made about the quality of an implementation and the optimization goal for the Evolutionary Algorithm. This will be done next by describing the calculation of the fitness function.

According to Definition 2.3.1 the system-synthesis task was described as an optimization problem. As the particular optimization goal depends on the specific design problem, the objective function f has to be designed individually for each problem. But the restriction to a feasible allocation and binding needs to be considered in any case. Hence, the fitness function F to be minimized can be given as

$$F = f(\alpha, \beta, \tau) + x_a p_a + x_b p_b + \sum_{i=1}^{q} x_i p_i. \tag{3.1}$$

The p values are the penalty terms, i.e., p_a is the penalty term for an infeasible allocation, p_b for a infeasible binding, and $p_1, \ldots, p_q$ are the possible penalties due to the violation of the constraint $g_1, \ldots, g_q$. The boolean variables x denote whether the corresponding constraint is violated or not, e.g., $x_a = 0$ if the allocation is feasible and $x_a = 1$ if the allocation is infeasible.

Example 3.4.1 *As an example consider the adaption of the trade-off between speed and cost of an implementation. A useful definition of the function f is*

$$f(\alpha, \beta, \tau) = c_c f_c + c_t f_t \tag{3.2}$$

where f can be split into two parts: First, there is a fitness contribution concerning the latency of the given implementation (f_t) and second, the fitness concerning the costs of the implementation (f_c). f_t and f_c are normalized by the user-specified constraints $MAXLATENCY$ and $MAXCOST$, i.e., $f_t = \frac{\tau_L(\beta,\tau)}{MAXLATENCY}$, $f_c = \frac{c(\alpha,\beta)}{MAXCOST}$. The sum is normalized such that $c_c f_c + c_t f_t \leq 1$ for all valid implementations by choosing the factors c_t and c_c such that $c_t + c_c = 1$. The value f_t is easily obtained as the minimal latency of a resource- constrained scheduling problem for a given allocation α and binding β. $\tau_L(\beta, \tau)$ gives the latency of the implementation, and is usually defined by $\tau_L(\beta, \tau) = max\{\tau(v) + delay(v, \beta) | v \in V_P\}$. The *value c is given by the amount of allocated hardware (see section 2.4.1). The constraints on cost and time are furthermore transformed into two constrains* $g_1 = MAXLATENCY - \tau_L(\beta, \tau)$ *and* $g_2 = MAXCOST - c(\alpha, \beta)$. *It is convenient to assure that all implementations violating at least one constraint have a worse fitness (higher fitness value) than any valid implementation. As the maximal value of the function* f *is* 1 *for all valid implementations, all penalty terms should be greater than 1. In particular, the following penalty terms have been shown to be useful:*

- $p_1 = 2.0$ *if the latency of the implementation is greater than* $MAXLATENCY$, *i.e., if* $f_t > 1$.
- $p_2 = 1.0$ *if the cost of the allocation* α_I *is greater than* $MAXCOST$, *i.e., if* $f_c > 1$.

This construction of the fitness function ensures by construction that all legal implementations have a fitness value less or equal to 1. By adjusting the coefficients c_c *and* c_t *the optimization goal can be adjusted towards fast implementations or cheap implementations.*

3.5 Parameters of the Evolutionary Algorithm

For the work herein the following parameter setting is used. The selection method is binary tournament selection. The specific chromosome encoding makes special crossover and mutation schemes necessary. In particular, for the allocations α_i uniform crossover was used [Syswerda, 1989], whereas for the several lists (repair allocation priority lists L_R, binding order lists L_O and the binding priority lists $L_B(v)$) order based crossover (also names position-based crossover) is applied (see, e.g. [Syswerda and Palmucci, 1991; Fox and McMahon, 1991]).

The probability of crossover is of 90% and due to the construction of the individuals no special repairing methods are necessary. Mutation is used with a probability of 10%.

The expected run-time of the optimization procedure for design space exploration is proportional to N, k_{max} and the time to evaluate one implementation.

