Design Automation for Embedded Systems, 3, 23–58 (1998)


# System-Level Synthesis Using Evolutionary Algorithms

TOBIAS BLICKLE teich@tik.ee.ethz.ch
*Computer Engineering and Communication Networks Lab (TIK), Swiss Federal Institute of Technology (ETH), Gloriastrasse 35, 8092 Zurich, Switzerland*

JÜRGEN TEICH
*Computer Engineering and Communication Networks Lab (TIK), Swiss Federal Institute of Technology (ETH), Gloriastrasse 35, 8092 Zurich, Switzerland*

LOTHAR THIELE
*Computer Engineering and Communication Networks Lab (TIK), Swiss Federal Institute of Technology (ETH), Gloriastrasse 35, 8092 Zurich, Switzerland*

**Editor:** R. Gupta

*Received February 6, 1996; Revised October 1, 1996*

**Abstract.** In this paper, we consider *system-level synthesis* as the problem of optimally mapping a task-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, busses and memories, (2) the *mapping* of the specification 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, do not consider communication (tasks and resources), or 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. The application and adaptation of an Evolutionary Algorithm to solve the tasks of optimization and design space exploration is described.

**Keywords:** System-synthesis, hardware/software partitioning, design space exploration, evolutionary algorithms.

## 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 automated synthesis 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 tasks, procedures, or processes onto a structural

specification with structural objects being general or special purpose processors, ASICs, busses and memories.

When searching for 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 a specification based on a data flow graph at the task-level 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 specification 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 of

- *providing a new system-level specification/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)–(2),
- *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., communicating sequential processes [4], [24], C-code [13] and extensions thereof [9] or finite-state-machine based specifications, e.g., [15]. In the mapping phase, (static) scheduling of tasks and communications cannot be done due to non-

determinism of execution times. Here, estimation mainly depends on profiling and simulation.

- *Data flow-dominant specification*: Methodologies in this class consider data flow specifications, e.g., data flow graphs [16], precedence- and taskgraphs or parallel languages, e.g., UNITY [1]. The approach in [12] belongs to the same category, however, with extensions to allow for operations with non-deterministic execution times.

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

- *Dedicated control & data path in VLSI*: [19], [18], [25] 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., [17]. All functionality is mapped onto a dedicated multi-chip architecture including busses.

- *Hardware/Software Architectures*: In the realm of hardware/software architectures, most approaches consider the target architecture to be fixed, e.g., [16] (one processor and custom hardware communicating via memory-mapped I/O), [9] (one RISC processor and one or more given custom blocks and predefined HW-modules that communicate by memory coupling using a single CSP type protocol), or [12] (one programmable component and multiple hardware modules communicating with each other using one system bus, the processor being the master). In [6], also the allocation of components is considered as a task of the mapping process.

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., [5], [19], [18], [25]. There, the goal is in most cases to meet chip capacity and time constraints. Most of these approaches present a clustering-based approach. In [17], multichip modules are considered, however no programmable components are allowed. In some clustering based approaches like [1], 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. In [6], e.g., a tool based on enumeration is used to find Pareto-implementations. 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 if the number of objects is rather small.

- *heuristics*: In [12], Gupta presents 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. [9] present an approach that starts with an initial partition in software and

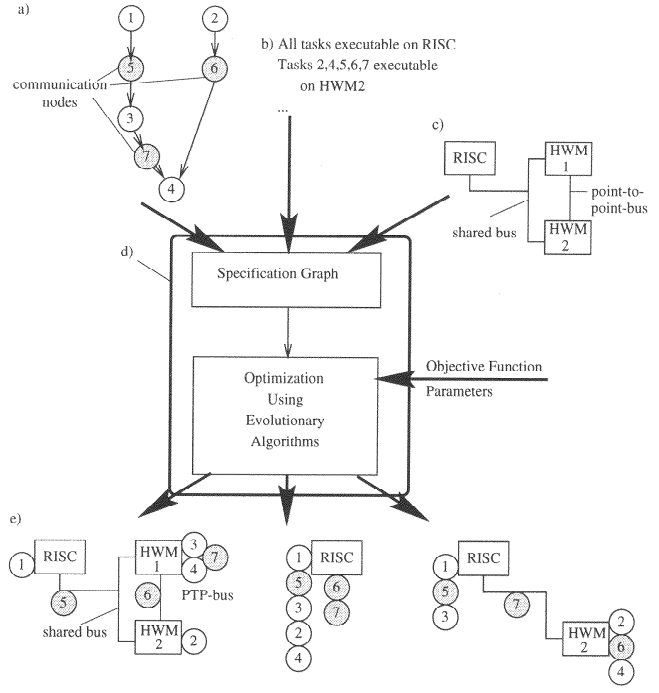


*Figure 1.* Overview of the proposed optimization methodology: a) data flow graph based specification, b) mapping constraints, c) architecture template, d) optimization methodology, e) implementations.

moves functionality to hardware until timing constraints are met. They use simulated annealing in an inner optimization loop and, adapt estimated parameters in an outer loop. Recently, the approach described in [6] has been improved from formerly using global enumeration techniques to applying evolutionary strategies for design space exploration [7]. However, the approach does not treat the problems of modeling and mapping of communications (zero-delay shared-memory model assumed).

### *1.3. Overview*

Figure 1 gives an overview of our optimization methodology, see also [23].

The specification consists of a data flow graph similar problem graph (see Fig. 1a), an architecture template (see Fig. 1c), user-defined mapping constraints (see Fig. 1b) and an optimization procedure (see Fig. 1d) using an Evolutionary Algorithm (EA). An Evolutionary Algorithm works on *populations of individuals* $J_i$, $i = 1, \ldots, N$ 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 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, Pareto-point (yes/no), 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.* Figure 1e shows some individuals out of a population $P_k$ of implementations. Shown is the binding of nodes of the problem graph in Fig. 1a including the binding of functional nodes to functional resources (RISC, hardware modules HWM1 and HWM2) and the binding of communication nodes (shaded nodes) to bus resources (shared bus, point-to-point bus). The scheduling of nodes is not shown. Note that each individual of a population may code a different architecture.

We will introduce a fitness function called *Pareto-ranking* for performing design space exploration in a single optimization run. The Pareto-optimal solutions (architecture, binding, schedule) are output and can be displayed graphically in the form of a Ganttchart.

This is an overview of the contents of forthcoming sections: First, the new specification model including the problem graph and the architecture specification 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 coding of implementations, the definition of useful fitness functions for evaluating the quality of an implementation, and the description of the genetic operators are outlined in Section 3. The realistic example of a video codec is used in Section 4 as a case study to explain our optimization methodology and to show its performance.

The complete methodology is part of the CodeSign framework at ETH Zurich. In Section 5, we explain how the proposed methodology is embedded in the framework and propose directions for further work.

## 2. Modeling Algorithms and Architectures

### *2.1. Specification*

The specification model consists of three main components:

- The problem that should be mapped onto 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 tasks 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 may be assigned to either nodes or edges of $G_S$.

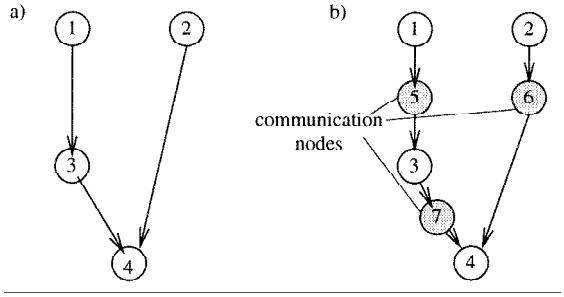


*Figure 2.* A data flow graph (a) and the corresponding problem graph (b).

- Associated with 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 the problem as well as architectures on different levels of abstraction.

*Definition 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 specification 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.* One can think of a problem graph as the graph obtained from a data flow graph by inserting communication nodes into some edges of the data flow graph (see Fig. 2). These nodes will be drawn shaded throughout the following examples.

Now, the architecture including functional resources and busses 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 an adder, a multiplier, a RISC processor, a dedicated processor, or an 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 3.* Figure 3a) 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). Figure 3b) shows the corresponding architecture graph $G_A$.

