# Flexibility/Cost-Tradeoffs of Platform-Based Systems[*]

Christian Haubelt[1], Jürgen Teich[1], Kai Richter[2], and Rolf Ernst[2]

[1] DATE, University of Paderborn, Paderborn, Germany
{haubelt, teich}@date.upb.de,
URL: http://www-date.upb.de
[2] IDA, Technical University of Braunschweig, Braunschweig, Germany
{richter, ernst}@ida.ing.tu-bs.de,
URL: http://www.ida.ing.tu-bs.de

**Abstract** This paper provides a quantitative characterization of an embedded system's capability to implement alternative behaviors. This new objective in system-level design is termed *flexibility* and is most notable in the field of adaptive and reconfigurable systems, where a system may change its behavior during operation. Different behaviors are also taken into consideration while implementing platform-based systems. Based on a *hierarchical graph model* which permits formal modeling of flexibility and implementation cost of a system, an efficient exploration algorithm to find the *optimal flexibility/cost-tradeoff-curve* is proposed. The feasibility of our approach is demonstrated by a case study concerning the design of a family of Set-Top boxes.

## 1   Introduction

Multi-dimensional optimization of a system for a given, single application is challenging, but has been formalized already by methods of graph-based allocation and binding (see [1]) which are used in commercial systems such as VCC by Cadence [2].

In platform-based design, however, a system should be dimensioned such that it is able to implement not only one particular application optimally, but a complete set of different applications or variants of a certain application. Hence, the task is to find a tradeoff between cost and flexibility of an architecture which is able to implement several alternative behaviors.

In adaptive systems which have to react to environmental changes, the definition of flexibility is of utmost importance. Here, it is also necessary to implement different behaviors. Unfortunately, this may cause additional cost. For this purpose, reconfigurable architectures seem to be an adequate choice.

As far as we are concerned, there is no approach that quantitatively tradeoffs implementational cost in terms of additional memory, hardware, network, etc. and the flexibility of a system which implements multiple behaviors. Here,

---

we introduce *flexibility* as a tentative to quantitatively describe the functional richness that the system under design is able to implement (Section 3). In order to describe a set of applications, a hierarchical specification is useful, e.g., see [3,4]. In this paper, we introduce a *hierarchical graph model* for describing alternatives of the behavior of a system. The same idea may be used in order to describe reconfigurable architectures on the implementation side, i.e., systems that change their structure over time.

With this model, we are then able to define the problem of dimensioning a system that is able to dynamically switch its behavior and/or structure at run-time. Basically, this problem extends previous approaches such as [1] to reconfigurable, platform-based systems that implement time-dependent functionality.

Finally, an efficient exploration algorithm for exploring the flexibility/cost-tradeoff-curve of a system under design is presented that efficiently prunes solutions that are not optimal with respect to both criteria (Section 4). The example of a flexible video Set-Top box is used as the guiding example throughout the paper.

## 2   Hierarchical Specification Model

Each embedded system is developed to cover a certain range of functionality. This coverage depends on the different types of tasks as well as on the scope every task is able to process. A specification of such an embedded system is depicted in Fig. 1.

The specification shows interacting processes of a digital television decoder. There are four top-level processes, $P_A$ to handle the authentification process, $P_C$ to control channel selection, frequency adjustment, etc., $I_D$ which performs decryption, and $I_U$ for uncompression. Here, the uncompression process requires input data from the decryption process. Furthermore, the controller and authentification process are well known and are most likely to be implemented equally in each decoder.

The main difference between TV decoders consists of the implemented combinations of decryption and uncompression algorithms. As shown in Fig. 1, we use hierarchical refinement to capture all alternative realizations. There are three decryption and two uncompression processes used in this decoder. Obviously, if we implement even more of these refinements, our decoder will support a greater number of TV stations. Consequently, the decoder possesses an increased flexibility.

Before defining the flexibility of a system formally, we have to introduce a specification model that is able to express flexibility. As shown in Fig. 1, our specification model is based on the concept of hierarchical graphs, where a hierarchical graph possesses vertices that can be refined by subgraphs. If it is possible to replace such hierarchical vertices by a set of alternative subgraphs, we call these subgraphs *clusters* and the hierarchical vertices are termed *interfaces*. Definitions 1 to 3 declare this basic structure of hierarchical graphs.
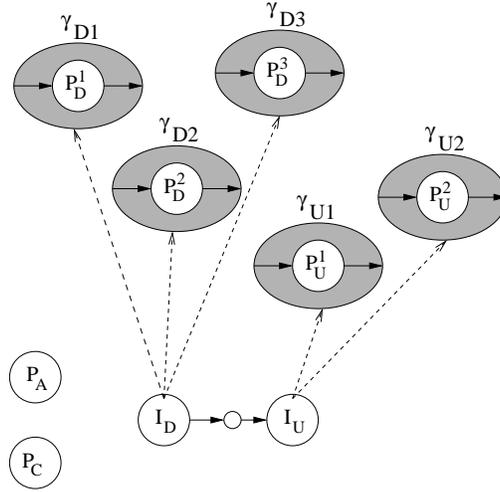
**Figure 1.** Specification of a Digital TV Decoder

**Definition 1 (cluster).** *A* cluster $\gamma(I,O,V,E,\Psi)$ *contains a directed non-hierarchical graph* $G = (V_G, E_G)$, *where $V$ and $\Psi$ define a partitioning of the set of vertices $V_G$ and $E_G$ is the set of edges $E$. Furthermore, we define:*

*$I = \{i_1, i_2, \ldots, i_{N_I}\}$ is the set of inputs,*
*$O = \{o_1, o_2, \ldots, o_{N_O}\}$ is the set of outputs,*
*$V = \{v_1, v_2, \ldots, v_{N_V}\}$ is the set of non-hierarchical vertices or* leaves,
*$E \subseteq (I \times \{V \cup I_\Psi\}) \cup (V \times \{V \cup I_\Psi\}) \cup (O_\Psi \times \{V \cup O\}) \cup (V \times O)$ is the set of edges, where $I_\Psi$ and $O_\Psi$ denote the unions of the inputs and outputs of the interfaces (see Def. 2), respectively, and*
*$\Psi = \{\psi_1, \psi_2, \ldots, \psi_{N_\Psi}\}$ is the set of hierarchical vertices or* interface *as defined by Definition 2.*

While leaves can not be refined, interfaces are used as placeholders to embed subgraphs. Since the in-degree and out-degree of an interface is not limited, we need the notion of *ports*. Interfaces are connected with vertices or other interfaces via ports. These ports are used to embed clusters into a given interface. In the sequel, we use the term *ports* for the union of the in- and outputs $(I \cup O)$.