Chapter 4

Case Studies

We explain our methodology using two examples. One is a video codec for image compression, the other example is taken from high-level synthesis. The second example is a solver for a partial differential equation using numerical integration. This example is used to show that the methodology can also be applied to controller & datapath synthesis as a special problem where bus, communication and pinlimits are in general ignored. Interestingly, we will show that several optimal implementations (pareto-points) have been obtained in a single optimization run.

4.1 Video Codec

Fig. 4.1 shows a a block diagram of the behavior of a hybrid coder used in digital signal applications. The coder is used to compress video images using the H.261 standard. Transformative (DCT) coding and predictive coding are combined. The compression factor of a single coded frame is augmented by a predictive method for motion estimation: a block within a frame is predicted from a block of a preceding frame.

The blocks in the behavioral specification in Fig. 4.1 possess a granularity of algorithms. No information is included about timing, architecture an mapping of blocks to resources.

4.1.1 Problem Specification

Fig. 4.2 shows the problem graph G_P of the video codec in Fig. 4.1. In addition to the operations of the coder, a second subgraph is shown which represents the problem graph for the decoder.

We want to map the complete graph onto a target architecture that is shown in Fig. 4.3. The architecture consists of a shared bus, several memory modules, two programmable RISC processors and two predefined hardware modules, namely a

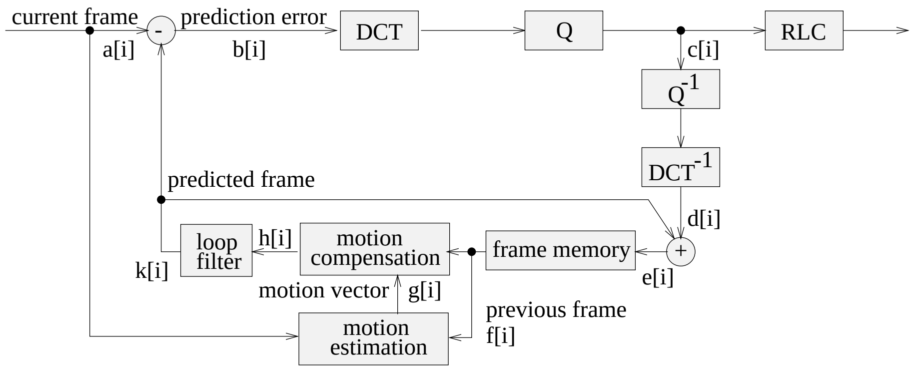

Figure 4.1: Behavioral specification of a video codec for video compression.

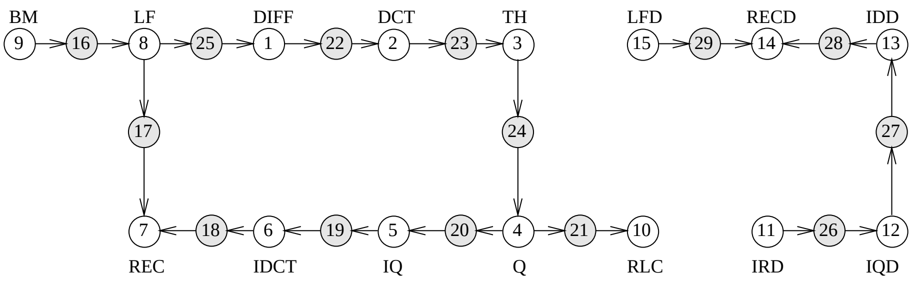

Figure 4.2: Problem graph of the video codec in Fig. 4.1.