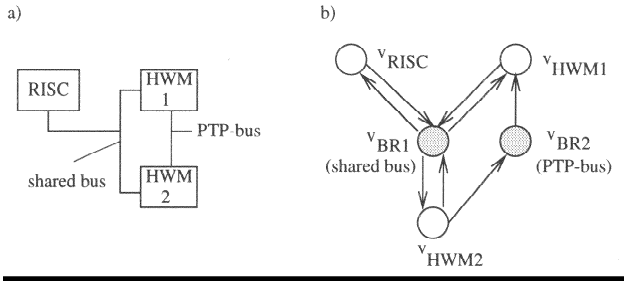


*Figure 3.* An example of an architecture (a), and the corresponding architecture graph $G_A$ (b).

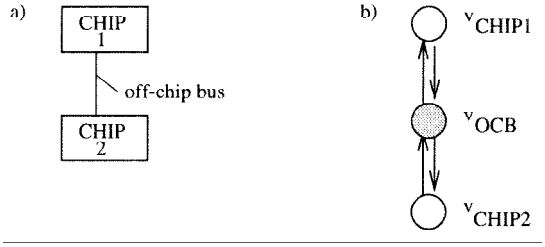


*Figure 4.* An example of a multi-chip architecture (a) and the corresponding chip graph $G_C$ (b).

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 4.* Figure 4a) 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. Figure 4b) shows the corresponding chip graph $G_C$.

Note that in the above example, 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.

*Definition 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 problem description (problem graph), architecture description (architecture graph) and system description (chip graph). The edges

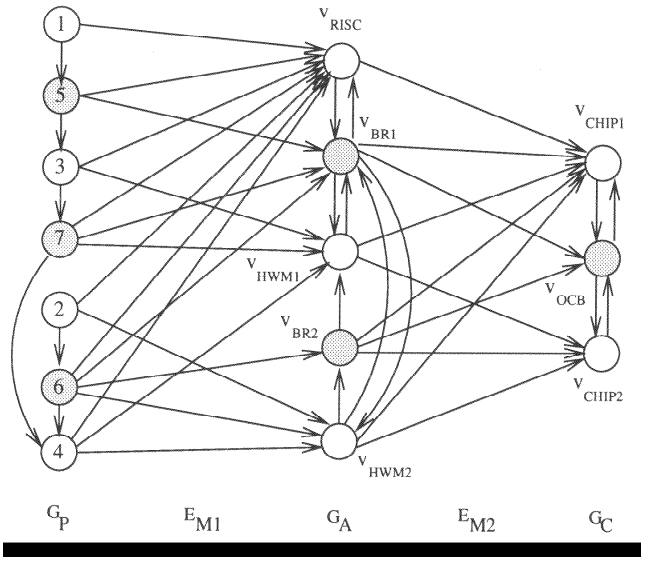


*Figure 5.* An example of a specification graph $G_S$.

represent user-defined mapping constraints in the form of a relation: "can be implemented by".

*Example 5.* Figure 5 shows an example of a specification graph using the problem graph of Example 2 (left), the architecture graph of Example 3 (middle) and the chip graph of Example 4 (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 definition, *allocation*, *binding* and *scheduling* will formally be defined in the next subsection.

*Definition 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 to 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 introduced so far, all nodes and edges of the problem graph contained in $G_S$ were necessary, i.e., activated. The determination of an implenentation 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 implementation will be formally defined as well as the main tasks of synthesis, namely *allocation*, *binding*, and *scheduling*.

*Definition 4.* [Allocation] An *allocation* $\alpha$ of a specification graph is the subset of all activated nodes and edges of the dependence graphs, i.e.,

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

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

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

*Definition 5.* [Binding] A *binding* $\beta$ is the subset of all activated mapping edges, i.e.,

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

*Definition 6.* [Feasible Binding] Given a specification $G_S$ and an allocation $\alpha$. A *feasible binding* $\beta$ 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.,

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

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.,

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

- 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.,

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

It is useful to determine the set of feasible allocations and feasible bindings in order to restrict the search space of the optimization procedure.

*Definition 7.* [Feasible Allocation] A *feasible allocation* $\alpha$ is an allocation that allows at least one feasible binding $\beta$.

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

*Proof.* See appendix. ■

Finally, it is necessary to define a *schedule*. Let $delay(v, \beta)$ denote the execution time of the 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 $\beta$. In other words, the execution time of an operation depends on the resource where it is going to be executed.

*Definition 8.* [Schedule] Given a specification $G_S$ containing a problem graph $G_1 = G_P$, a feasible binding $\beta$, 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 $\tau\colon V_P \mapsto \mathbb{Z}^+$ 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 execution time of a communication node denotes the number of time units necessary to transfer the associated data on the bus resource it is bound to. 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 6.* 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. 6). The 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

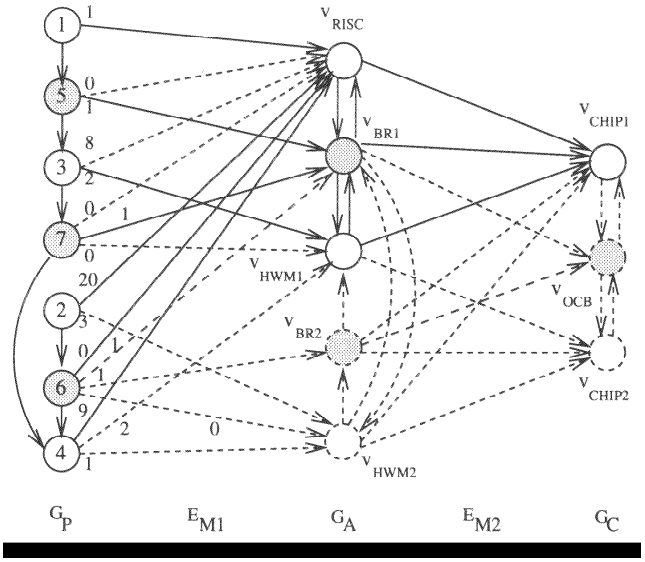


*Figure 6.* An example of an implementation from the specification given in Fig. 5.

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 9.* [Implementation] Given a specification graph $G_S$, a *(valid) implementation* is a triple $(\alpha, \beta, \tau)$ where $\alpha$ is a feasible allocation, $\beta$ is a feasible binding, and $\tau$ is a schedule.

*Example 7.* Figure 6 shows an implementation of the specification depicted in Fig. 5. The nodes and edges which are not allocated are shown dotted as well as the edges $e \in E_M$ that are not activated. The allocation of nodes is $\alpha_V = V_P \cup \{v_{RISC}, v_{HWM1}, v_{BR1}, v_{CHIP1}\}$ and the binding is $\beta = \{(v_1, v_{RISC}), (v_2, v_{RISC}), (v_3, v_{HWM1}), (v_4, v_{RISC}), (v_5, v_{BR1}), (v_6, v_{RISC}), (v_7, v_{BR1}), (v_{RISC}, v_{CHIP1}), (v_{BR1}, v_{CHIP1}), (v_{HWM1}, v_{CHIP1})\}$. 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 previously, the task of system synthesis can be formulated as an optimization problem.

*Definition 10.* [System Synthesis] The task of *system synthesis* is the following optimization problem:

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

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

The constraints on $\alpha$, $\beta$ and $\tau$ define the set of valid implementations. Additionally, there are functions $g_i$, $i = 1, \ldots, q$, that together with the objective function $f$ describe the optimization goal.

*Example 8.* Let the specification graph $G_S$ consist of a problem graph $G_P$ and an architecture graph $G_A$ only. 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 a 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., if $\tilde{v} \in \alpha$. 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 8 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 specification consists only of a problem graph $G_P$ and an architecture graph $G_A$.

*Definition 11.* [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 12.* [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.

*Definition 13.* [Types] The node set $V_P$ is partitioned into *types* $T_i \in \mathcal{T}$, i.e., each node belongs exactly to one type from among the set of types.

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

$$c_t(T_i, \tilde{v}) = max\{c_a(e) \mid e = (v, \tilde{v}) \in \beta \wedge v \in T_i\}$$

With these definitions, a more realistic modeling of the cost of an implementation is possible. In particular, the cost $c(\alpha, \beta)$ of an implementation may now be computed 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.,

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

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 $MAXCOST_v$, i.e., limits the maximum program size or chip area:

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

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

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

*Example 9.* Figure 7 demonstrates the use of function types to model costs of shared resources. One function type is defined in Fig. 7b). $T_1 = \{v_1, v_3\}$, 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 ($c_h(v_{RISC}, \beta_1)$) is carried out for the particular binding $\beta_1$ given by the implementation in Fig. 7c).