An *interface* is defined as follows:

**Definition 2 (interface).** *An* interface $\psi(I, O, \Gamma, \Phi)$ *is a 4-tuple* $(I, O, \Gamma, \Phi)$, *where*

*$I = \{i_1, i_2, \ldots, i_{N_I}\}$ denotes the set of inputs,*
*$O = \{o_1, o_2, \ldots, o_{N_O}\}$ denotes the set of outputs,*
*$\Gamma = \{\gamma_1, \gamma_2, \ldots, \gamma_{N_\Gamma}\}$ is the set of associated clusters, and*
*$\Phi : I_\Gamma \cup O_\Gamma \to I \cup O$ is a function which maps the ports of all associated clusters $\gamma \in \Gamma$ onto the ports of this interface, where $I_\Gamma$ and $O_\Gamma$ denote the unions of the inputs and outputs of the associated clusters (see Def. 1), respectively. In the following, we term this function as* port mapping.

With the Definition 1 and 2 we are able to define *hierarchical graphs*.

**Definition 3 (hierarchical graph).** *A* hierarchical graph *is a cluster (defined by Def. 1) where the set of in- and outputs are empty, i.e.,* $I = O = \emptyset$.

Figure 1 shows a digital TV decoder as a hierarchical graph with its top-level graph depicted at the bottom. The top-level graph consists of two non-hierarchical vertices, $V = \{P_A, P_C\}$ and two interfaces ($\Psi = \{I_D, I_U\}$). The decryption interface $I_D$ itself can be refined by three clusters $\gamma_{D1}$, $\gamma_{D2}$, and $\gamma_{D3}$, where each cluster represents an alternative refinement of $I_D$. The set of clusters is given by $\Gamma = \{\gamma_{D1}, \gamma_{D2}, \gamma_{D3}, \gamma_{U1}, \gamma_{U2}\}$. For simplicity, we omit the ports of the vertices, interfaces, and clusters.

All clusters associated with the interface $\psi$ represent *alternative refinements* of $\psi$. The process of *cluster-selection* associated with each interface $\psi$ determines exactly one cluster to implement $\psi$ at each instant of time. In order to avoid a loss of generality, we do not restrict cluster-selection to system start-up. Thus, reconfigurable and adaptive systems may be modeled via time-dependent switching of clusters.

The set of leaves of a hierarchical graph $G$ is defined by the recursive equation:[1]

$$V_l(G) = G.V \cup \bigcup_{\psi \in G.\Psi} \bigcup_{\gamma \in \psi.\Gamma} V_l(\gamma) \tag{1}$$

As defined by Equation (1), the set of leaves $V_l(G)$ of graph $G$ shown in Figure 1 computes to $V_l(G) = \{P_A, P_C\} \cup \{\gamma_{D1}.P_D^1, \gamma_{D2}.P_D^2, \gamma_{D3}.P_D^3\} \cup \{\gamma_{U1}.P_U^1, \gamma_{U2}.P_U^2\}$.

So far, we only considered the behavioral part of the specification. For implementation, we also require information about the possible structure of our system, i.e., the underlying architecture. This leads to a graphical model for embedded system specification, the so called *specification graph* $G_S = (G_P, G_A, E_M)$. It mainly consists of three components: a *problem graph*, an *architecture graph*, and *user-defined mapping* edges (see also [1]). The respective graphs $G_P$ and $G_A$ are based on the concept of hierarchical graphs as defined in Def. 3.

**Problem Graph.** The *problem graph* $G_P$ is a directed hierarchical graph $G_P = (V_P, E_P, \Psi_P, \Gamma_P)$ for modeling the required system's behavior (see Fig. 1). Vertices $v \in V_P$ and interfaces $\psi \in \Psi_P$ represent processes or communication operations at system-level. The edges $e \in E_P$ model dependence relations, i.e., define a partial ordering among the operations. The clusters $\gamma \in \Gamma_P$ are possible substitutions for the interfaces $\psi \in \Psi_P$.

**Architecture Graph.** The class of possible architectures is modeled by a directed hierarchical graph $G_A = (V_A, E_A, \Psi_A, \Gamma_A)$, called *architecture graph*. Functional or communication resources are represented by vertices $v \in V_A$ and interfaces $\psi \in \Psi_A$, interconnections are specified by the edges $e \in E_A$. Again, the clusters $\gamma \in \Gamma_A$ represent potential implementations of the associated interfaces. All the resources are viewed as potentially allocatable components.

---

[1] The $G.V$ notation is used as decomposition operation, e.g., to access the set of vertices $V$ inside the graph $G$.

**Mapping Edges.** *Mapping edges $e \in E_M$* indicate user-defined constraints in the form of a "can be implemented by"-relation. These edges link leaves $V_l(G_P)$ of the problem graph $G_P$ with leaves $V_l(G_A)$ of the architecture graph $G_A$.

Additional parameters, like priorities, power consumption, latencies, etc., which are used for formulating implementational and functional constraints, are annotated to the components of $G_S$. For simplicity, a specification graph can also be represented only by its vertices and edges: $G_S = (V_S, E_S)$. The set of vertices $V_S$ covers all non-hierarchical vertices, interfaces, and clusters contained in the problem or architecture graph. The set of edges $E_S$ consists of all edges and port mappings in the specification graph.

An example of a specification graph is shown in Figure 2. Again, the problem graph specifies the behavior of our digital TV decoder. The architecture graph is depicted on the right. It is composed of a $\mu$-Controller ($\mu$P), an ASIC (A), and an FPGA. There are two busses $C_1$ and $C_2$ to handle the communication between the $\mu$-Controller and FPGA and ASIC, respectively. Figure 2 also shows the allocation cost for each resource in the architecture graph.
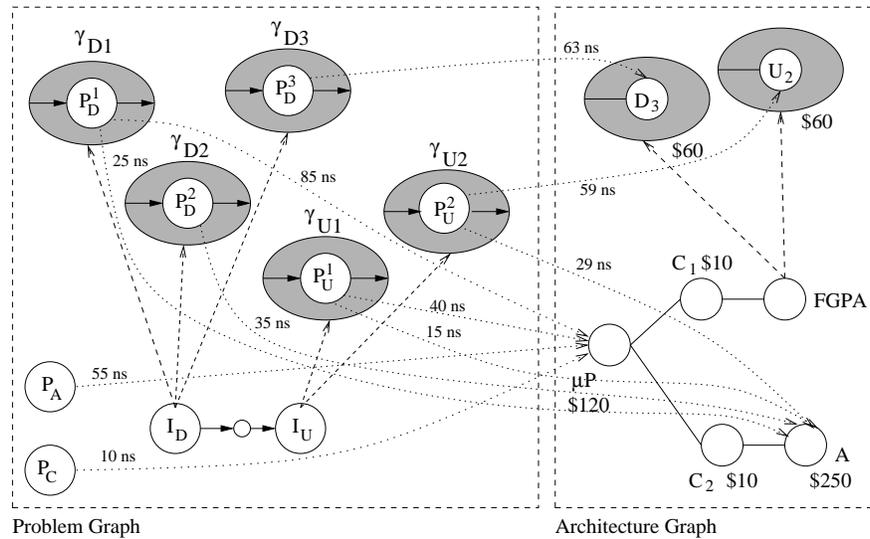


**Figure 2.** Hierarchical Specification Graph

The mapping edges (dotted edges in Fig. 2) outline the possible bindings of processes of the problem graph to resources of the architecture graph. The latencies to execute a given process on a specific resource are annotated to the respective mapping edges. For example, the uncompression process $P_U^1$ is executable on resource $\mu$P with a latency of 40 ns or on resource A with a latency of 15 ns.

