UvA-DARE (Digital Academic Repository)

Scenario-Based Design Space Exploration

Pimentel, A.; van Stralen, P.

DOI
[10.1007/978-94-017-7267-9_10](10.1007/978-94-017-7267-9_10)

Publication date
2017

Document Version
Final published version

Published in
Handbook of Hardware/Software Codesign

License
Article 25fa Dutch Copyright Act

Link to publication

Scenario-Based Design Space Exploration

9

Andy Pimentel and Peter van Stralen

Abstract

Modern embedded systems are becoming increasingly multifunctional, and, as a consequence, they more and more have to deal with dynamic application workloads. This dynamism manifests itself in the presence of multiple applications that can simultaneously execute and contend for resources in a single embedded system as well as the dynamic behavior within applications themselves. Such dynamic behavior in application workloads must be taken into account during the early system-level Design Space Exploration (DSE) of Multiprocessor System-on-Chip (MPSoC)-based embedded systems. *Scenario-based DSE* utilizes the concept of application scenarios to search for optimal mappings of a multi-application workload onto an MPSoC. To this end, scenario-based DSE uses a multi-objective genetic algorithm (GA) to identify the mapping with the best average quality for all the application scenarios in the workload. In order to keep the exploration of the scenario-based DSE efficient, fitness prediction is used to obtain the quality of a mapping. This fitness prediction implies that instead of using the entire set of all possible application scenarios, a small but representative subset of application scenarios is used to determine the fitness of mapping solutions. Since the representativeness of such a subset is dependent on the application mappings being explored, these representative subsets of application scenarios are dynamically obtained by means of coexploration of the scenario subset space. In this chapter, we provide an overview of scenario-based DSE and, in particular, present multiple techniques for fitness prediction using representative subsets of application scenarios: a stochastic, deterministic, and hybrid combination.

A. Pimentel (✉)
University of Amsterdam, Amsterdam, The Netherlands
e-mail: a.d.pimentel@uva.nl

P. van Stralen
Philips Healthcare, Best, The Netherlands
e-mail: peter.van.stralen@philips.com

Acronyms	
ASIC	Application-Specific Integrated Circuit
ASIP	Application-Specific Instruction-set Processor
DSE	Design Space Exploration
ESL	Electronic System Level
FIFO	First-In First-Out
FS	Feature Selection
GA	Genetic Algorithm
JPEG	Joint Photographic Experts Group
KPN	Kahn Process Network
MJPEG	Motion JPEG
MoC	Model of Computation
MPSoC	Multi-Processor System-on-Chip
SBS	Sequential Backward Selection
SFS	Sequential Forward Selection

Contents

9.1   Introduction

To cope with the design complexities of *Multi-Processor System-on-Chip* (MPSoC)-based embedded systems [15], Electronic System Level (ESL) design [6, 12] has become a promising approach for raising the abstraction level of design and thereby increasing the design productivity. Early design space exploration (DSE) is an important ingredient of such ESL design, which has received significant research attention in recent years [8, 10, 19]. The majority of all these DSE efforts still evaluates and explores MPSoC architectures under single-application workloads. This is, however, increasingly unrealistic since modern embedded devices, especially in the consumer electronics domain, are nowadays highly multifunctional and feature dynamic application workloads. For example, a mobile

phone has become a multimedia device that is not only used for calling, but it is also connected to the Internet and has become a decent camera. With the increased capabilities of current smartphones, they have almost the same possibilities as desktop computers. Another trend is to make consumer devices "smart." Smart digital cameras are able to directly share photos on the Internet. Photos can be edited on the camera and can also be tagged with a GPS location. Similarly, smart televisions also enhance the functionality of the television as they not only show an incoming video stream from a decoder but can, e.g., also show photos from a memory card or install additional applications. This trend of smartness does not only increase the number of applications but also the dynamism of the application workloads on these embedded systems. For old analogue televisions with CRT, the characteristics of the incoming video stream were exactly known: the size of the frames, the frame rate, etc. Currently, however, these streams become more dynamic: High-definition (HD) television may, for example, have varying frame rates and frame sizes. Additionally, 3D television may double the number of frames that need to be decoded.