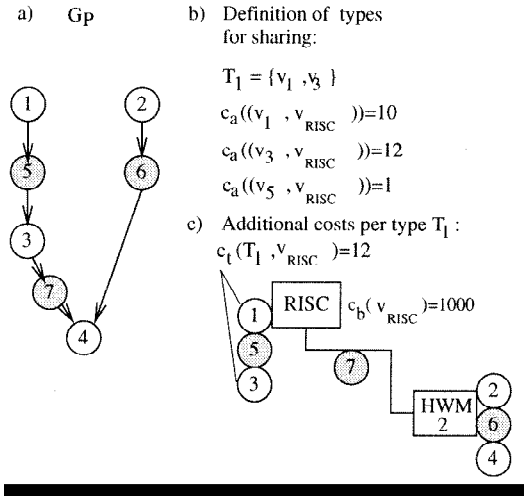


*Figure 7.* Modeling of resource sharing (program memory or hardware modules) by definition of types for sharing.

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 additional costs 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 becomes

$$\begin{aligned} c_h(v_{RISC}, \beta_1) &= c_b(v_{RISC}) + \sum_{T \in \{T_1, \{v_5\}\}} c_t(T, v_{RISC}) \\ &= c_b(v_{RISC}) + c_t(T_1, v_{risc}) + c_t(\{v_5\}, v_{RISC}) \\ &= 1000 + 12 + 1 = 1013 \end{aligned}$$

### *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 4). 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 4) is only a bundle of wires. The possible sequentialization of operations or communications is done by 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, we explain how to model the number of pins in order to specify an upper bound *PINLIMIT* of pins 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 nodes $\tilde{v} \in V_A$ that imply a communication of chip $\bar{v} \in V_C$ to the outside, i.e.,

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

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

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

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 10.* Consider the specification of Example 5. 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. 8a) with binding $\beta_a$. 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 nodes that imply communications 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. 8a) violates the constraint as $g = 20 - pin(v_{CHIP1}) = -14 < 0$.

On the other hand, the implementation in Fig. 8b) with binding $\beta_b$ does satisfy this constraint. As $V_B(v_{CHIP1}, \beta_b) = \{v_{BR2}\}$, the pins of CHIP1 calculate to

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

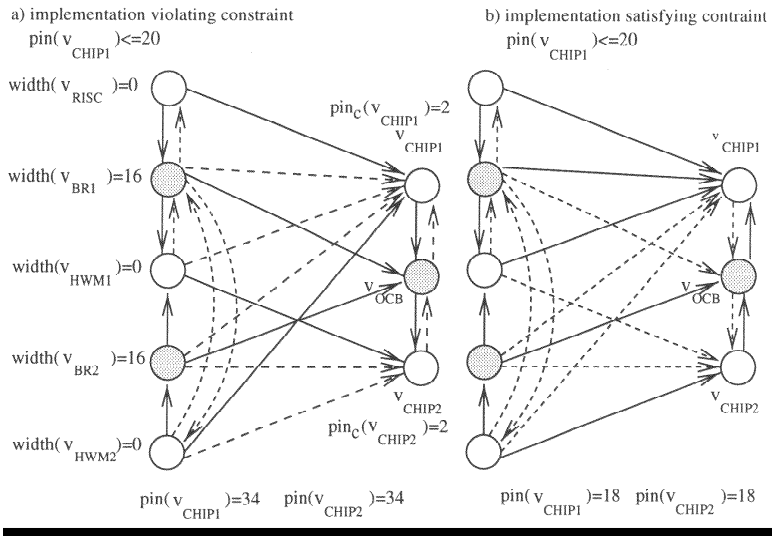


*Figure 8.* Two implementations of the specification of Fig. 5: a) violating the pinlimit constraint $pin(v_{CHIP1}) \leq 20$ and b) satisfying the pinlimit constraint $pin(v_{CHIP1}) \leq 20$.

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

## 3. Optimization Method

In this section, the application of an Evolutionary Algorithm (EA) is described to solve the problem of system synthesis as specified by Definition 10. The Evolutionary Algorithm is responsible for the determination of allocations and bindings. The schedule for each allocation and binding is then computed by a scheduling heuristic. The key to this division of work (see Fig. 9) is motivated by the fact that the search space for these tasks is large and discrete and already the determination of a feasible binding is NP-complete (see Theorem 1). Hence, for reasonable problem sizes exact methods are intractable. As there are good heuristics available for solving the scheduling problem for a given allocation and binding, it is not necessary to load the EA with this task.

In the remainder of this section, the basic principles of Evolutionary Algorithms are explained, then the coding of implementations as individuals is described. After that, a fitness function called *Pareto-ranking* that is particularly useful for design space exploration 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)

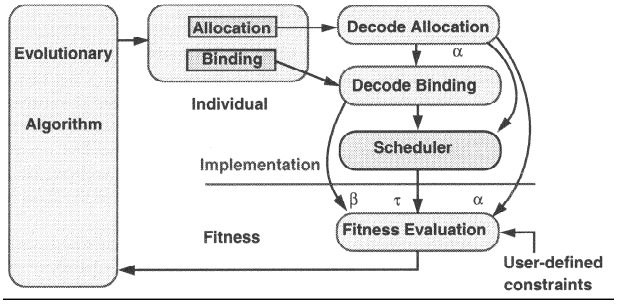


*Figure 9.* The decoding of an individual to an implementation.

of the optimization problem simultaneously sample the search space. This *population* $P = \{J_1, J_2, \ldots, J_N\}$ 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 of survival. Selection thereby focuses on the search on promising regions in the search space. The quality of an individual is measured by a fitness function $F\colon \mathbf{J} \mapsto \mathbf{R}$. Various selection methods exist (see e.g., [3]).

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 for the task of system synthesis, 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 have 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 $D$ subgraphs $G_i$, $1 \leq i \leq D$ 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()*).

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 $1 \leq i < D$.
**Output:** The allocation $\alpha$ and the binding $\beta$ if both are feasible,
({}, {}) if no feasible binding is represented by the individual $J$

decode($J$):
  $\alpha \leftarrow V_1 \cup E_1$
  $\beta \leftarrow \{\}$
  **for** $i \leftarrow 1$ **to** D-1 **do**
    $\bar{\alpha} \leftarrow$ node_allocation($alloc_i(J)$, $L_{Ri}(J)$)
    $\bar{\beta} \leftarrow$ binding($L_{Bi}(J)$, $L_{Oi}(J)$, $\bar{\alpha}$)
    **if** $\bar{\beta} = \{\}$
      **return** ({}, {})
    **endif**
    $\beta \leftarrow \beta \cup \bar{\beta}$
    $\alpha \leftarrow \alpha \cup$ update_allocation($\bar{\alpha}$, $\bar{\beta}$)
  **od**
  **return** ($\alpha$, $\beta$)

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.

### *3.2.1. The Function node_allocation()*

The allocation of nodes is directly encoded in the chromosome, i.e., for each level $i$ there exists a vector $alloc_i$ that contains a vector element for each node $v \in V_{i+1}$ 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 5 (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., $V_{Bi} = \{v \in V_i \mid \forall \tilde{v} \in V_{i+1} : (v, \tilde{v}) \in E_{Mi} \wedge a(\tilde{v}) = 0\}$. 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 allocations for the first mapping level of Example 5 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 $\alpha$

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**
  $V_{Bi} \leftarrow$ not_bindable_nodes($\alpha$)
  $\tilde{v}_r \leftarrow$ first($L_{Ri}$)
  **while** ($V_{Bi} \neq \{\}$) **do**
    **if** ($V_{Bi} \neq$ not_bindable_nodes($\alpha \cup \{\tilde{v}_r\}$)
      $\alpha \leftarrow \alpha \cup \{\tilde{v}_r\}$
      $V_{Bi} \leftarrow$ not_bindable_nodes($\alpha$)
    **endif**
    $\tilde{v}_r \leftarrow$ next($L_{Ri}$)
  **od**
  **return** $\alpha$

The order in which additional 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 introduction of a *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