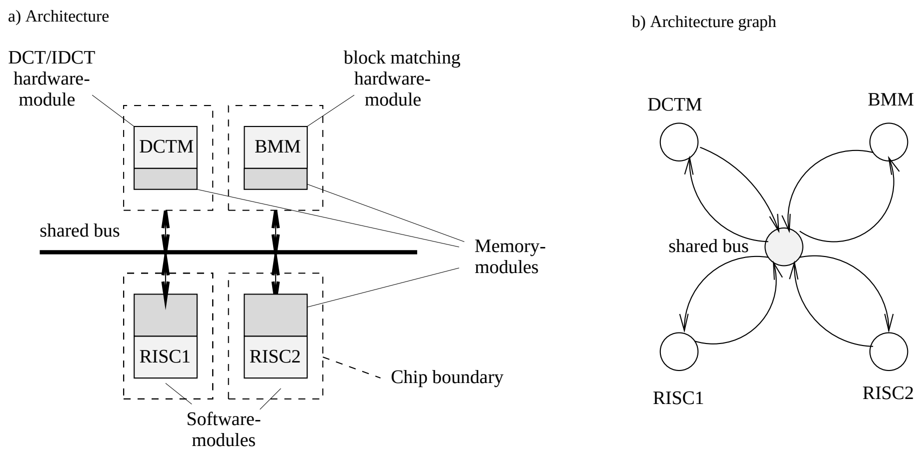

Figure 4.3: Architecture template of architectures to implement the problem graph in Fig. 4.2.

block matching module (BMM) and a module for performing DCT/IDCT operations (DCTM).

Possible mappings and costs of all modules are shown in Fig. 4.4. Fig. 4.4a) shows that all nodes can be implemented in software on each of the RISC processors. Those edges are simply left out. The block-matching node v_9 can be realized on resource BMM, the DCT- and IDCT-operations may be realized on module DCTM. Fig. 4.4b) shows that there is one shared bus between all components.

The user-defined constraints have been chosen as $MAXCOST = 5000$, and $MAXLATENCY = 70$ cycles. Hence, an implementation that uses all functional resources would be too expensive.

4.1.2 Results

We use the fitness function according to example 3.4.1 with a simple cost function according to example 2.3.1, i.e.,

$$f = 0.4 \frac{\sum_{\tilde{v} \in \alpha} c_h(\tilde{v})}{5000} + 0.6 \frac{\tau_L(\beta, \tau)}{70} \tag{4.1}$$

where $\tau_L(\beta, \tau)$ gives the latency of the implementation, i.e. $\tau_L(\beta, \tau) = max\{\tau(v) + delay(v, \beta) | v \in V_P\}$. The following constraints were included:

$$\begin{aligned} g_1(\alpha, \beta, \tau) &= 70 - \tau_L(\beta, \tau) \\ g_2(\alpha, \beta, \tau) &= 5000 - \sum_{\tilde{v} \in \alpha} c_h(\tilde{v}) \end{aligned}$$

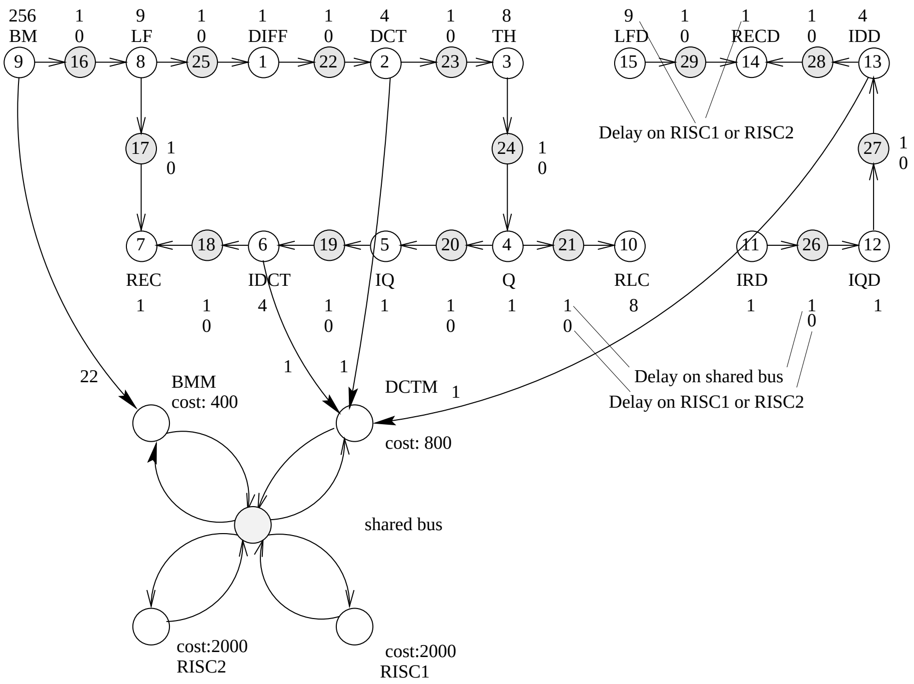

Figure 4.4: Resource graphs of the video codec in Fig. 4.3

The considered penalties were set to $p_1 = 2$ and $p_2 = 1$ (see example 3.4.1).