This chapter will therefore use the concept of application scenarios [7, 17] to introduce *scenario-based DSE* [28, 30]. Application scenarios are able to describe the dynamism of embedded applications and the interaction between the different applications on the embedded system. The concept of application scenarios is illustrated in Fig. 9.1. An application scenario consists of two parts: an inter- and an intra-application scenario. An *inter-application scenario* describes the interaction between multiple applications, i.e., which applications are concurrently executing at a certain moment in time. Inter-application scenarios can be used to prevent the overdesign of a system. If some of the applications cannot run concurrently, then there is no need of reserving resources for the situation where these applications are running together. *Intra-application scenarios*, on the other hand, describe the different execution modes (or operation modes) for each individual application. In the example application scenario in Fig. 9.1, the left-hand side shows the selected
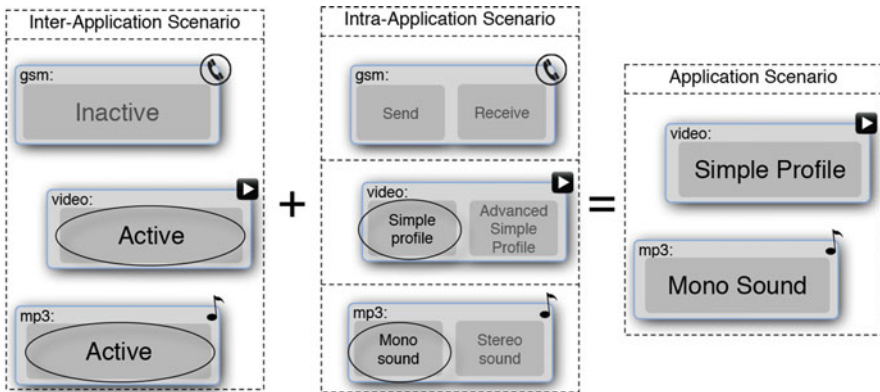

Fig. 9.1 An illustration of application scenarios

inter-application scenario. In this case, the video and the MP3 applications are active, while the GSM application is inactive. In the middle, the intra-application scenarios are shown. The video application can, for example, decode video using a simple profile and an advanced simple profile. For the intra-application scenario, it is decided to decode video using a simple profile and to play mono music with the MP3 application. As the GSM is inactive, no operation mode needs to be selected for the GSM application. Hence, the application scenario is the sum of the inter- and intra-application scenarios: the video application is decoding using a simple profile, and the MP3 application is playing music in a mono sound.

9.2   Application Dynamism

To illustrate the consequences of dynamic application behavior in terms of extra-functional aspects (like system performance and power consumption), this section presents a small, motivational case study in which a Motion JPEG (MJPEG) decoder application – with different intra-application scenarios – is mapped onto a heterogeneous bus-based MPSoC architecture with four processors and a single shared memory. First, we have randomly picked three mappings of the MJPEG application on our bus-based architecture. For each of these mappings, we used the Sesame system-level MPSoC simulation framework [3, 18] to determine the fitness values (in this case, execution time and power consumption) for each *individual intra-application scenario*. The resulting fitness values of these mappings are shown in Fig. 9.2, where the horizontal and vertical axes refer to execution time and
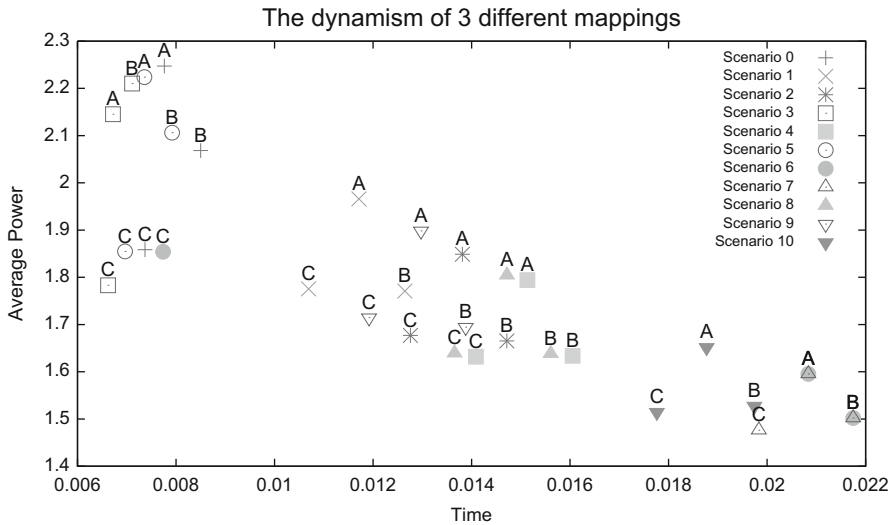