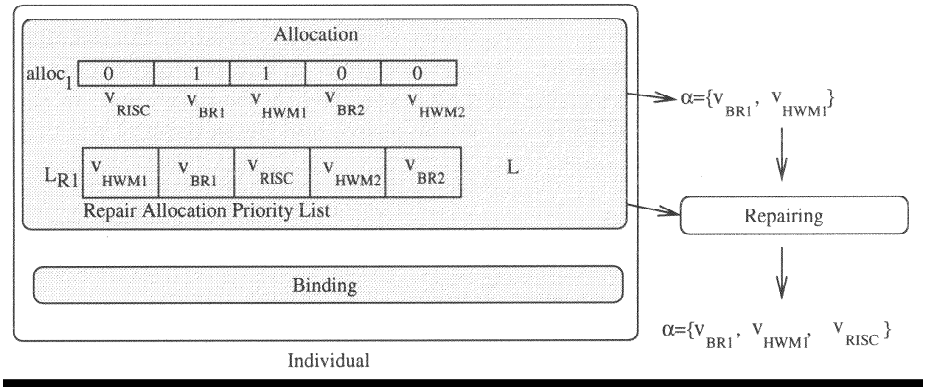


*Figure 10.* The decoding of an allocation.

the order the nodes will 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 11.* Consider the specification of Example 5. In Fig. 10, the allocation information for the first mapping level as stored in the individual is shown (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 $v_{BR1}$ belong already to the allocation. The allocation of $v_{RISC}$ resolves the conflict for both $v_1$ or $v_2$, hence it is allocated. 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 for each allocated node $v \in V_i$ is obtained by activating exactly one of its outgoing edges $e \in E_{Mi}$. 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 sucessor nodes $\tilde{v} \in V_{i+1}$ of $v$, i.e., $(v, \tilde{v}) \in E_{Mi}$. This list is seen as a priority list and the first node $\tilde{v}_k$ with $e_k = (v, \tilde{v}_k)$ 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 6). Details are given

in the two algorithms. Note that it is possible that no feasible binding is specified by the individual. In this case, $\beta$ 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 $\alpha$ of an individual $J$.
**Output:** The binding $\beta$, or {} if no feasible binding was decoded.

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

*Example 12.* Figure 11 shows an example of a binding as it is coded in the individual for the first mapping level of the specification from Example 5 (Fig. 5). 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 the RISC and so on. As the allocation from Example 11 does not contain HWM2 and BR2, this node is finally bound to RISC.

Algorithm 4: (Check for Valid Binding)

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

```
is_valid_binding(e = (v, ṽ), β, α):
    forall ê = (w, v) ∈ E_i ∩ α do
        forall w̃: (w, w̃) ∈ β do
```