As shown in Figure 2, the hierarchical specification graph permits modeling of adaptive systems by interchanging clusters in the problem graph. In our example,

we have to select a certain decryption and uncompression process to match the requirements imposed by the TV station. Generally, an adaptive system responds to environmental changes by selecting clusters according to the requirements of input/output data at runtime. Therefore, clusters with various parameters or perhaps totally different functionality are activated in the problem graph. On the other hand, interchanging clusters in the architecture graph modifies the structure of the system. If this cluster-selection is performed at runtime, the architecture model characterizes reconfigurable hardware. For example, in order to execute process $P_D^3$, we have to configure the FPGA with the respective design $D_3$ (see Figure 2).

In order to specify an *implementation*, i.e., a concrete mapping, Blickle et al. [1] introduce the concept of *activation* of vertices and edges. The activation of a specification graph's vertex or edge describes its use in the implementation. Since we use hierarchical graphs, we have to define *hierarchical timed activation* or, for short, *hierarchical activation*:

**Definition 4 (hierarchical activation).** *The* hierarchical activation *of a specification graph $G_S = (V_S, E_S)$ is a function $a : \{V_S \cup E_S\} \times T \to \{0, 1\}$ that assigns to each edge $e \in E_S$ and to each vertex $v \in V_S$ the value 1 (activated) or 0 (not activated) at time $t \in T(= \mathbb{R})$.*

Hierarchical activation should support synthesis in such a way that no infeasible implementations are caused by the following rules. Here we summarize the *hierarchical activation rules*:

1. The activation of an interface at time $t$ implies the activation of exactly one associated cluster at the same time:

$$a(\psi, t) = \begin{cases} 1 \text{ if } t \in T^1 \\ 0 \text{ if } t \in T^0 \end{cases} \Leftrightarrow \sum_{\gamma \in \psi.\Gamma} a(\gamma, t) = \begin{cases} 1 \text{ if } t \in T^1 \\ 0 \text{ if } t \in T^0 \end{cases} \qquad (2)$$

2. The activation of a cluster $\gamma$ at time $t$ activates all embedded vertices and edges in $\gamma$:

$$a(\gamma, t) = \begin{cases} 1 \text{ if } t \in T^1 \\ 0 \text{ if } t \in T^0 \end{cases} \Leftrightarrow \forall x \in \{\gamma.V \cup \gamma.\Psi\} : a(x, t) = \begin{cases} 1 \text{ if } t \in T^1 \\ 0 \text{ if } t \in T^0 \end{cases} \qquad (3)$$

3. Each activated edge $e \in E_S$ has to start and to end at an activated vertex. This must hold for all times $t \in T$:

$$a(e, t) = \begin{cases} \{0, 1\} \text{ if } \quad a(v_i, t) = a(v_j, t) = 1 \\ 0 \qquad \text{else} \end{cases} \qquad (4)$$

4. Due to (perhaps implied) timing constraints, the activation of all top-level vertices and interfaces in the problem graph $G_P$ is required, i.e., $\forall t \in T : a(G_P, t) = 1$.

For a given selection of clusters, the hierarchical model can be flattened. With the formalism of hierarchical activation rules, we are able to determine

the overall activation of the specification graph. The result is a non-hierarchical specification.

With the definition of hierarchical activation, we formally define the term *implementation*, where a feasible implementation consists of a feasible *allocation* and a corresponding feasible *binding*.

**Definition 5 (timed allocation).** *A* timed allocation $\alpha(t)$ *of a specification graph* $G_S$ *is the subset of all activated vertices and edges of the problem and architecture graph at time $t$, i.e.,*

$$
\begin{aligned}
\alpha(t) \quad &= \alpha_V(t) \cup \alpha_E(t) \\
\alpha_V(t) &= \{v \in \{V_l(G_P) \cup V_l(G_A)\} \mid a(v,t) = 1\} \\
\alpha_E(t) &= \{e \in E_S \backslash E_M \mid a(e,t) = 1\} \,.
\end{aligned}
$$

$\alpha_V(t)$ denotes the set of activated leaves in the specification graph at time $t$. $\alpha_E(t)$ is the set of activated edges in the problem and architecture graph at time $t$.

**Definition 6 (timed binding).** *A* timed binding $\beta(t)$ *is the subset of all activated mapping edges at time $t$, i.e.,*

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

In order to restrict the combinatorial search space, it is useful to determine the set of feasible timed allocations and feasible timed bindings.

**Definition 7 (feasible timed binding).** *Given a specification graph* $G_S$ *and a timed allocation $\alpha(t)$, a feasible timed binding is a timed binding $\beta(t)$ that satisfies the following requirements:*

1. *Each activated edge $e \in \beta(t)$ starts and ends at a vertex, activated at time $t$, i.e.,*
$$
\forall e = (v, \widetilde{v}) \in \beta(t) : v, \widetilde{v} \in \alpha(t)
$$

2. *For each activated leaf $v \in \{V_l(G_P) \cap \alpha(t)\}$ of the problem graph $G_P$, exactly one outgoing edge $E \in E_M$ is activated at time $t$, i.e.,*
$$
\begin{aligned}
|\{e \in \beta(t) \mid e = (v, \widetilde{v}), v \in \{V_l(G_P) \cap \\
\alpha(t)\} \wedge \widetilde{v} \in V_l(G_A)\}| = 1
\end{aligned}
$$

3. *For each activated edge $e = (v_i, v_j) \in E_P \cap \alpha_E(t)$:*
   − *either both operations are mapped onto the same vertex, i.e.,*
$$
\widetilde{v}_i = \widetilde{v}_j \quad with \quad (v_i, \widetilde{v}_i), (v_j, \widetilde{v}_j) \in \beta(t),
$$

   − *or there exists an activated edge $\widetilde{e} = (\widetilde{v}_i, \widetilde{v}_j) \in \{E_A \cap \alpha_E(t)\}$ to handle the communication associated with edge $e$, i.e.,*
$$
\begin{aligned}
(\widetilde{v}_i, \widetilde{v}_j) &\in \{E_A \cap \alpha_E(t)\} \\
with \quad (v_i, \widetilde{v}_i), &(v_j, \widetilde{v}_j) \in \beta(t).
\end{aligned}
$$

**Definition 8 (feasible timed allocation).** *A* feasible timed allocation *is a timed allocation $\alpha(t)$ that allows at least one feasible timed binding $\beta(t)$ for all times $t$.*

In Figure 2, an infeasible binding would be caused by binding decryption process $P_D^2$ onto the ASIC A and the uncompression process $P_U^1$ onto the FPGA. Since no bus connects the ASIC and the FPGA, there is no way to establish the communication between these processes.

