An Evolutionary Approach to System- Level Synthesis

Jürgen Teich, Tobias Blickle, Lothar Thiele

***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, 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 new *graph-based mapping* model is introduced that handles all mentioned requirements and allows the task of system-synthesis to be specified as an optimization problem. An Evolutionary Algorithm is applied to solve the synthesis problem.**

I. Introduction

State-of-the art logic and high level synthesis tools help 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 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*: 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.

A. 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 authors are with the Computer Engineering and Communication Networks Lab (TIK), Swiss Federal Institute of Technology (ETH), CH-8092 Zürich, Switzerland, email: teich@tik.ee.ethz.ch

Our approach provides

- *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,
- *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.

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), and c) 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.

B. Existing Approaches to System-Level Synthesis

There exist already many different approaches to system-level synthesis. Some approaches focus on *control-dominant systems*, e.g., [7], [5], and [8]. Other approaches deal with *dataflow-dominant designs* like [1], and [6]. The term system synthesis, however, is weakly defined. In some approaches, the result of the synthesis procedure is the description of a *dedicated control & data path in VLSI*, e.g., [11], [10], [12] or a *multi-chip dedicated VLSI architecture*, e.g., [9]. In other cases, it is a mixed *hardware/software architecture*. In most of these cases, the architecture is already fixed, see, e.g., [5], or [6] and the synthesis problem is the problem of partitioning functionality into hardware and software blocks, see, e.g., [3], [11], [10], [12].

C. Overview

Figure 1 gives an overview of our optimization methodology. The specification consists of a dataflow graph based problem graph (see Fig. 1a), an architectural model (see Fig. 1c), user-defined mapping constraints (see Fig. 1b) and an optimization procedure (see Fig. 1d) using an *Evolutionary Algorithm* (EA) that 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 EA consists of an optimization

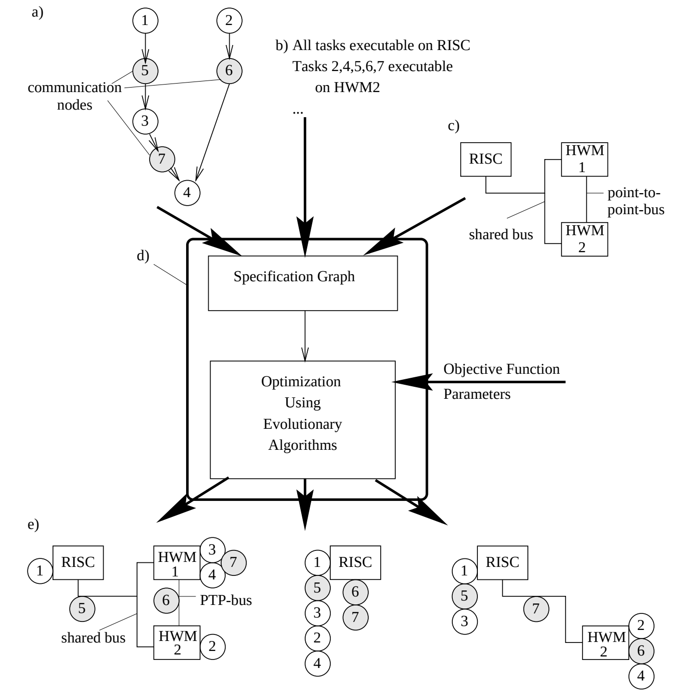
a)
communication nodes
b) All tasks executable on RISC
Tasks 2,4,5,6,7 executable on HWM2
c)
RISC
HWM 1
HWM 2
point-to-point-bus
shared bus
d)
Specification Graph
Optimization Using Evolutionary Algorithms
Objective Function Parameters
e)
PTP-bus
shared bus

Fig. 1. Overview of the proposed optimization methodology: a) dataflow based algorithm specification, b) mapping constraints, c) architecture template, d) optimization methodology, e) implementations

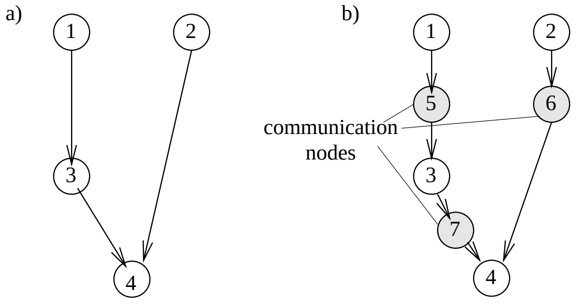

Fig. 2. A dataflow graph (a) and the corresponding problem graph (b)

loop (see Fig. 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 EA terminates after a certain number k_{max} of generated populations and outputs those implementations with the best fitness values.

Example 1 *Fig. 1e) shows some individuals out of a population P_k of implementations. They correspond to the problem graph in Fig. 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 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.