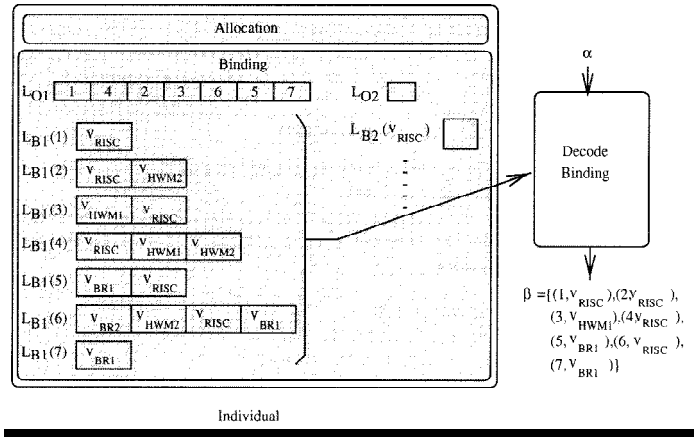


*Figure 11.* An example of the coding of a binding and the resulting binding using the allocation of Fig. 10. The obtained architecture is the one depicted in Fig. 6 of Example 7.

```
            if (ṽ ≠ w̃ ∧ (w̃, ṽ) ∉ E_{i+1})
                return false
            endif
        od
    od
    forall ê = (v, w) ∈ E_i ∩ α do
        forall w̃: (w, w̃) ∈ β do
            if (ṽ ≠ w̃ ∧ (ṽ, w̃) ∉ E_{i+1})
                return false
            endif
        od
    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 allocation.

Algorithm 5: (Update Allocation)

**Input:** The current allocation $\alpha$ and binding $\beta$ of the mapping level

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

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

*Example 13.* Consider the allocation and binding of Example 11 and 12, 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 $(v_{HWM1}, v_{BR1}) \in E_A$ as node $v_3$ is bound to resource $v_{HWM1}$ and node $v_7$ is bound to resource $v_{BR1}$.

These algorithms describe in detail the transformation of an individual into an allocation $\alpha$ and a binding $\beta$. Now, both $\alpha$ and $\beta$ are used as input to the scheduler to obtain the complete implementation.

### *3.3. Scheduling*

In this section, the tasks of a scheduler are briefly described and a new simple heuristic to obtain iterative (pipelined) schedules (for acyclic graphs) is presented, because these arise typically in data flow-dominant systems. As the allocation and binding is fixed when the scheduler is invoked, only the task of latency minimization under resource constraints is discussed.

A scheduler assigns to each node $v \in V_P$ of the problem graph $G_P$ a start time $\tau(v)$ such that the overall execution time (latency) is minimized. Additionally, all precedence constraints and resource constraints must be fulfilled. The scheduler introduced here is based on a list scheduler, see e.g., [8]. List scheduling denotes a class of constructive scheduling heuristics that successively plan those tasks that have no unplanned predecessors and thereby consider resource constraints. List scheduling is able to schedule a non-cyclic task graph and generates non-iterative schedules. However, in data flow-dominant applications iterative (periodic) schedules are of interest. This is due to the fact that usually the operations specified by the problem graph are executed repeatedly. Iterative schedules are characterized by the fact that the schedule repeats after a certain time (called period $P$).

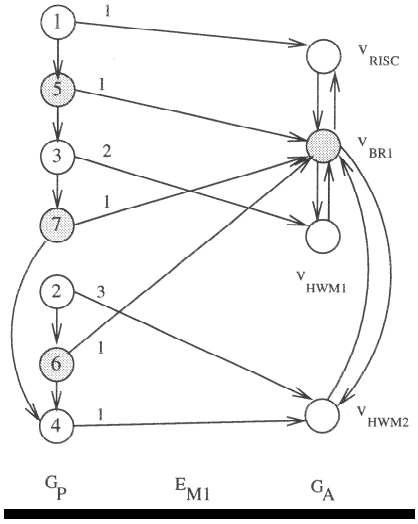


*Figure 12.* An allocation and binding of the specification graph of Example 5.

*Example 14.* Consider the specification graph of Example 5 with the allocation and binding shown in Fig. 12. An iterative schedule is given in Fig. 13. The period of the schedule is $P = 4$ as it repeats every four time units. However, the latency is $L = 8$: Calculation of one data set (drawn as unfilled rectangles) starts at time 0 with operation $v_1$ and $v_2$ and finishes with operation $v_4$ at time step 8. Note that the operations $v_4$ starting at time step 3 and $v_7$ starting at time step 0 executed in the first period belong to the previous data set.

The basic idea of our iterative scheduling heuristic is to remove edges (dependencies) from the problem graph such that the non-iterative list scheduler can be used.

The detailed algorithm works as follows: First, a non-iterative schedule is obtained using a list scheduler. In this initial try, all operations belong to the same iteration interval $\varphi(v) = 1, \forall v \in V_P$. Then, all operations that end later than the minimum period $P_{min}$ are moved into the next iteration interval. "Moving" an operation $v$ corresponds to the deletion of all precedence constraints on that operation, i.e., to remove all edges $e \in \{e = (u, v) \in G_E \mid \tau(v) + delay(v, \beta) > P_{min}\}$. The new problem graph (with some edges removed) is scheduled again using a list scheduler. The latency of this schedule is then equivalent to the period $P$ of the iterative schedule. This procedure repeats until the current period is greater than the previous one. A parameter of the algorithm limits the maximum number of iteration intervals $\varphi_{max}$ to be used.

$P_{min}$ may be computed as the maximum occupation of any resource. As allocation and binding is already fixed, this value computes to

$$P_{min} = \max_{\tilde{v} \in \alpha} \sum_{v \in V_P \wedge (v, \tilde{v}) \in \beta} delay(v, \beta) \tag{1}$$

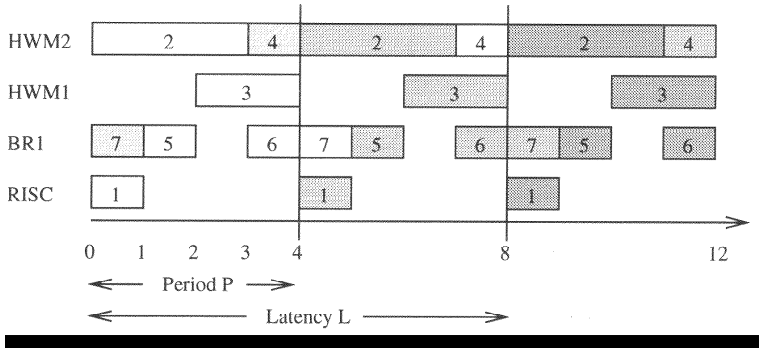


*Figure 13.* An iterative schedule for the allocation and binding of Fig. 12.

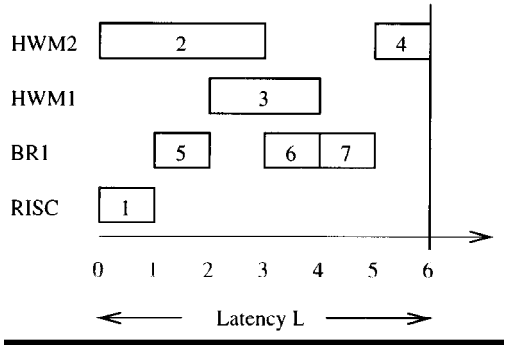


*Figure 14.* Non-iterative schedule of the allocation and binding of Fig. 12 obtained with list scheduling.

Details of the scheduling algorithm can be found in [2].

*Example 15.* Consider again the allocation and binding of Fig. 12. The minimum period calculates to $P_{min} = max\{1, 1+1+1, 2, 3+1\} = 4$. A non-iterative schedule is given in Fig. 14. According to the proposed scheduling heuristic, operations $v_4$ and $v_7$ are moved into the next iteration, i.e., the edges $(v_3, v_7)$ and $(v_6, v_4)$ are removed from the problem graph $G_P$ in Fig. 12. Note that the edge $(v_7, v_4)$ is not removed, as both operations belong to the same iteration interval. If the modified problem graph is scheduled again, the schedule of Fig. 13 is obtained. As the period of this schedule is optimal, the calculation is finished.

### *3.4. Fitness Function and Constraint-Handling*

In the preceding paragraph the coding of an individual was presented. The quality of an implementation and the optimization goal for the Evolutionary Algorithm will be discussed by describing the calculation of the fitness function.

According to Definition 10, the task of system-synthesis is described as an optimization problem. As the particular optimization goal depends on the specific design problem, the objective function is designed individually for each problem. However, the restriction of a feasible allocation and binding still needs to be considered. The repairing heuristics introduced in the previous section already capture a large part of infeasible allocations and bindings. The remaining infeasible implementations are handled by penalty terms. Hence, the fitness function $F$ to be minimized can be given as

$$F(J) = \begin{cases} x_a(J)p_a + x_b(J)p_b & : \; x_a(j) = 1 \vee x_b(J) = 1 \\ F'(J) & : \quad else \end{cases} \tag{2}$$