Note that our hierarchical model extends the specification model proposed in [1] by two important features:

1. modeling of alternative refinements in the problem graph (behavior) as well as in the architecture graph (structure)
2. time-variant allocations and bindings.

These major extensions are necessary to model flexibility (reconfigurability) of the behavior (architecture).

With the hierarchical refinements, one has to know exactly what happens if a cluster's execution is interrupted by its own displacement. The request for interchanging clusters while in execution can cause two possible reactions (see also [4]):

1. *safe termination* ensures that a subsystem terminates in a known state without information about the time needed for this.
2. *explicit kill* immediately starts the interchanging process by ignoring the state of computation of a subsystem.

In both cases, the system may become unpredictable. In the following, we neglect the impact of reconfiguration times, context switches, etc.

So far, we have not accounted for system performance. Whether or not the implementation meets the application's performance requirements in terms of throughput (e. g. frames per second) and latencies, depends on the existence of a *feasible schedule*. Although it is possible to schedule any feasible implementation as defined above, the resulting schedule may fail performance requirements. Such scheduling or performance analysis is complex, especially for distributed systems, and is not the scope of this paper. Thus, we do not include a complete analysis in the exploration in Section 4. Rather, we quickly estimate the processor utilization and use the 69% limit as defined in [5] to accept or reject implementations due to performance reasons.

## 3   Definition of Flexibility

With the hierarchical specification model described above, we are able to quantify the amount of implemented functionality. Subsequently, we denote this objective *flexibility*.

For example, consider the problem graph $G_P$ shown in Figure 3. This graph is an extension of the TV decoder example in Figure 1. Here, our goal is to design a Set-Top box family which supports multiple applications. Besides the already known digital TV decoder, there are two more possible applications:
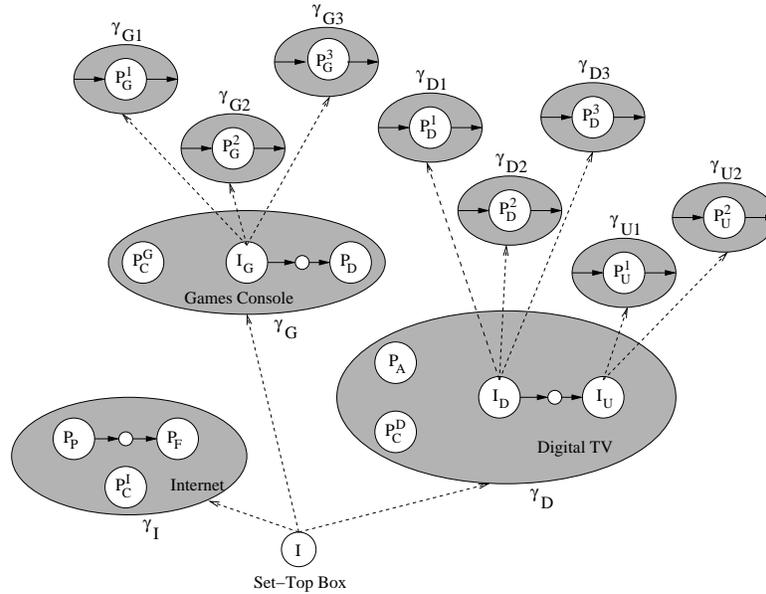
**Figure 3.** Example for System's Flexibility

1. An Internet browser, consisting of a controller process $P_C^I$, parser process $P_P$ for parsing HTML pages and a formatter process $P_F$ for formatting the output.
2. A game console, modeled by a controller process $P_C^G$, the game's core interface $I_G$, and the graphics accelerator $P_D$. The game's core interface can be refined by three different game classes denoted $P_G^1$, $P_G^2$, and $P_G^3$ in Fig. 3. Since the output is constrained to a minimal frame period and the graphic accelerator depends on data produced by the game core process, also the game's core process has to obey some timing constraints.

It should be clear that implementing only one of the three possible applications results in an inflexible system. Furthermore, implementing a digital TV decoder with a great number of decryption algorithms (supported TV stations) is desirable.

So, the basic idea, as stated here, is to enumerate the possible interchanges of implementing clusters in the whole system's problem graph. For example, the flexibility of a trivial system with just one activated interface directly increases with the number of activatable clusters.

The key concepts of *flexibility* are as follows:

– Since each cluster represents an alternative for the same functionality, we know that implementing more clusters for a given interface increases system flexibility in the sense that the system may switch at runtime to select a different cluster.

  – A cluster itself can contain interfaces, which can be implemented with different degrees of flexibility.
  – Although flexibility depends on the implementation, we neglect the impact of the underlying architecture on flexibility, e.g., we do not distinguish whether the flexibility of a system is obtained by the use of either reconfigurable or dedicated hardware components.

With these assumptions, flexibility can be defined as follows:

**Definition 9 (flexibility).** *The flexibility $f_{\mathrm{impl}}$ of a given cluster $\gamma$ is expressed as:*

$$f_{\mathrm{impl}}(\gamma) = a^+(\gamma) \cdot \begin{cases} \left[ \sum_{\psi \in \gamma.\Psi} \sum_{\widehat{\gamma} \in \psi.\Gamma} f_{\mathrm{impl}}(\widehat{\gamma}) \right] \\ -(|\gamma.\Psi| - 1) \quad for \quad \gamma.\Psi \neq \emptyset \\ 1 \hspace{3.2cm} otherwise \end{cases}$$

*where the term $a^+(\gamma)$ describes the activation of the cluster $\gamma$ in the future. If cluster $\gamma$ will be selected at any time in the future then $a^+(\gamma) = 1$. Otherwise $a^+(\gamma) = 0$, meaning it will not be implemented at all.*

*In other words: The flexibility of a cluster $\gamma$, if ever activated, is calculated by the sum of all its interfaces' flexibilities minus the number of its interfaces less 1, and 1 if there is no interface in the given cluster. The flexibility of an interface is the sum of flexibilities of all its associated clusters. If a cluster will never be activated, its flexibility is 0.*