With the population size of $N = 30$ individuals and $k_{max} = 50$ generations, the following results have been obtained: Fig. 4.5 shows two optimal implementations found, summarized in Table 4.1.

Implementation	BMM	RISC1	RISC2	DCTM	latency	cost	fitness
1	1	1	-	-	60	2500	0.71
2	1	1	-	1	58	3300	0.76

Table 4.1: Obtained implementations by the EA for the H.261 video codec (non-pipelined implementation).

The first implementation is slower than the second, however, this implementation is cheaper. The total run-time of this design space exploring run took 13 CPU-seconds on a SUN4-20.

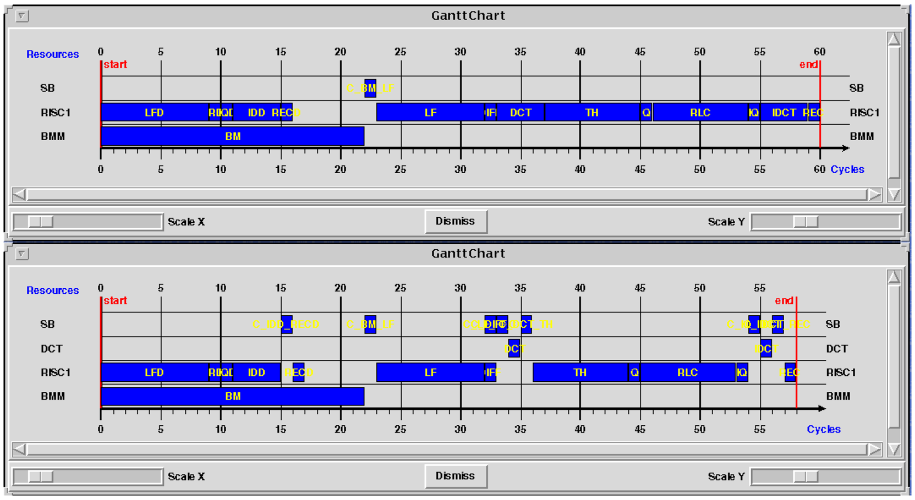

Figure 4.5: Results of applying the EA system synthesizer to the H.261 video codec. Two optimal implementations have been found.

4.2 PDE-Integrator

The following code is a VHDL specification which solves a partial differential equation of the form $y'' + 3xy' + 3y = 0$ in the interval $[x, a]$ with step-size dx and initial values $y(0) = y$, $y'(0) = u$ using the Euler method (see, e.g. [DeMicheli, 1994]).

```
ENTITY dgl IS
  PORT(x_in,y_in,u_in,dx_in,a_in: IN REAL;
       activate: IN BIT; y_out: OUT REAL);
END dgl;

ARCHITECTURE behavioral OF dgl IS BEGIN
  PROCESS (activate)
    VARIABLE x, y, u, dx, a, x1, u1, y1: REAL;
  BEGIN
    x := x_in; y := y_in; u := u_in; dx := dx_in; a := a_in;
    LOOP
      x1 := x + dx;
      u1 := u - (3 * x * u * dx) - (3 * y * dx);
      y1 := y + (u * dx);
      x := x1; u := u1; y:= y1;
      EXIT WHEN x1 > a;
    END LOOP;
    y_out <= y;
  END PROCESS;
```