The $p$ values are the penalty terms, i.e., $p_a$ is the penalty term for an infeasible allocation and $p_b$ for a infeasible binding. The boolean variables $x$ denote whether the corresponding constraint is violated or not, e.g., $x_a(J) = 0$ if the allocation is feasible and $x_a(J) = 1$ if the allocation is infeasible. The values for the penalty terms $p_a$ and $p_b$ should be chosen such that any infeasible allocation or binding has a larger fitness value than any feasible implementation. The modified fitness function $F'(J)$ has to reflect the additional constraints $g_i$. For this purpose numerous methods for constraint-handling can be applied (see, for example, [20]). Here, only the case of *Pareto-optimization* will be considered. Examples of other fitness functions and design goals are discussed in [2].

Usually, in system-synthesis many different criteria have to be optimized, for example, cost of an implementation, data-throughput, power consumption, maintainability. The concept of Pareto-optimality gives a measure of concurrently comparing implementations to several criteria.

*Definition 14.* [Pareto-points] A point $J_i$ is *dominated* by point $J_k$ if point $J_k$ is better than or equally good as $J_i$ in each criteria, denoted $J_i \succ J_k$. A point (implementation) is said to be Pareto-optimal (or Pareto-point) if it is not dominated by any other point.

*Example 16.* Consider the trade-off between cost $c$ and speed (period $P$) of an implementation. Fig. 15 shows a typical design-space exploration. The implementations are marked by crosses. Pareto-points are located at the origins of the horizontal and vertical lines that separate regions of dominated design points.

An Evolutionary Algorithm allows to determine the Pareto-set in a single optimization run as the EA operates on populations. The necessary preconditions (fitness function and selection scheme) will be discussed next.

In [11], a Pareto-ranking scheme is proposed for multi-modal optimization. Thereby, the fitness of an individual $J$ is determined by the number of individuals of the population that are better in at least one criterion than individual $J$ (Definition 14), i.e.,

$$F'(J) = \sum_{i=1,\ldots,N, J \neq J_i} \begin{cases} 1 & : \; J_i \prec J \\ 0 & : \quad else \end{cases} \tag{3}$$

All Pareto-points in the population have an (optimal) fitness of zero. Note that the fitness of an individual depends on the population.

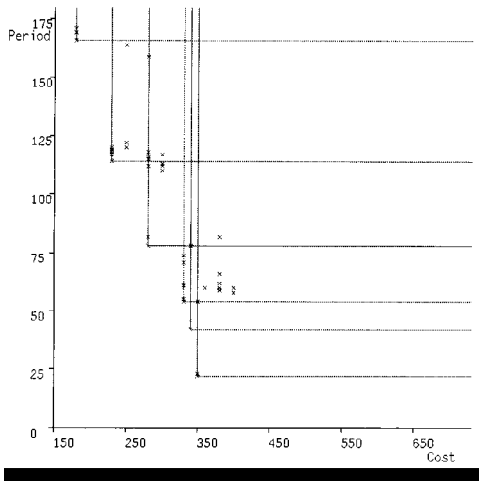


*Figure 15.* Design space with Pareto-points (see also Table 3). Pareto-points are located at the origins of the horizontal and vertical lines.

*Example 17.* As an example for Pareto-optimization, consider the case of two-dimensional optimization with the criteria cost $c(\alpha, \beta)$ and period $P(\tau)$. We obtain the fitness function:

$$F'(J) = \sum_{i=1,\ldots,N, J \neq J_i} \begin{cases} 1 & : \ (c(J) > c(J_i) \wedge P(J) \geq P(J_i)) \\ 1 & : \ (c(J) \geq c(J_i) \wedge P(J) > P(J_i)) \\ 0 & : \ \textit{else} \end{cases} \tag{4}$$

Here, $c(J)$ is used as an abbreviation of $c(\alpha(J), \beta(J))$ and the dependence of $\alpha$ and $\beta$ on the individual $J$ is explicitly denoted by $\alpha(J)$, $\beta(J)$. Similar $P(J)$ stands for $P(\tau(J))$.

### *3.5. Parameters of the Evolutionary Algorithm*

In order to apply an Evolutionary Algorithm successfully to a specific optimization problem, several parameters have to be adjusted. Most important are the coding mechanism and the fitness function that have been described above. For understanding its functionality, the selection scheme and recombination mechanism are briefly outlined.

The selection method should maintain a high diversity in the population, i.e., not only the particular fitness value of an individual is of interest (as in standard selection schemes), but also its phenotypical "uniqueness". This means that many copies of a good individual should be avoided but different individuals with a good fitness value should be preserved. This is usually achieved by replacing the most "similar" individual out of a randomly chosen crowd of the population by the new individual if it has a better fitness. The "similarity" introduces a new selection criterion and makes a metric necessary to define similarity. Herein, this metric is the number of differently bound functional nodes. The particular selection method used herein is called *restricted tournament selection* [14].

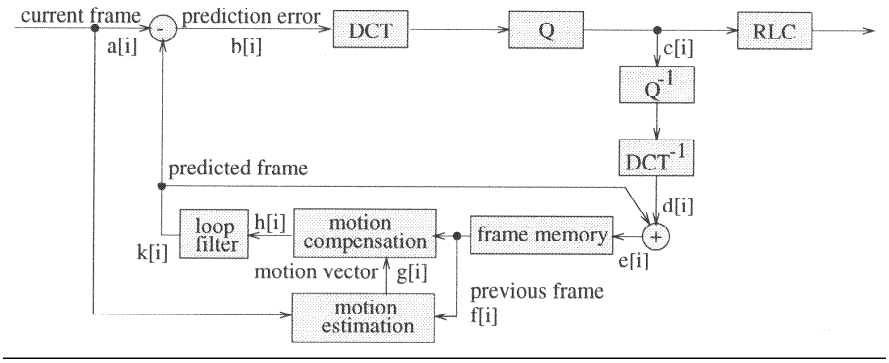


*Figure 16.* Behavioral specification of a video codec for video compression.

The specific encoding of an individual makes special crossover and mutation schemes necessary. In particular, for the allocations $\alpha_i$ uniform crossover is used [21] that randomly swaps a bit between two parents with a probability of 50%. For the lists (repair allocation priority lists $L_{Ri}$, binding order lists $L_{Oi}$ and the binding priority lists $L_{Bi}(v)$), order based crossover (also named position-based crossover) is applied (see e.g., [22, 10]). Order based crossover ensures that only permutations of the elements in the chromosomes are created, i.e., parts of the list of the parents are combined and repaired such that a legal permutation is obtained. The probability of crossover is 50%. The construction of the individuals makes further repairing methods unnecessary.

A mutation of an allocation $\alpha_i$ consists in simply swapping the allocation bit with a probability of 50%. The mutation operator for the lists creates a new permutation of a list by swapping two randomly chosen elements of the list. Mutation is applied to 20% of the individuals of a population.

## 4. Case Study

### *4.1. Problem Specification*

We explain our methodology using the example of a video codec for image compression using the H.261 standard. Its block diagram is shown in Fig. 16.

The behavioral specification is refined to the problem graph shown in Fig. 17 (coder). The synthesis problem is restricted to single level of hierarchy, i.e., only the mapping of the problem graph $G_P$ to the architecture graph $G_A$ is examined. Motion estimation is done by the block matching operation (BM), the block subtraction is named DIFF (difference) and the block addition REC (recover). Additionally, the quantization is split up into threshold calculation (TH) and quantization (Q). The operations are performed on macro blocks and each communication node represents a transmission of one macro block. However, the block matching operation needs in average three macro blocks of the previous frame for its operation. This is symbolized by a “3” in the corresponding communication node (node 17).

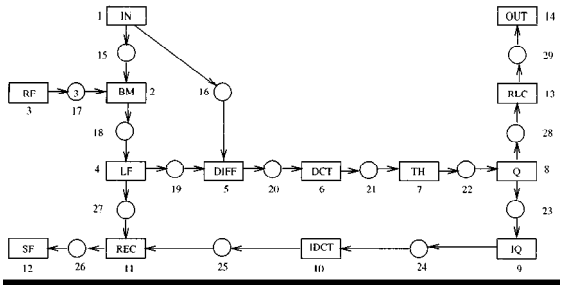


*Figure 17.* Problem graph of the video coder in Fig. 16. Functional nodes are symbolized by squares, communication nodes by circles.

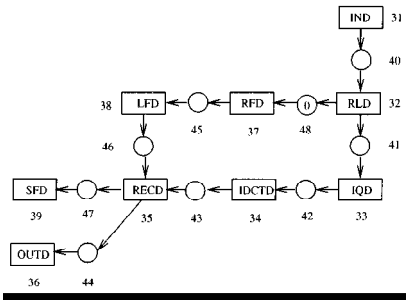


*Figure 18.* Problem graph of the video decoder.