The flexibility $f(G_{\mathrm{P}})$ of this problem graph shown in Figure 3 is computed as follows:

$$\begin{aligned} f(G_{\mathrm{P}}) &= a^+(G_{\mathrm{P}}) \cdot [f(\gamma_{\mathrm{I}}) + f(\gamma_{\mathrm{G}}) + f(\gamma_{\mathrm{D}})] \\ &= a^+(G_{\mathrm{P}}) \cdot [a^+(\gamma_{\mathrm{I}}) + a^+(\gamma_{\mathrm{G}}) \cdot [a^+(\gamma_{\mathrm{G1}}) + \\ &\quad a^+(\gamma_{\mathrm{G2}}) + a^+(\gamma_{\mathrm{G3}})] + a^+(\gamma_{\mathrm{D}}) \cdot \\ &\quad [a^+(\gamma_{\mathrm{D1}}) + a^+(\gamma_{\mathrm{D2}}) + a^+(\gamma_{\mathrm{D3}}) + \\ &\quad a^+(\gamma_{\mathrm{U1}}) + a^+(\gamma_{\mathrm{U2}}) - 1]] \end{aligned}$$

Based on this equation, the system's flexibility is obtained by specifying the utilization of each cluster $\gamma$ in the future, denoted by $a^+(\gamma)$. If all clusters can be activated in future implementations, system's flexibility calculates to $f(G_{\mathrm{P}}) = 8$. This is also the maximal flexibility $f_{\mathrm{max}}$. If, e.g., cluster $\gamma_{\mathrm{G}}$ is not used in future implementations the flexibility will decrease to $f(G_{\mathrm{P}}) = 5$.

For the sake of simplicity, we have omitted the architecture graph and the mapping edges. Obviously, a cluster only contributes to the total flexibility if it is bindable as per Def. 7. A more sophisticated definition of flexibility can established by using weighted sums in Definition 9. The weight associated with each cluster shows the cluster's inherent functionality.

## 4   Design Space Exploration

Because of the accepted use of tools on lower design levels of abstraction, exploration becomes the next step in order to prevent under- or over-designing

a system. Typically, a system has to meet many constraints and should optimize many different design objectives and constraints simultaneously such as execution time, cost, area, power consumption, weight, etc.

A single solution that optimizes all objectives simultaneously is very unlikely to exist. Instead, it should be possible to first explore different optimal solutions or approximations, and subsequently select and refine one of those solutions.
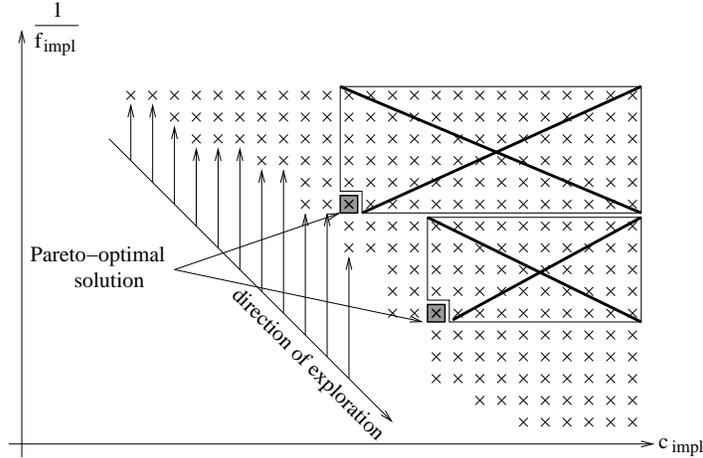


**Figure 4.** Flexibility/Cost-Design Space

### 4.1   The Flexibility/Cost-Design-Space and the Optimization Goal

In this paper, we consider the two objectives *flexibility* $f_{\mathrm{impl}}(\alpha(t))$, as described in Section 3, and *cost* $c_{\mathrm{impl}}(\alpha(t))$. Here we use the so-called *allocation cost model* $c_{\mathrm{impl}}(\alpha(t))$, where $c_{\mathrm{impl}}(\alpha(t))$ is the sum of all realization cost of resources in the allocation $\alpha(t)$.

Figure 4 shows a typical tradeoff-curve between cost and the reciprocal value of flexibility. As already mentioned we are concerned with a MOP (Multiobjective Optimization Problem). Our MOP consists of two objective functions $c_{\mathrm{impl}}(\alpha(t))$ and $\frac{1}{f_{\mathrm{impl}}(\alpha(t))}$, where the parameter $\alpha(t)$ is the decision vector. The optimization goal is to minimize $c_{\mathrm{impl}}(\alpha(t))$ and $\frac{1}{f_{\mathrm{impl}}(\alpha(t))}$ simultaneously, i.e., to maximize system's flexibility for minimal cost implementations.

**Definition 10 (feasible set).** *Let allocation $\alpha(t)$ denote our decision vector. The* feasible set $\mathbf{X}_f$ *is defined as the set of allocations $\alpha(t)$ that satisfy the definition for feasible allocation (Def. 8) for all times $t \in T$.*

**Definition 11 (Pareto-optimality).** *For any two decision vectors $\mathbf{a}(t), \mathbf{b}(t) \in \mathbf{X}_f$, $\mathbf{a}(t)$ dominates $\mathbf{b}(t)$ iff $\left( c_{\mathrm{impl}}(\mathbf{a}(t)) < c_{\mathrm{impl}}(\mathbf{b}(t)) \wedge \frac{1}{f_{\mathrm{impl}}(\mathbf{a}(t))} \leq \frac{1}{f_{\mathrm{impl}}(\mathbf{b}(t))} \right) \vee$*

$\left( c_{\mathrm{impl}}(\mathbf{a}(t)) \leq c_{\mathrm{impl}}(\mathbf{b}(t)) \wedge \frac{1}{f_{\mathrm{impl}}(\mathbf{a}(t))} < \frac{1}{f_{\mathrm{impl}}(\mathbf{b}(t))} \right)$. *A decision vector* $\mathbf{x}(t) \in \mathbf{X}_f$ *is said to be* non-dominated *with respect to a set* $\mathbf{A} \subseteq \mathbf{X}_f$ *iff* $\nexists \mathbf{a}(t) \in \mathbf{A} : \mathbf{a}(t)$ *dominates* $\mathbf{x}(t)$. *Moreover,* $\mathbf{x}(t)$ *is said to be* Pareto-optimal *iff* $\mathbf{x}(t)$ *is non-dominated regarding* $\mathbf{X}$ *(see also [6]).*

Figure 4 shows two Pareto-optimal design points. The goal of design space exploration is to find *all* Pareto-optimal design points that also fulfill all timing requirements. The points in Figure 4 represent possible solutions, where not every solution has to be feasible in the sense of Def. 7, and not every feasible solution has to meet the timing requirements. If we have found a Pareto-optimal solution $x$ that meets all requirements, the class of all design points dominated by $x$ can be pruned. This is shown in Figure 4 by boxes. In the following, we will introduce an algorithm for efficiently exploring flexibility/cost-tradeoff-curves.