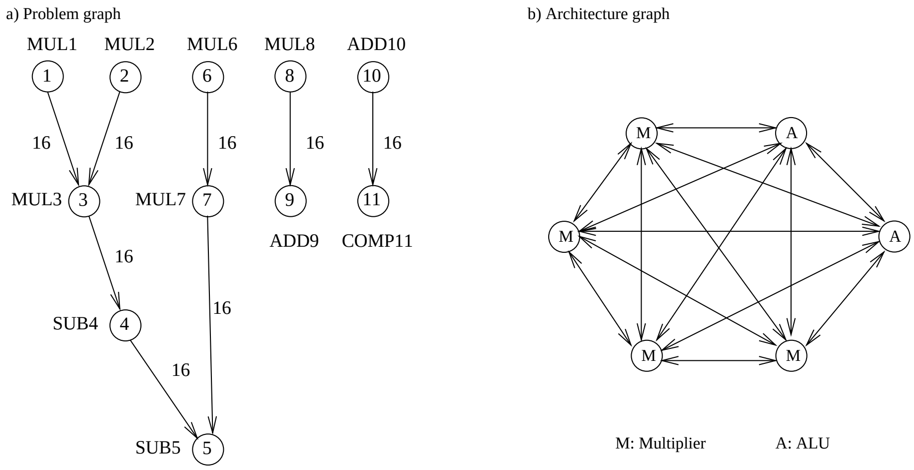

Figure 4.6: Problem graph of the PDE-Integrator a) and the architecture graph for the PDE-Integrator b)

```
END behavioral;
```

4.2.1 Problem Specification

A problem graph and an architecture graph for the above specification is shown in Fig. 4.6.

The architecture reveals that each functional module can transfer data to any other functional module using a dedicated link. A maximum of 4 multiplier modules and 2 ALUs may be allocated (see Fig. 4.6b)). The cost of each functional resource is simply set to one unit. Each multiplication may be mapped onto a multiplier node. All other nodes must be implemented on an ALU (A). The delay of any multiplication is two time units, the delay of any other operation (on the ALUs) is one unit. There are no additional costs when mapping several nodes to the same functional resource. Here, we are interested in obtaining as many non-inferior realizations as possible. The user-defined constraints are set to $MAXCOST = 6$ and $MAXLATENCY = 17$ (minimal latency with just one multiplier and one ALU).

4.2.2 Results

We use the fitness function according to example 3.4.1 with a simple cost function according to example 2.3.1, i.e.,

$$f = 0.4 \frac{\sum_{\tilde{v} \in \alpha} c_h(\tilde{v})}{6} + 0.6 \frac{\tau_L(\beta, \tau)}{17} \tag{4.2}$$

There are no penalties for pin limitations. Resource capacity limits are never reached by construction. Also neither $MAXCOST$ nor $MAXLATENCY$ may be exceeded by any non-inferior implementation.

With a population size of $N = 30$ individuals, the implementations given in table 4.2 have been obtained after $k_{max} = 20$ iterations.

Implementation	# mul	# alu	latency	cost	fitness
1	2	1	8	3	0.48
2	1	1	13	2	0.49
3	3	1	7	4	0.51
4	2	2	7	4	0.51
5	4	1	6	5	0.55
6	3	2	6	5	0.55

Table 4.2: Obtained implementations by the EA for the PDE-integrator.

As can be seen from Table 4.2, many good implementations have been found. The implementations 5,3 and 2 are shown in the Ganttchart representation in Fig. 4.7 from top to bottom.

Note, that many other solutions with equal cost and latency are possible here being just permutations of a binding. This is typical for high level-synthesis problems where many equal functional resources are allocated and hence, many permutations represent equally good solutions in fitness. The run time of the complete optimization run took 1.2 CPU-seconds on a SUN4-20.