II. The Problem of System Synthesis

A. Graph-Based Specification

- 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* $G(V,E)$.
- User-defined mapping constraints between the operations and architecture components result in a specification graph $G_S(V_S,E_S)$. Additional parameters which are used to formulate 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 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 dataflow 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 define a partial ordering among the operations.

Example 2 *A problem graph is obtained from the task-level dataflow graph by inserting a communication node for each edge of the dataflow graph (see Fig. 2). 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, ASIC, dedicated processor) 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 *Fig. 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). Fig. 3b) 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.

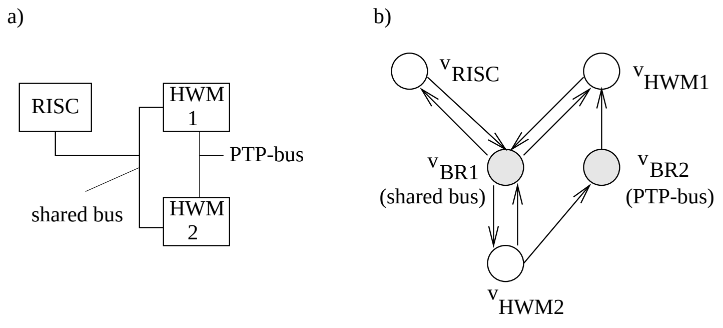

Fig. 3. An example of an architecture (a), and the corresponding architecture graph G_A (b)

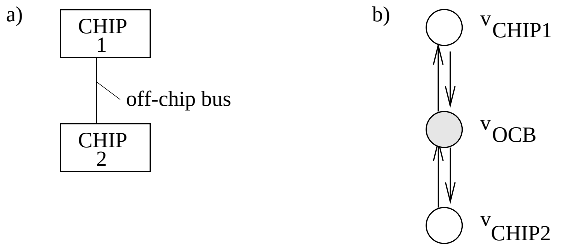

Fig. 4. An example of a multi-chip architecture (a) and the corresponding chip graph G_C (b).

Example 4 *Fig. 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. Fig. 4b) shows the corresponding chip graph G_C.*

Note that in the above 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.

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 $1 \le i \le D$ 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 $1 \le i < D$.*

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 5 *Fig. 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}.*

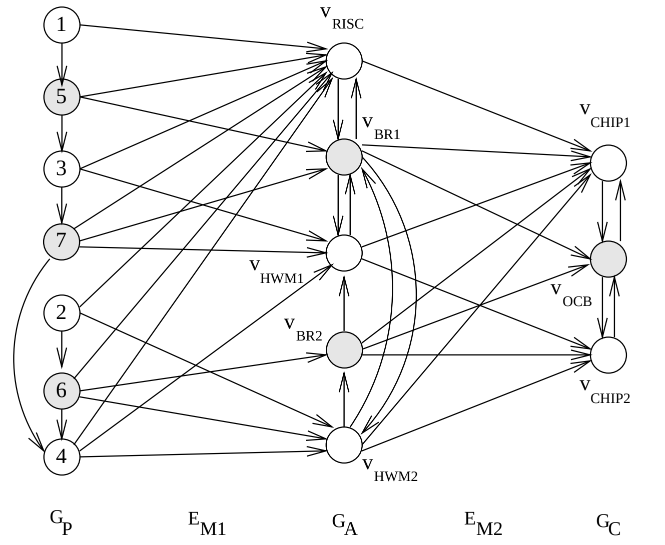

Fig. 5. An example of a specification graph G_S

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}. 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 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 presented so far, all nodes and edges of the problem graph contained in G_S have been selected, i.e., activated. The determination of 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.

B. System Synthesis

Definition 4 (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 5 (Binding) *A* binding β *is the subset of all activated mapping edges, i.e.,*

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

Definition 6 (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.*
$$\tilde{v}_i = \tilde{v}_j \quad with \quad (v_i, \tilde{v}_i), (v_j, \tilde{v}_j) \in \beta$$
   - *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.*
$$(\tilde{v}_i, \tilde{v}_j) \in \alpha_E \quad with \quad (v_i, \tilde{v}_i), (v_j, \tilde{v}_j) \in \beta$$

The synthesis procedure which will be presented simultaneously determines allocation and binding.

Definition 7 (Feasible Allocation) *A* feasible allocation α *is an allocation that allows at least one feasible binding β.*

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 EA.

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

Proof: By polynomial reduction of BOOLEAN SATISFIABILITY, see [2]. ■

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.