## 4.2   The Exploration Algorithm

Figure 4 shows the general distribution of design points. At this stage, we do not distinguish between feasible and non-feasible solutions. Our objective is to find all Pareto-optimal solutions that meet all timing constraints. At a glance, a good strategy for design space exploration is to investigate each design point at the front, where we use the order of increasing implementation cost (direction of exploration). If a point represents an infeasible implementation or it misses some performance requirements, we discard it and pick up the next one on the front.

The problem of this idea is, that the set of possible implementations is unknown. A possible modification of this strategy would be to determine all points in advance, i.e., to determine all possible $2^{|V_S|}$ activations. Since binding is a NP-complete problem (see [1]), this exhaustive search approach seems not to be a viable solution.

To avoid superfluous computation of non-Pareto-optimal solutions, we propose two methods for search space reduction:

1. **Possible Resource Allocations**. A *possible resource allocation* is a partial allocation of resources in the architecture graph which allows the implementation of *at least one* feasible problem graph activation by neglecting the feasibility of binding first. Usually, we have to investigate all $2^{|V_S|}$ design points. But only the elements covering a possible resource allocation represent meaningful activations in the sense that at least a required minimum of problem graph vertices is bindable.
2. **Flexibility Estimation**. With the possible resource allocations we are able to sort the remaining design points by increasing cost. If we inspect the elements of this sorted list by increasing cost, a new calculated solution is Pareto-optimal, iff it possesses a greater flexibility than each solution that has been already implemented, as defined in Def. 11.

As shown in our case study (see Section 5), by using these two techniques, we dramatically reduce the invocations of the solver for the NP-complete binding problem.

With our approach of hierarchical specification and activations, we are able to first determine the set of possible resource allocations: For each vertex $v_i$ inside a given cluster $\gamma_j$, we determine the set $R_{ij}$ of reachable resources. A resource $r$ is reachable if a mapping edge between $v_i$ and $r$ exists. Derived from the hierarchical activation rules, only leaves $v \in G_A.V$ of the top-level architecture graph or whole clusters of the architecture graph are considered. Next, we set up the outer conjunction $R_j$ of all power sets $2^{R_{ij}}$. Consequently, the set $R_j$ describes *all* combinations of resource activations for implementing the non-hierarchical vertices $v \in \gamma_j.V$ of cluster $\gamma_j$ by ignoring the feasibility of binding.

Finally, we have to inspect all hierarchical components $\gamma_j.\Psi$ of cluster $\gamma_j$. Since all clusters associated with an interface $\psi \in \gamma_j.\Psi$ represent alternative refinements of $\psi$, we compute the union of possible resource allocations for the associated clusters.

The elements of the resulting set are the possible resource allocations, which we sort by increasing implementation cost. The algorithm PRA to determine this set of **P**ossible **R**esource **A**llocations $\mathcal{A}(\gamma)$ for a given cluster $\gamma$ is listed below:

PRA
    IN:     specification graph $G_S$
    IN:     current cluster $\gamma_{\mathrm{cur}}$
    OUT:  set of possible resource allocations $\mathcal{A}$
    BEGIN
      $\mathcal{A} = \emptyset$
      FOR each vertex $v \in \gamma_{\mathrm{cur}}.V$ DO
        $\mathcal{A}_v = \{v_a \mid (v_i, v_a) \in E_M \wedge v_a \in V_A\}$
        $\mathcal{A}_\gamma = \{\gamma_a \mid (v, v_a) \in E_M \wedge v_a \in \gamma_a.V\}$
        $\mathcal{A} = \mathcal{A} \times 2^{\mathcal{A}_v \cup \mathcal{A}_\gamma} \backslash \epsilon$
      ENDFOR
      FOR each interface $\psi \in \gamma_{\mathrm{cur}}.\Psi$ DO
        $\mathcal{X} = \emptyset$
        FOR each cluster $\gamma \in \psi.\Gamma$ DO
          $\mathcal{X} = \mathcal{X} \cup \mathsf{PRA}(\gamma)$
        ENDFOR
        $\mathcal{A} = \mathcal{A} \times \{2^{\mathcal{X}} \backslash \epsilon\}$
      ENDFOR
    END

Here, $\epsilon$ denotes the element representing the empty set $\{\emptyset\}$. For example, the set $\mathcal{A}$ of possible resource allocations for the specification given in Figure 2 computes to:

$$\mathcal{A} = \{\mu P, \mu PC_1, \mu PC_2, \mu PC_1 C_2, \mu PD_3, \mu PU_2,$$
$$\mu PC_1 D_3, \mu PC_2 D_3, \mu PC_1 U_2, \mu PC_2 U_2,$$
$$\mu PC_1 C_2 D_3, \dots, \mu PC_1 C_2 D_1 U_2 A\}$$

The elements of the ordered set of possible resource allocations are inspected in ascending order of their allocation cost $c_{\mathrm{impl}}$ (see Fig. 4). For each possible

resource allocation, we remove all resources that are not activated from the architecture graph $G_A$. By removing these elements, also mapping edges are removed from the specification graph. Next, we delete all vertices in the problem graph with no incident mapping edge. This results in a reduced specification graph.

In order to avoid superfluous computation of non-Pareto-optimal solutions, we use a lower bound to restrict our search space: With Definition 9, the maximal flexibility $f_{max}$ of the reduced specification graph can be calculated. Since we explore flexibility/cost-objective-space by increasing cost (see Figure 4), we are only interested in design points with a greater flexibility than already implemented. So we use the already maximal implemented flexibility as lower bound for pruning the search space. With the known maximal implemented flexibility, we therefore may skip specifications with a lower implementable flexibility. Only for specifications with greater expected flexibility, we try to construct a feasible implementation next.

Generally, more than one activatable cluster for a problem graph's interface remains in the specification graph. Consequently, we have to identify so-called *elementary cluster-activations*, which are defined as follows. Let $\Gamma_{act}$ denote the set of activatable clusters which is computed by traversing the problem graph and checking the existing of at least one mapping edge per leaf. Only if all leaves are incident to at least one mapping edge, the cluster is meant to be activatable. An elementary cluster-activation $ecs$ is a set $ecs = \{\gamma_i \mid \gamma_i \in \Gamma_{act}\}$, where exactly one cluster is selected per activated interface. Since every activatable cluster has to be part of the implementation to obtain the expected flexibility, we have to determine a coverage [7] of $\Gamma_{act}$ by elementary cluster-activations.

Given an elementary cluster-activation, we can select these clusters for implementation. Furthermore, we must determine valid cluster activations in the architecture graph, e.g., it is possible that two problem graph clusters are mapped on different configurations of the same FPGA. So, we have to guarantee that each elementary cluster-activation can be bound to a non-ambiguous architecture, i.e., there is exactly one activated cluster for every activated interface in the architecture graph, e.g., one activated configuration for an FPGA.