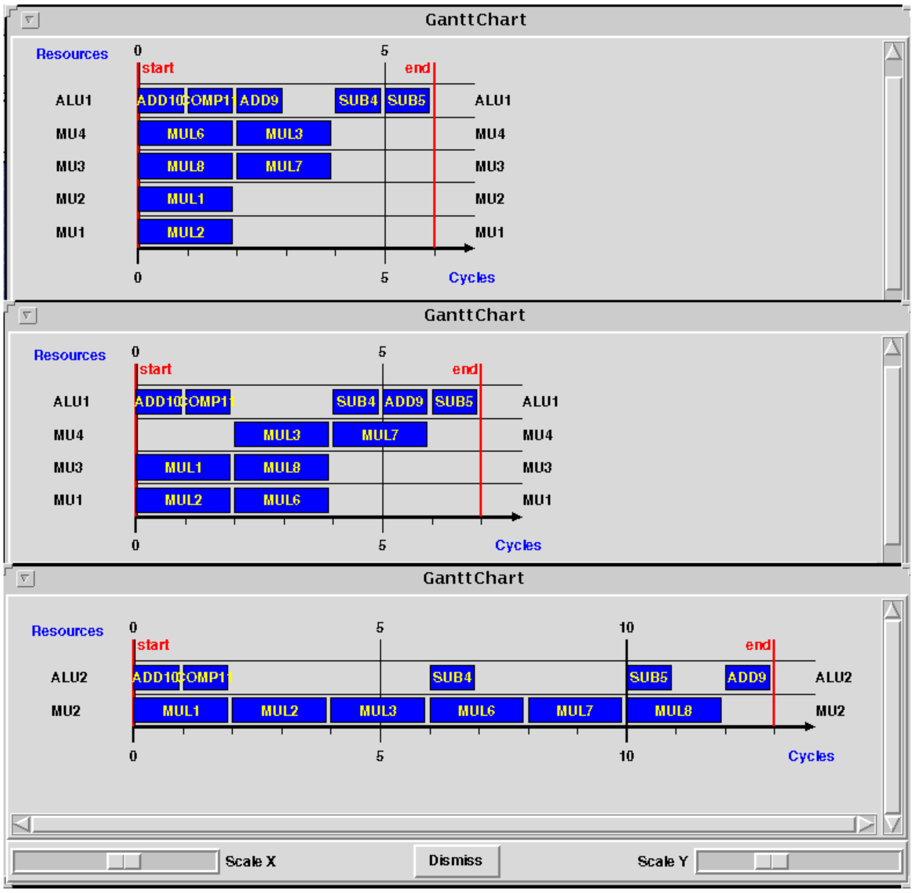

Figure 4.7: Results of applying the EA system synthesizer to the PDE with the fitness function in (4.2). The shown implementations correspond to the architectures numbered 5, 3 and 2 in Table 4.2 (top to bottom).

Chapter 5

Summary and Conclusions

An approach to system-level synthesis for dataflow-dominant hardware/software systems is presented. Contrary to existing approaches the architecture is not fixed and the mapping problem is not just understood as a simple binary hardware/software partitioning problem. We use a graph-theoretic framework to describe algorithms, sets of architectures and user-defined mapping constraints. The architectures can be single- or multiple-chip architectures. Resource sharing and limit pin numbers are considered as well as maximal cost and latency constraints. Furthermore, a capacity constraint can be formulated for each resource. Communication delays are modeled dependent on selected buses, bus-widths and transfer rates.

The main optimization loop contains an evolutionary algorithm. Based on a populations of implementations it performs a parallel search for optimal architecture selection and binding in each iteration. For each architecture and binding, a resource-constrained schedule is computed using a list-scheduler. Then, the fitness function is evaluated for each implementation. This way, the design space can be explored in one single optimization run. Different fitness functions have been implemented and tested.

We claim that the combination of a evolutionary algorithm for architecture selection with an heuristic scheduler after architecture selection and binding is suited for system-level synthesis.

- *Constructive heuristics* that determine a mapping based on global or local priority computations do not work well when many choices can be made to implement a function. This problem is already hard if the target architecture is fixed. Experiments show that they work well if the problem graph is static (no mapping dependent node delays or dependencies) and the nature of objects to be scheduled and bound is homogeneous. Clustering based approaches mostly do only consider the binding problem.

- *Iterative heuristics* like simulated annealing yield only one single solution.

Finally, evolutionary algorithms can be easily parallelized. This is a topic of future investigation.

Other results that have not presented here are the extension of the problem graph to cyclic schedules and the consideration of multiple rate constraints. The approach presented has the nice feature that other resource-constrained schedulers may be plugged into the optimizer.

Appendix A

Proof of Theorem 2.2.1

Proof: Obviously, the test of a binding for feasibility can be done in polynomial time, see Def. 2.2.3. Now, the Boolean Satisfiability problem will be polynomially reduced to the determination of a feasible binding.