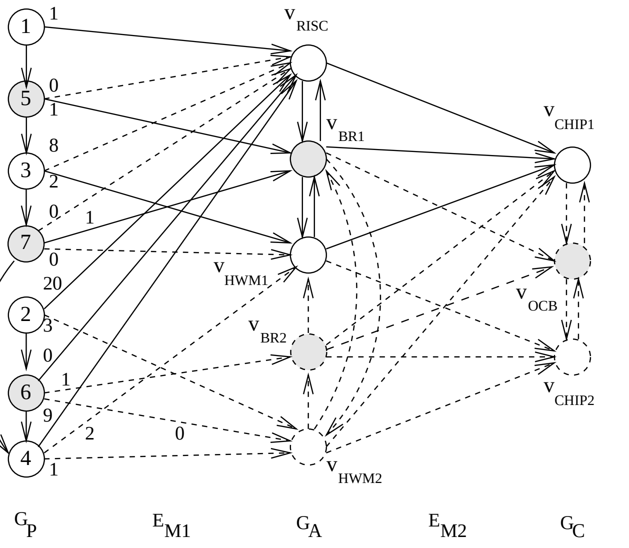

Fig. 6. An example of an implementation from the specification given in Fig. 5.

Definition 8 (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.

Definition 9 (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 6 *Fig. 6 shows an implementation of the specification from Fig. 5. 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 $\tau(v_1) = 0$, $\tau(v_2) = 1$, $\tau(v_3) = 2$, $\tau(v_4) = 21$, $\tau(v_5) = 1$, $\tau(v_6) = 21$, $\tau(v_7) = 4$.

C. The Tasks of System Synthesis

With the model introduced above 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

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 7 *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 $f(\alpha, \beta, \tau) = max\{\tau(v) + delay(v, \beta))\ |v \in V_P\}$.*

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.

III. Optimization Method

In this section the application of an Evolutionary Algorithm is described to solve the problem of system synthesis as specified by Definition 10. 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 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 division of work is depicted in Fig. 7.

A. Coding of Implementations

As an EA includes a certain amount of randomness one has to address the question how to handle infeasible allocations and infeasible bindings. Obviously, if allocations and bindings are (at least in some sense) randomly chosen, a lot of them may be infeasible. In general, there are two different methods to handle these invalid implementations. First, 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 bad performance. On the other hand, one could "repair" invalid allocations and bindings with some mechanism and incorporate domain knowledge in these repair mechanisms. However, the determination of a feasible allocation or binding is NP-complete, so that 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 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 $D-1$. In each level the following steps are executed: first, the allocation of nodes V_{i+1} is decoded from the individual and repaired with a simple heuristic, next the binding of the edges $e \in E_{Mi}$ is performed. 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.

The scheduler chosen here is a list scheduler, see, e.g., [4] 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 scheduler performs 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 for a given 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.

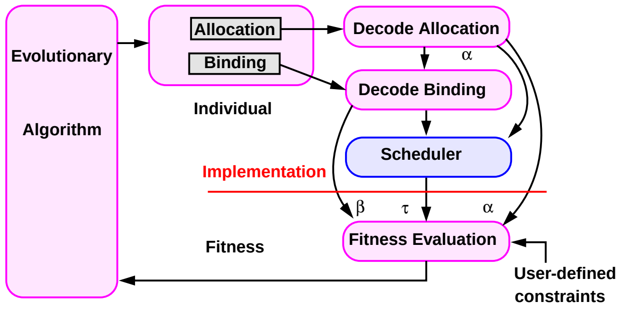

Fig. 7. The decoding of an individual to an implementation.

B. *Cost Evaluation: The Choice of the Fitness Function*

The fitness function F to be minimized is given as

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

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 constraints $g_1, \ldots, g_q$. 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 8 *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*

$$f(\alpha, \beta, \tau) = c_c f_c + c_t f_t \qquad (2)$$

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., $f_t = \frac{latency(\alpha,\beta,\tau)}{MAXLATENCY}$, $f_c = \frac{c(\alpha,\beta)}{MAXCOST}$. 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. The value c is given by the amount of allocated hardware (see Example 7). The constraints on cost and time are furthermore transformed into two constraints $g_1 = MAXLATENCY - latency(\alpha, \beta, \tau)$ 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. 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.

IV. Case Study

We explain our methodology using the example of a video codec for image compression. Fig. 8 shows the problem graph G_P of the video codec.

We want to map the graph onto a target architecture that is shown in Fig. 9. The architecture consists of a shared bus, several memory modules, two programmable RISC processors and two predefined hardware modules, namely a block matching module (BMM) and a module for performing DCT/IDCT operations (DCTM).

All nodes can be implemented in software on each of the RISC processors. The block-matching node v_9 can be realized on resource BMM, the DCT- and IDCT-operations may be realized on module DCTM. There is one shared bus between all components. The user-defined constraints have been chosen as $MAXCOST = 5000$, and $MAXLATENCY = 70$ cycles (cost parameters of components not shown here due to space requirements). Hence, an implementation that uses all functional resources would be too expensive.