The coder is usually combined with a decoder. Its problem graph is depicted in Fig. 18. Basically, it consists of the lower part of the loop of the coding algorithm. However, the motion compensation vector has to be extracted from the input data after the run-length decoding operation (RLD). As the amount of transmitted data is small as compared to the size of a macro block, the transmission is assumed to take zero time (symbolized by a "0" in communication node 48).

The complete graph is mapped onto a target architecture shown in Fig. 19. The architecture consists of three shared busses with different data rates, two memory modules (a single and a dual ported memory), two programmable RISC processors, a signal processor (DSP), several predefined hardware modules (namely a block matching module (BMM), a module for performing DCT/IDCT operations (DCTM), an subtract/adder module (SAM) and a Huffman coder (HC)), and I/O devices (INM and OUTM).

The basic idea behind this architecture set is the allowance of a wide range of possible implementations. The three busses are modeled to be alternatively used, as each module is intended to have only a single bus connector. This can be easily done when constructing the specification graph. Only the BM-module may use two ports of which one is dedicated to one port of the (potentially used) dual ported frame memory (DPFM). This special point-to-point bus allows a fast architecture to be synthesized. The risc processors (RISC1 and RISC2) as well as the signal processor (DSP) are capable of performing any functional

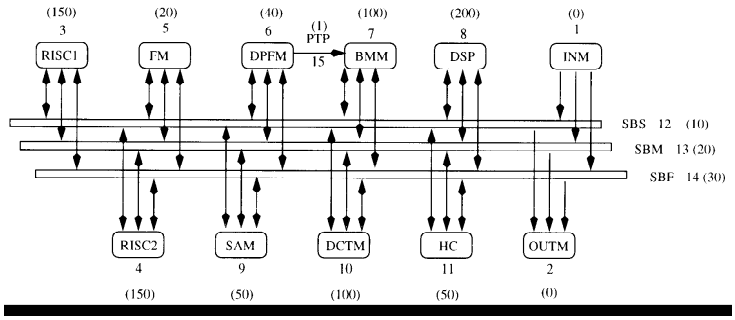


*Figure 19.* Architecture template of architectures to implement the problem graph of the video codec. The values in brackets give the basic cost $c_b$ of each resource.

operation. However, the DSP executes the operations faster and is more expensive in acquisition. The other hardware modules are dedicated to special groups of operations. For example, the DCTM can only perform the three operations DCT, IDCT, and IDCTD. Possible mappings and delays of all modules are shown in Table 1. Furthermore, several communication operations may be internal, i.e., handled by functional resources. The mappings of communication nodes to resources are specified in Table 2.

The cost of an implementation is simply reflected by the sum of the costs of the allocated hardware (see Example 8). The cost of each module is given in brackets in Fig. 19.

The number of possible bindings is $1.9 \cdot 10^{27}$. This demonstrates the intractability of enumerative or exact methods to explore the search space.

### *4.2. Optimizing Architectures*

Now, we aim to perform a design space exploration. For this purpose, the Pareto-ranking function introduced in Example 17 is used. The fitness value directly gives the number of implementations that dominate an individual $J$ and the Pareto-implementations have a fitness value of zero. Note that the Pareto-points determined in the way above are only Pareto-points according to the current population. In order to obtain many Pareto-optimal implementations, the population size is chosen as $N = 100$. The Pareto-set found in a single optimization run after 200 generations is shown in Table 3 and depicted in Fig. 15. Shown are the design points of the final population. One can see that whole design space is covered and several Pareto-points have been obtained.

In general, there is no evidence that the Pareto-points obtained above are the true Pareto-points of the problem. Surprisingly, all points in Table 3 turn out to be true Pareto-points of the problem.

Figure 20 shows the Ganttchart and the architecture of the fastest implementation found. The minimal period of the implementation is obviously determined by the execution time of the block matching module (BMM).

The cheapest implementation is shown in Fig. 21. There are no surprises here as the cheapest way to implement the codec algorithm is to use the cheapest module capable of

*Table 1.* The mapping of the functional nodes to architectural nodes.

| Node # | Operation | Resource/ Delay | Resource/ Delay | Resource/ Delay |
|---|---|---|---|---|
| 1 | IN | INM/0 | | |
| 31 | IND | INM/0 | | |
| 14 | OUT | OUTM/0 | | |
| 36 | OUTD | OUTM/0 | | |
| 2 | BM | BMM/22 | DSP/60 | RISC/88 |
| 3 | RF | FM/0 | DPFM/0 | |
| 37 | RFD | FM/0 | DPFM/0 | |
| 12 | SF | FM/0 | DPFM/0 | |
| 39 | SFD | FM/0 | DPFM/0 | |
| 4 | LF | HC/2 | DSP/3 | RISC/9 |
| 38 | LFD | HC/2 | DSP/3 | RISC/9 |
| 5 | DIFF | SAM/1 | DSP/2 | RISC/2 |
| 6 | DCT | DCTM/2 | DSP/4 | RISC/8 |
| 10 | IDCT | DCTM/2 | DSP/4 | RISC/8 |
| 34 | IDCTD | DCTM/2 | DSP/4 | RISC/8 |
| 7 | TH | HC/2 | DSP/8 | RISC/8 |
| 8 | Q | HC/1 | DPS/2 | RISC/2 |
| 9 | IQ | HC/1 | DSP/2 | RISC/2 |
| 33 | IQD | HC/1 | DSP/2 | RISC/2 |
| 11 | REC | SAM/1 | DSP/2 | RISC/2 |
| 35 | RECD | SAM/1 | DSP/2 | RISC/2 |
| 13 | RLC | HC/2 | DSP/8 | RISC/8 |
| 32 | RLD | HC/2 | DSP/8 | RISC/8 |

handling all functionalities, the cheapest bus and the cheapest memory module.

As a final example, consider the Pareto-point with cost $c = 280$ and period $P = 78$. The Ganttchart and architecture are shown in Fig. 22.

In this section, a real-world synthesis problem has been considered, the synthesis of an architecture for a H.261 video codec. The complexity of the problem is realistic and typical for system-level synthesis. The methodology is able to simultaneously select the architecture (allocation), perform the assignment of operations to resources (binding) and annotate starting times to operations (scheduling). The implementation of the model is based on Evolutionary Algorithms. Remarkable is the fact that the EA is capable of exploring the design space and identifying the Pareto-points of a synthesis problem in a single optimization run.

## 5. Summary and Conclusions

An approach to system-level synthesis for data flow-dominant hardware/software systems has been 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 limited pin numbers are considered as well as maximal

*Table 2.* The mapping of the communication nodes to architectural nodes.

| Comm. Node # | Res./ Delay | Res./ Delay | Res./ Delay | Res./ Delay | Res./ Delay | Res./ Delay |
|---|---|---|---|---|---|---|
| 15 | SBF/1 | SBM/2 | SBS/3 | | | |
| 16 | SBF/1 | SBM/2 | SBS/3 | | | |
| 17 | SBF/3 | SBM/6 | SBS/9 | PTP/1 | | |
| 18 | SBF/1 | SBM/2 | SBS/3 | RISC/0 | DSP/0 | |
| 19 | SBF/1 | SBM/2 | SBS/3 | RISC/0 | DSP/0 | |
| 20 | SBF/1 | SBM/2 | SBS/3 | RISC/0 | DSP/0 | |
| 21 | SBF/1 | SBM/2 | SBS/3 | RISC/0 | DSP/0 | |
| 22 | SBF/1 | SBM/2 | SBS/3 | RISC/0 | DSP/0 | HC/0 |
| 23 | SBF/1 | SBM/2 | SBS/3 | RISC/0 | DSP/0 | HC/0 |
| 24 | SBF/1 | SBM/2 | SBS/3 | RISC/0 | DSP/0 | |
| 25 | SBF/1 | SBM/2 | SBS/3 | RISC/0 | DSP/0 | |
| 26 | SBF/1 | SBM/2 | SBS/3 | | | |
| 27 | SBF/1 | SBM/2 | SBS/3 | RISC/0 | DSP/0 | |