At first, let us suppose that Boolean variables are associated to the mapping edges $e \in E_M$. Moreover, if $e \in \beta$, then the corresponding variable is 1 (true), if $e \notin \beta$, the variable is 0 (false). Now, it is shown how to construct a specification graph from a given set of clauses such that a feasible binding yields values for the Boolean variables which make all clauses true iff there is a solution.

Instead of using $(a \vee b \vee \cdots) = 1$ for the clauses we transform equations of the dual form $(c \wedge d \wedge \cdots) = 0$ into a specification graph. The specification graph contains two dependence graphs G_1 and G_2 only and all nodes and edges of G_1 and G_2 are allocated. The necessary components for building the specification graph are given in Fig. A.1. Any set of clauses can now be represented as follows:

1. For any variable in the set of clauses use a structure like in Fig. A.1a.
2. For any clause with n literals use $n-1$ times the structure in Fig. A.1d concatenated and linked to the variable structures constructed in step 1 by structures like Fig. A.1b.
3. Use structures as in Fig. A.1c to guarantee that all clauses are true.

It can be seen that there is a one-to-one correspondence between a solution to a given satisfiability problem and a feasible binding in the corresponding specification graph. $\square$

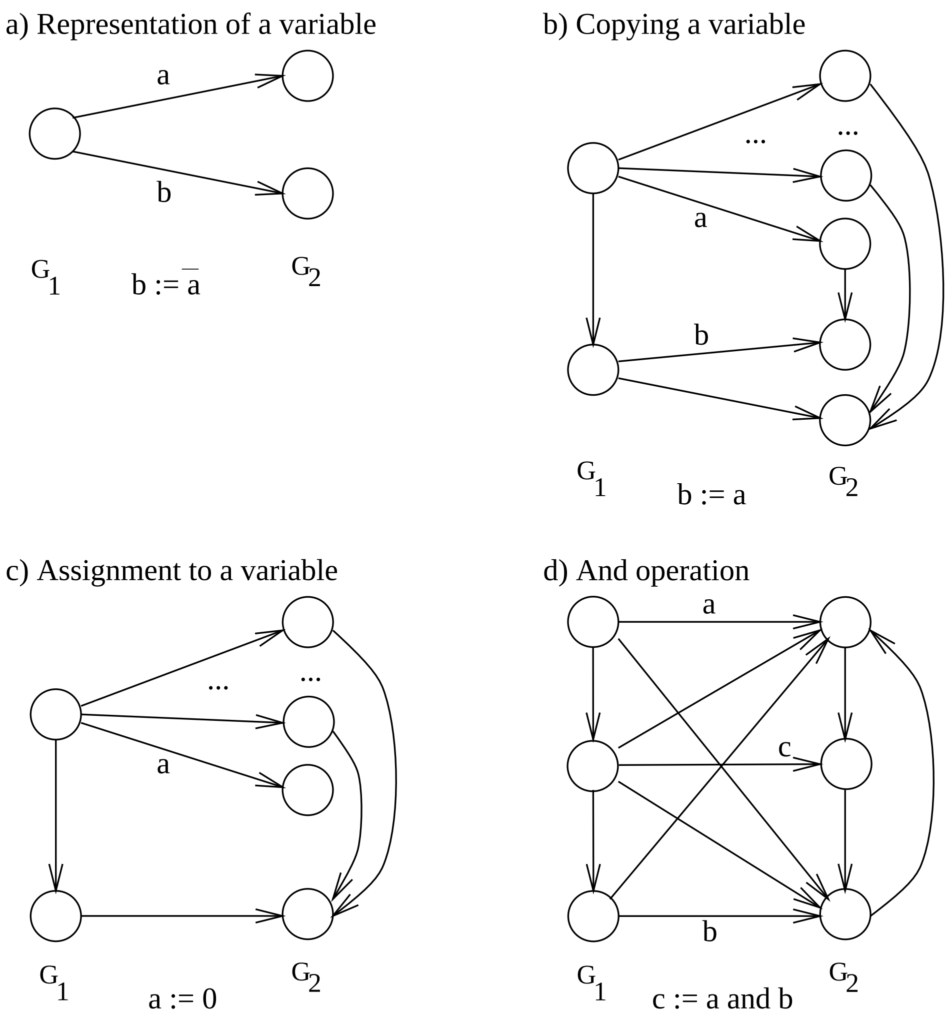

Figure A.1: Components for the transformation of Boolean expressions to a specification graph

Bibliography

[Barros and Rosenstiel, 1992] E. Barros and W. Rosenstiel. A method for hardware software partitioning. In *Proc. 1992 COMPEURO: Computer Systems and Software Engineering*, pages 580–585, The Hague, Netherlands, May 1992.