Finally, we validate all timing constraints that are imposed on our implementation. Here, we use a statistical analysis method to check for fulfillment. Only if a new calculated implementation

1. is feasible as defined in Def. 7,
2. possesses a greater flexibility as already implemented, and
3. obeys all performance constraints,

it is Pareto-optimal.

With these basic ideas of pruning the search space, we formulate our exploration algorithm based on a branch-and-bound strategy [7,8]. For the sake of clarity, we omit details for calculating a coverage of activatable problem graph clusters or successive flexibility estimation, etc. The following code should be self-explanatory with the previous comments.

```
EXPLORE
   IN:      specification graph G_S
   OUT:   Pareto-optimal set O
   BEGIN
      f_cur = 0
      A = G_S.possibleResourceAllocations()
      f_max = G_S.computeMaximumFlexibility()
      FOR each candidate a ∈ A DO
         f = a.computeMaximumFlexibility()
         WHILE f_cur < f_max THEN
            α = G_S.computeAllocation(a)
            β = G_S.computeBinding(α)
            i = new Implementation(α, β)
            IF i.isFeasibleImplementation() THEN
               IF i.meetsAllConstraints() THEN
                  IF i.flexibility() > f_cur THEN
                     O = O ∪ i
                     f_cur = i.flexibility()
                  ENDIF
               ENDIF
            ENDIF
         ENDWHILE
      ENDFOR
   END
```

In the worst case, this algorithm is not better than an exhaustive search algorithm. But, a typical search space with $10^5$-$10^{12}$ design points can be reduced by the EXPLORE-algorithm to a few $10^3$-$10^4$ possible resource allocations. Since we only try to implement design points with a greater expected flexibility than the already implemented flexibility, again, only a small fraction of these point has to be taken into account, typically less than 100.

## 5   Case Study

In our case study we investigate the specification of our Set-Top box depicted in Figure 5. Again, we increased the complexity of our example. The architecture graph is now composed of two processors ($\mu P_1$ and $\mu P_2$), three ASICs ($A_1$ to $A_3$), and an FPGA. The ASICs are used to improve performance for the decryption, uncompression, game's core, and graphic acceleration processes. The FPGA can also be used as coprocessor for the third decryption, the second uncompression, or the first game core class. The allocation cost of each component are annotated in Fig. 5.

In Figure 5, we have omitted the mapping edges. Possible mappings and respective core execution times are given in Table 1. Furthermore, we assume that all communications can be performed on every resource. No latencies for
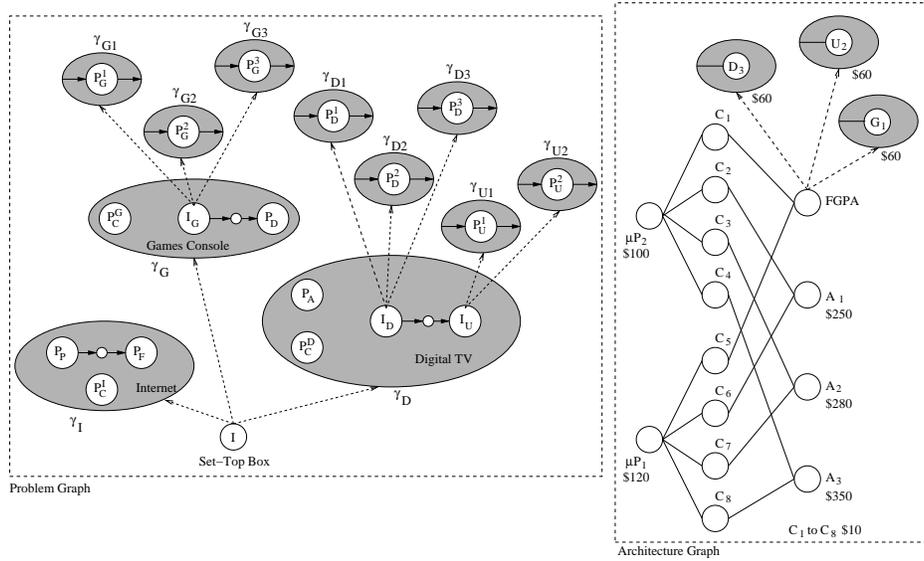
**Figure 5.** Specification of a Set-Top Box

external communications are taken into account. Timing constraints for the game console and digital TV are given by the minimal periods of the output processes ($P_D$, $P_U^1$, and $P_U^2$). $P_D$ has to be executed every 240 ns. The output for the digital TV box is less restrictive: $P_U^1$ and $P_U^2$ should be executed at least every 300 ns if activated.

As described above, our algorithm starts with the determination of the set of all possible resource allocations. Here, elements that are obviously not Pareto-optimal or no feasible implementations are left out, e. g., all combinations of a single functional component and an arbitrary number of communication resources. The beginning of the ordered subset $\mathcal{A}$ of possible resource allocations is given by:

$$\begin{aligned} \mathcal{A} = \{&\mu P_2, \mu P_1, \mu P_2 D_3 C_1, \mu P_2 U_2 C_1, \mu P_2 G_1 C_1, \\ &\mu P_1 D_3 C_5, \mu P_1 U_2 C_5, \mu P_1 G_1 C_5, \mu P_2 D_3 U_2 C_1, \\ &\mu P_2 D_3 G_1 C_1, \mu P_2 U_2 G_1 C_1, \mu P_1 D_3 U_2 C_5, \\ &\mu P_1 D_3 G_1 C_5, \mu P_1 U_2 G_1 C_5, \mu P_1 \mu P_2, \dots\} \end{aligned}$$

Next, we determine all elementary cluster activations that can be activated under the given resource allocation. For the first resource allocation ($\mu P_2$), we find the elementary cluster activations $\gamma_I$, $\gamma_{G1}$, and $\gamma_{D1}\gamma_{U1}$. The estimated flexibility as defined by Def. 9 calculates to $f_{impl} = 3$. Since our already implemented flexibility is 0 (there is no feasible implementation yet), we try to find feasible implementations for the given elementary cluster activations. With Figure 5 and Table 1, we are able to find a feasible allocation and binding for all elementary cluster activations.