| 28 | SBF/1 | SBM/2 | SBS/3 | RISC/0 | DSP/0 | HC/0 |
| 29 | SBF/1 | SBM/2 | SBS/3 | | | |
| 30 | SBF/0 | SBM/0 | SBS/0 | | | |
| 40 | SBF/1 | SBM/2 | SBS/3 | | | |
| 41 | SBF/1 | SBM/2 | SBS/3 | RISC/0 | DSP/0 | HC/0 |
| 42 | SBF/1 | SBM/2 | SBS/3 | RISC/0 | DSP/0 | |
| 43 | SBF/1 | SBM/2 | SBS/3 | RISC/0 | DSP/0 | |
| 44 | SBF/1 | SBM/2 | SBS/3 | | | |
| 45 | SBF/1 | SBM/2 | SBS/3 | | | |
| 46 | SBF/1 | SBM/2 | SBS/3 | RISC/0 | DSP/0 | |
| 47 | SBF/1 | SBM/2 | SBS/3 | | | |

*Table 3.* Implementations obtained as Pareto-points in a single run of the Evolutionary Algorithm.

| | $J_1$ | $J_2$ | $J_3$ | $J_4$ | $J_5$ | $J_6$ |
|---|---|---|---|---|---|---|
| Period $P$ | 22 | 42 | 54 | 78 | 114 | 166 |
| Cost $c$ | 350 | 340 | 330 | 280 | 230 | 180 |

cost and latency constraints. Furthermore, a capacity constraint can be formulated for each resource. Communication delays are modeled dependent on selected busses, bus-widths and transfer rates.

The main optimization loop contains an Evolutionary Algorithm. Based on 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 scheduling heuristic. Then, the fitness function is evaluated for each implementation. In case of Pareto-ranking, the design space can be explored in one single optimization run.

We claim that the combination of an Evolutionary Algorithm for architecture selection with an heuristic scheduler after architecture selection and binding is a promising approach to system-level synthesis. Finally, Evolutionary Algorithms can be easily parallelized. This is a topic of future investigation.

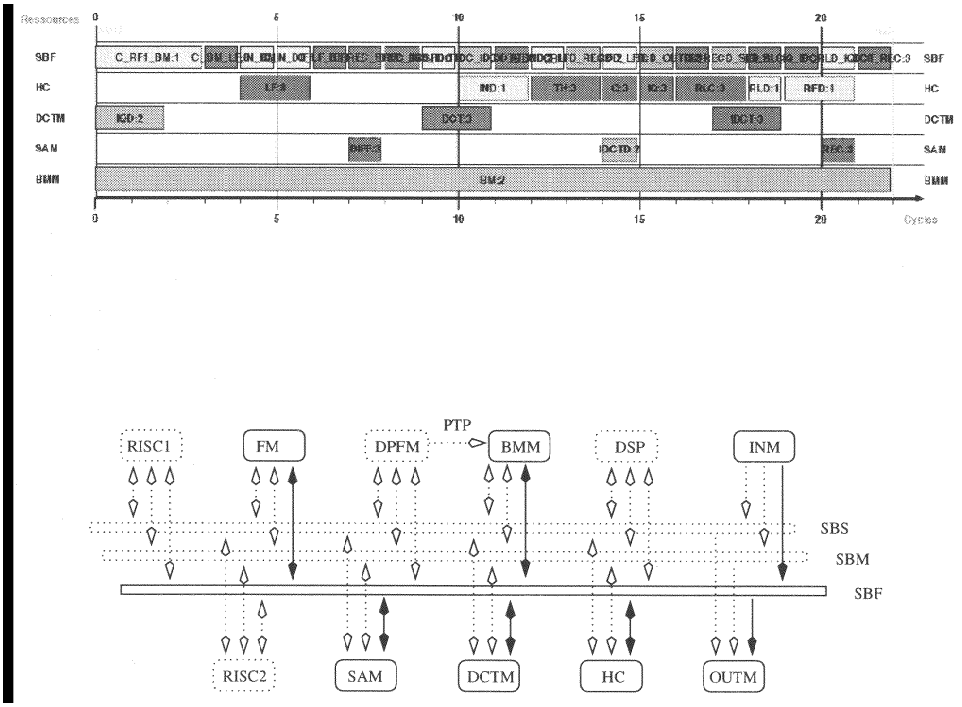


*Figure 20.* Ganttchart and architecture of the fastest implementation (cost $c = 350$, period $P = 22$).

The optimization methodology presented here is part of the CodeSign project at ETH Zurich. Due to space requirements, the underlying design framework Code-Sign could not be explained here, e.g., specification based on Petri-nets, simulation, applied synthesis and estimation tools. At the current state of implementation, the design parameters are extracted from synthesized library components using commercial tools (Synopsys, Xilinx, etc.). The design data for the video codec have been elaborated in cooperation with Siemens AG, Munich, Germany.

## Appendix

*Proof of Theorem 1.* Obviously, the test of a binding for feasibility can be done in polynomial time, see Def. 6. 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

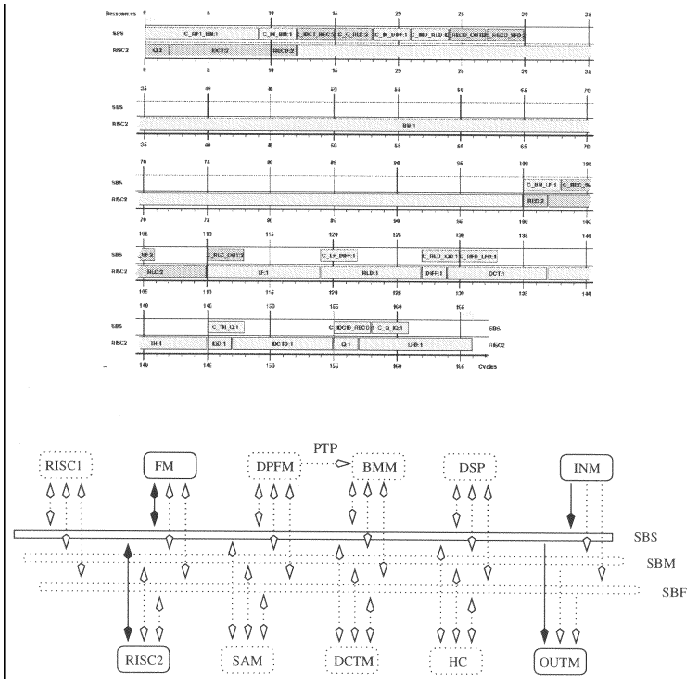


*Figure 21.* Ganttchart and architecture of the cheapest implementation (cost $c = 180$, period $P = 166$).

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.

## References

1. E. Barros and W. Rosenstiel. A method for hardware software partitioning. In *Proc. 1992 COMPEURO: Computer Systems and Software Engineering*, pp. 580–585. The Hague, Netherlands, May 1992.
2. T. Blickle. *Theory of Evolutionary Algorithms and Application to System Synthesis*. PhD thesis, Institute TIK, ETH Zurich, Switzerland, 1997. TIK-Schriftenreihe Nr. 17, vdf, Hochsch.-Verl.
3. T. Blickle and L. Thiele. A comparison of selection schemes used in evolutionary algorithms. *Evolutionary Computation* 4(4), 1996.

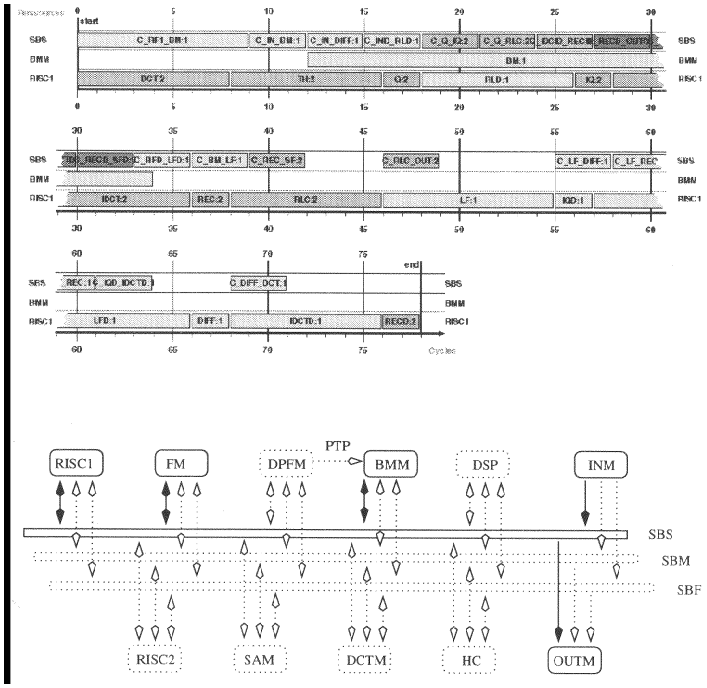


*Figure 22.* Ganttchart and architecture of the implementation with cost $c = 280$ and period $P = 78$.


4. K. Buchenrieder, A. Sedlmeier, and C. Veith. Codes a framework for modeling heterogeneous systems. In J. Rozenbilt and K. Buchenrieder, editors, *Codesign: Computer-Aided Software/Hardware Engineering*. IEEE Press, Piscataway, NJ, U.S.A., 1995, pp. 378–392.
5. R. Camposano and R. K. Brayton. Partitioning before logic synthesis. In *Proc. ICCAD*, 1987.
6. J. G. D'Ambrosio and X. Hu. Configuration-level hardware/software partition for real-time embedded systems. In *Proc. of CODES/CASHE'94, Third Intl. Workshop on Hardware/Software Codesign*, pp. 34–41, Grenoble, France, September 1994.
7. J. G. D'Ambrosio, X. Hu, and G. W. Greenwood. An evolutionary approach to configuration-level hardware/software partitioning. Technical Report R & D-8465, General Motors Corporation, R & D Center, 30500 Mound Road, Box 9055, Warren, Michigan 40090-9055, December 1995.
8. G. DeMicheli. *Synthesis and Optimization of Digital Circuits*. McGraw Hill, New York, 1994.
9. R. Ernst, J. Henkel, and T. Benner. Hardware-software cosynthesis for microcontrollers. *IEEE Design & Test of Computers*, pp. 64–75, December 1994.
10. B. R. Fox and M. B. McMahon. Genetic operators for sequencing problems. In Gregory J. E. Rawlins, editor, *Foundations of Genetic Algorithms*, Morgan Kauffman Publishers, 1991, pp. 284–300.
11. D. E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, 1989.
12. 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.
13. 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*, pp. 1–10, Passau, Germany, March 1995.
14. G. R. Harik. Finding multimodal solutions using restricted tournament selection. In Larry J. Eshelman, editor, *Proceedings of the Sixth International Conference on Genetic Algorithms (ICGA6)*, Morgan Kaufmann, 1995.

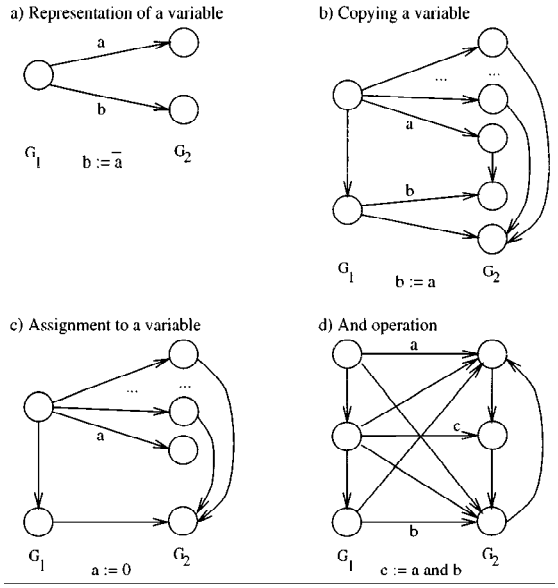


*Figure A.1.* Components for the transformation of Boolean expressions to a specification graph.

15. 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)*, pp. 464–473, 1994.
16. 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.
17. 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.
18. 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.
19. M. C. McFarland. Using bottom-up design techniques in the synthesis of hardware from abstract behavioral descriptions. In *Proc. 23rd Design Automation Conference*, pp. 474–480, June 1986.
20. Z. Michalewicz. Genetic algorithms, numerical optimization, and constraints. In Larry J. Eshelman, editor, *Proceedings of the Sixth International Conference on Genetic Algorithms (ICGA6)*, Morgan Kaufmann, 1995, pp. 151–158.
21. G. Syswerda. Uniform crossover in genetic algorithms. In J. David Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, Morgan Kaufmann Publishers, San Mateo, CA, 1989, pp. 2–9.
22. G. Syswerda and J. 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*, Morgan Kaufmann Publishers, San Mateo, CA, 1991, pp. 502–508.
23. J. Teich, T. Blickle, and L. Thiele. An evolutionary approach to system-level synthesis. In *Proc. of Codes/CASHE'97 - The 5th International Workshop on Hardware/Software Codesign*, Braunschweig, Germany, March 1997, pp. 167–171.
24. D. E. Thomas, J. K. Adams, and H. Schmitt. A model and methodology for hardware-software codesign. *IEEE Design & Test of Computers* 10(3): 6–15, September 1993.
25. F. Vahid and D. Gajski. Specification partitioning for system design. In *Proc. 29th Design Automation Conference*, Anaheim, CA, June 1992, pp. 219–224.