[Blickle and Thiele, 1995] Tobias Blickle and Lothar Thiele. A comparison of selection schemes used in genetic algorithms (2. edition). Technical Report 11, Computer Engineering and Communication Networks Lab (TIK), Swiss Federal Institute of Technology (ETH) Zurich, Gloriastrasse 35, CH-8092 Zurich, Dec 1995.

[Camposano and Brayton, 1987] R. Camposano and R. K. Brayton. Partitioning before logic synthesis. In *Proc. ICCAD*, 1987.

[DeMicheli, 1994] G. DeMicheli. *Synthesis and Optimization of Digital Circuits.* Mc Graw Hill, New York, 1994.

[Ernst *et al.*, 1994] R. Ernst, J. Henkel, and T. Benner. Hardware-software cosynthesis for microcontrollers. *IEEE Design & Test of Computers*, pages 64–75, December 1994.

[Fox and McMahon, 1991] B. R. Fox and M. B. McMahon. Genetic operators for sequencing problems. In Gregory J. E. Rawlins, editor, *Foundations of Genetic Algorithms*, pages 284–300. Morgan Kaufmann Publishers, 1991.

[Gupta and Micheli, 1992] R. Gupta and G. De Micheli. System-level synthesis using re-programmable components. In *Proc. of the European Conference on Design Automation (EDAC)*, pages 2–7, 1992.

[Hardt and Camposano, 1995] W. Hardt and R. Camposano. Specification analysis for hw/sw-partitioning. In *Proc. GI/ITG Workshop Application of formal Methods during the Design of Hardware Systems*, pages 1–10, Passau, Germany, March 1995.

[Ismail *et al.*, 1994] T. B. Ismail, K. O'Brien, and A. A. Jerraya. Interactive system-level partitioning with PARTIF. In *Proc. of the European Conference on Design Automation (EDAC)*, pages 464–473, 1994.

[Kalavade and Lee, 1994] A. Kalavade and E. A. Lee. A global criticality/local phase driven algorithm for the constrained hardware/software partitioning problem. In *Proc. of CODES/CASHE'94, Third Intl. Workshop on Hardware/Software Codesign*, Grenoble, France, September 1994.

[Küçükçakar and Parker, 1995] K. Küçükçakar and A. C. Parker. A methodology and design tools to support system-level VLSI design. *IEEE Trans. on VLSI systems*, 3(3):355–369, September 1995.

[Lagnese and Thomas, 1991] E. D. Lagnese and D. E. Thomas. Architectural partitioning for system level synthesis of integrated circuits. *IEEE Trans. on CAD*, 10(7):847–860, July 1991.

[McFarland, 1986] M. C. McFarland. Using bottom-up design techniques in the synthesis of hardware from abstract behavioral descriptions. In *Proc. 23rd Design Automation Conference*, pages 474–480, June 1986.

[Syswerda and Palmucci, 1991] Gilbert Syswerda and Jeff Palmucci. The application of genetic algorithms to resource scheduling. In R.K. Belew and L.B. Booker, editors, *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 502–508, San Mateo, CA, 1991. Morgan Kaufmann Publishers.

[Syswerda, 1989] Gilbert Syswerda. Uniform crossover in genetic algorithms. In J. David Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 2–9, San Mateo, CA, 1989. Morgan Kaufmann Publishers.

[Vahid and Gajski, 1992] F. Vahid and D. Gajski. Specification partitioning for system design. In *Proc. 29th Design Automation Conference*, pages 219–224, Anaheim, CA, June 1992.