Fig. 9.2 An illustration of the dynamism of different scenarios. For three different mappings, the fitness of each individual scenario is shown

power consumption, respectively. These values are only used to compare different mappings and intra-application scenarios. Therefore, they do not have a unit. In this graph, the letters A–C are the different mappings, whereas 0–10 are the different intra-application scenarios of the MJPEG application.

Irrespective of the mapping, scenario 7 is the intra-application scenario that consumes the least amount of power. The scenario with the highest power consumption, on the other hand, depends on the mapping. For mappings A and C, scenario 0 has the highest power consumption. In case of mapping B, however, scenario 3 has the highest power consumption. To explain this behavior, both the scenarios and the mappings must be analyzed. Scenario 3 involves the decoding of a frame that has a much higher compression ratio than scenario 0. As a consequence, scenario 3 requires less communication than scenario 0. Moreover, for mapping B, the shared bus is fully utilized, while in mappings A and C, there is still some capacity left on the shared bus. As a result, the reduction in communication between scenarios 0 and 3 has more effect on the execution time of mapping B than it has on mappings A and C. Although the consumed energy for scenario 3 is lower than the consumed energy for scenario 0 for all of the three mappings, the larger difference in execution time results in higher power consumption for mapping B.

Figure 9.2 also shows other interesting behavior with respect to scenario 6. For example, scenario 6 with mappings A and B has the same fitness as scenario 7. For mapping C, however, the execution time of scenario 6 is much lower than for scenario 7. Without going into details, scenario 6 would lead to the conclusion that mapping C consumes more power than mapping A. This conclusion contrasts with the conclusion that can be drawn from comparing the power consumption using the other scenarios (i.e., the power consumption of mapping C is lower than mapping A). In a sense, scenario 6 gives a deceiving view of the quality-ordering relation between the different mappings.

To provide more insight into this problem, the potential Pareto dominance relations between the mappings of the experiment illustrated in Fig. 9.2 are listed in Table 9.1. See ► Chaps. 6, "Optimization Strategies in Design Space Exploration"

Table 9.1 The Pareto dominance relations comparing the mappings from Fig. 9.2 for each individual scenario. The symbol || stands for incomparable fitness values

Scenario	$A \ldots B$	$A \ldots C$	$B \ldots C$	Front
0	||	>	>	C
1	||	>	||	B, C
2	||	>	||	B, C
3	$\leq$	>	>	C
4	||	>	>	C
5	||	>	>	C
6	||	||	||	A, B, C
7	||	>	>	C
8	||	>	||	B, C
9	||	>	||	B, C
10	||	>	>	C

and ▶ 7, "Hybrid Optimization Techniques for System-Level Design Space Exploration" for a discussion on Pareto dominance. In the three columns in the middle of the list in Table 9.1, the unique mapping comparisons are shown: mapping A versus B, mapping A versus C, and mapping B versus C. Next, for each individual intra-application scenario, the fitness values for the different mappings are compared. In this way, three different types of relations are obtained: (1) a mapping is equal to or fully dominates the other mapping ($\leq$), (2) a mapping is dominated by another mapping ($>$), and (3) the mappings are not comparable using the Pareto dominance relation ($\|$). Finally, the last column shows the Pareto front based on the fitness values of the specific intra-application scenario.