**Table 1.** Possible Mappings in Figure 5

| Process | $\mu P_1$ | $\mu P_2$ | $A_1$ | $A_2$ | $A_3$ | D3 | U2 | G1 |
|---------|-----------|-----------|-------|-------|-------|------|------|------|
| $P_C^I$ | $10ns$ | $12ns$ | - | - | - | - | - | - |
| $P_P$ | $15ns$ | $19ns$ | - | - | - | - | - | - |
| $P_F$ | $50ns$ | $75ns$ | - | - | - | - | - | - |
| $P_C^G$ | $25ns$ | $27ns$ | - | - | - | - | - | - |
| $P_G^1$ | $75ns$ | $95ns$ | $15ns$ | $15ns$ | $15ns$ | - | - | $20ns$ |
| $P_G^2$ | - | - | $25ns$ | $22ns$ | $22ns$ | - | - | - |
| $P_G^3$ | - | - | $50ns$ | $45ns$ | $35ns$ | - | - | - |
| $P_D$ | $70ns$ | $90ns$ | $30ns$ | $30ns$ | $25ns$ | - | - | - |
| $P_C^D$ | $10ns$ | $10ns$ | - | - | - | - | - | - |
| $P_A$ | $55ns$ | $60ns$ | - | - | - | - | - | - |
| $P_D^1$ | $85ns$ | $95ns$ | $25ns$ | $22ns$ | $22ns$ | - | - | - |
| $P_D^2$ | - | - | $35ns$ | $33ns$ | $32ns$ | - | - | - |
| $P_D^3$ | - | - | - | - | - | $63ns$ | - | - |
| $P_U^1$ | $40ns$ | $45ns$ | $15ns$ | $12ns$ | $10ns$ | - | - | - |
| $P_U^2$ | - | - | $29ns$ | $27ns$ | $22ns$ | - | $59ns$ | - |

Next, we have to check all timing constraints. Therefore, we define a maximal processor utilization of 69%. If the estimated utilization exceeds this upper bound, we reject the implementation as infeasible. Since the Internet browser need not meet any timing constraints, this particular implementation is indeed feasible.

For the validation of the digital TV application, we need some more information. As we know the timing constraint imposed on the uncompression and the decryption process, we only need information about how often the authentification and controller processes are executed. The execution of the authentification is scheduled once at system start up. Statistically, the controller process makes up about 0.01% of all process calls in the digital TV application. So, we neglect the authentification and controller process in our estimation. For fulfillment of the performance constraint, the sum of the core execution times of process $P_D^1$ and $P_U^1$ ($95ns + 45ns$) must be less than $0.69 \cdot 300ns$. Evidently, this constraint is met.

Unfortunately, we have to reject the implementation of the application of the game console violating the upper utilization bound ($95ns + 90ns \nleq 0.69 \cdot 240ns$). So our implemented flexibility calculates to $f_{\mathrm{impl}} = 2$ which is still greater than the already implemented flexibility.

Now, we continue with the next possible resource allocation, i.e., $\mu P_1$. Due to space limitations, we only present the results. The set of Pareto-optimal solutions for this example is shown in Table 2. At the beginning, our search space consisted of $2^{25}$ design points. By calculating the set of possible resource allocations, this design space was reduced to $2^{14}$ design points. That is, by traversing our specification graph and setting up one boolean equation we are able to reject about 99.9% of our design points as non-Pareto-optimal. After investigating approx. 7000 design points, we have found all 6 Pareto-optimal solutions. For

**Table 2.** Pareto-Optimal Solutions

| Resources | Clusters | $c$ | $f$ |
|---|---|---|---|
| $\mu P_2$ | $\gamma_I, \gamma_{D1}, \gamma_{U1}$ | \$100 | 2 |
| $\mu P_1$ | $\gamma_I, \gamma_{G1}, \gamma_{D1}, \gamma_{U1}$ | \$120 | 3 |
| $\mu P_2, G_1, U_2, C_1$ | $\gamma_I, \gamma_{G1}, \gamma_{D1}, \gamma_{U1}, \gamma_{U2}$ | \$230 | 4 |
| $\mu P_2, D_3, G_1, U_2, C_1$ | $\gamma_I, \gamma_{G1}, \gamma_{D1}, \gamma_{D3},$ $\gamma_{U1}, \gamma_{U2}$ | \$290 | 5 |
| $\mu P_2, A_1, C_2$ | $\gamma_I, \gamma_{G1}, \gamma_{G2}, \gamma_{G3},$ $\gamma_{D1}, \gamma_{D2}, \gamma_{U1}, \gamma_{U2}$ | \$360 | 7 |
| $\mu P_2, A_1, D_3, C_1, C_2$ | $\gamma_I, \gamma_{G1}, \gamma_{G2}, \gamma_{G3},$ $\gamma_{D1}, \gamma_{D2}, \gamma_{D3}, \gamma_{U1}, \gamma_{U2}$ | \$430 | 8 |

these design points, we estimated the implementable flexibility by solving a single boolean equation. In only approx. 1050 cases (0.0032% of the original search space) the estimated flexibility was greater than the already implemented flexibility. Only for these points, we needed to try to construct an implementation. Hence, our exploration algorithm typically prunes the search space so much that industrial size applications can be efficiently explored within minutes.

## Conclusions and Future Work

Based on the concept of hierarchical graphs, we have introduced a formal definition of system flexibility. Furthermore, an algorithm for exploring the flexibility/cost design space was presented. Due to the underlying branch-and-bound strategy, we are able to prune about 99.9% of a typical search space, while still finding all Pareto-optimal implementations. Hence, industrial size applications can be explored efficiently.

In the future, we would like to extend the proposed approach by an exact scheduling method to check performance constraints. In [9], first results in scheduling hierarchical dataflow graphs are presented.

## References

1. Blickle, T., Teich, J., Thiele, L.: System-Level Synthesis Using Evolutionary Algorithms. In Gupta, R., ed.: Design Automation for Embedded Systems. Number 3. Kluwer Academic Publishers, Boston (1998) 23–62
2. Cadence: Virtual Component Co-design (VCC). (2001) http://www.cadence.com.
3. Chatha, K.S., Vemuri, R.: MAGELLAN: Multiway Hardware-Software Partitioning and Scheduling for Latency Minimization of Hierarchical Control-Dataflow Task Graphs. In: Proc. CODES'01, Ninth International Symposium on Hardware/Software Codesign, Copenhagen, Denmark (2001)
4. Richter, K., Ziegenbein, D., Ernst, R., Thiele, L., Teich, J.: Representation of Function Variants for Embedded System Optimization and Synthesis. In: Proc. 36th Design Automation Conference (DAC'99), New Orleans, U.S.A. (1999)

5. Liu, C.L., Layland, J.W.: Scheduling Algorithm for Multiprogramming in a Hard-Real-Time Environment. Journal of the ACM **20** (1973) 46–61
6. Pareto, V.: Cours d'Économie Politique. Volume 1. F. Rouge & Cie., Lausanne, Switzerland (1896)
7. Hachtel, G.D., Somenzi, F.: Logic Synthesis and Verification Algorithms. 2 edn. Kluwer Academic Publishers, Norwell, Massachusetts 02061 USA (1998)
8. Micheli, G.D.: Synthesis and Opimization of Digital Circuits. McGraw-Hill, Inc., New York (1994)
9. Bhattacharya, B., Bhattacharyya, S.: Quasi-static Scheduling of Reconfigurable Dataflow Graphs for DSP Systems. In: Proc. of the International Conference on Rapid System Prototyping, Paris, France (2000) 84–89