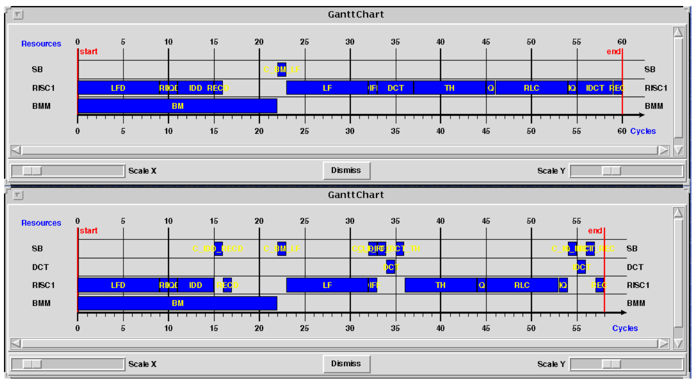

Fig. 10. Results of applying the EA system synthesizer to the H.261 video codec. Two optimal implementations have been found in a single optimization run.

We use the fitness function according to example 8 with a simple cost function

$$f = 0.4 \frac{\sum_{\tilde{v} \in \alpha} c_h(\tilde{v})}{5000} + 0.6 \frac{\tau_L(\beta, \tau)}{70} \qquad (3)$$

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:

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

The considered penalties were set to $p_1 = 2$ and $p_2 = 1$ (see Example 8).

With the population size of $N = 30$ individuals and $k_{max} = 50$ generations, the following results have been obtained: Fig. 10 shows two optimal implementations found: 1) uses BMM and RISC1, $\tau_L(\beta, \tau) = 60$, $\sum_{\tilde{v} \in \alpha} c_h(\tilde{v}) = 2500$ and has a fitness of 0.71, 2) uses BMM, RISC1 and DCTM, $\tau_L(\beta, \tau) = 58$, $\sum_{\tilde{v} \in \alpha} c_h(\tilde{v}) = 3300$ and has a fitness of 0.76.

V. 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.

The main optimization loop contains an Evolutionary Algorithm. Based on populations of implementations it

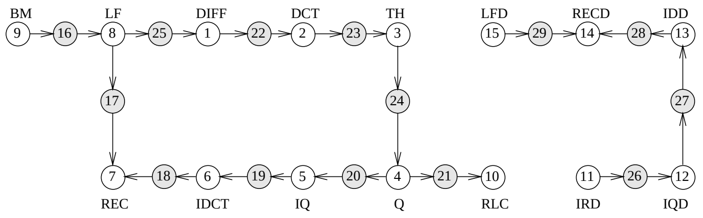

Fig. 8. Problem graph of the video codec.

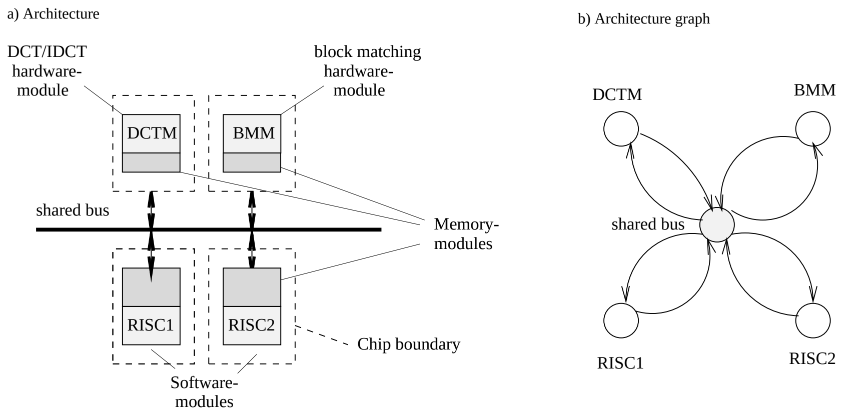

Fig. 9. Architecture template of architectures to implement the problem graph in Fig. 8.

performs a parallel search for optimal architecture selection (allocation) and binding in each iteration. For each architecture and binding, a resource-constrained schedule is computed using a scheduler. This way, the design space can be explored in one single optimization run. Different fitness functions have been implemented and tested.

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 feature that other resource-constrained schedulers may be plugged into the optimizer.

References

[1] 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.

[2] T. Blickle, J. Teich, and L. Thiele. System-level synthesis using evolutionary algorithms. Technical Report 16, Computer Engineering and Communication Networks Lab (TIK), Swiss Federal Institute of Technology (ETH) Zurich, Gloriastrasse 35, CH-8092 Zurich, April 1996. http://www.tik.ee.ethz.ch/ Publications/ TIK-Reports/ TIK-Report16.ps.Z.

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

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

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

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

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