As a first observation, one can see that for none of the relations, it is the case that all scenarios fully agree on the type of the relation. For the first two relations (where mapping A is compared with mappings B and C), only one scenario differs with respect to the relation type. In case of the comparison between mapping A and mapping B, the fitness values for most of the scenarios determine that mapping A is incomparable with mapping B. Only the fitness values of scenario 3 lead to a different conclusion: mapping A dominates mapping B. Similarly, mapping C dominates mapping A for most of the intra-application scenarios. Only the fitness values of scenario 6 are incomparable for mappings A and C.

The problem arises with the relation between mapping B and mapping C. Judging on 6 out of the 11 scenarios, mapping B is better than mapping C. Based on the other five scenarios, however, one comes to the conclusion that mapping B is incomparable with mapping C. These kinds of uncertainties complicate the scenario-based DSE. The DSE ends up with a Pareto front, but not all the intra-application scenarios agree on what the Pareto front should be. In the example of Table 9.1, three different Pareto fronts are observed, from which the front with only mapping C is the most common.

Based on the most common Pareto front with only mapping C, one could conclude that the set $\{0, 3, 4, 5, 7, 10\}$ of intra-application scenarios is *representative* for the MJPEG application. This representativeness, however, is completely dependent on which mappings are evaluated. In case only mappings A and B would have been taken into account, intra-application scenario 3 would have been interpreted as an unrepresentative scenario. However, if this scenario 3 is excluded for the comparison between mapping B and mapping C, there is no majority anymore for one of the Pareto dominance relation types. In the next section, which introduces our scenario-based DSE framework, we will explain how we deal with the above problem.

9.3 Scenario-Based DSE Framework

Conceptually, scenario-based DSE [28, 30] is an exploration technique for embedded systems with a dynamic multi-application workload. In this chapter, an exploration framework for scenario-based DSE is presented that aims to provide a *static mapping* of a multi-application workload onto an MPSoC. The mapping is to be used during the entire lifetime of an embedded system. Consequently, the
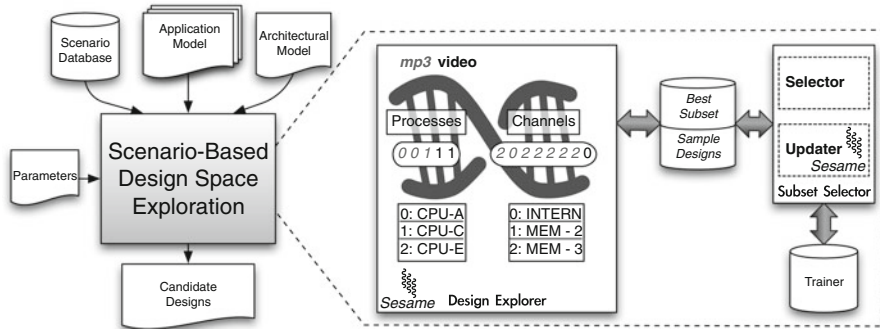
Fig. 9.3 The exploration framework for scenario-based DSE

average behavior of the designed MPSoC must be as good as possible for all the different application scenarios. Currently, we assume an equal likelihood for each application scenario. The approach, however, can easily be generalized to different probability distributions.

In this work, we assume that a multi-application workload mapping implicitly consists of two aspects: (1) *allocation* and (2) *binding*. The allocation selects the architectural components for the MPSoC platform architecture. Only the selected components will be deployed on the MPSoC platform. These components include processors, memories, and supporting interconnects like communication buses, crossbar switches, and so on. Subsequently, the binding specifies which application task or application communication is performed by which (allocated) MPSoC component.

Figure 9.3 shows the exploration framework. The left part of the picture provides a general flow, whereas the right part illustrates the scenario-based DSE in more detail. As an input, the scenario-based DSE requires a scenario database, application models, and an MPSoC platform architecture model. Binding is performed for a multi-application workload, and the description of this workload is split into two parts: (1) the structure and (2) the behavior. The structure of applications is described using application models. For these models, the *Kahn Process Network* (KPN) model of computation is used [11], which models applications as a network of concurrent processes communicating via FIFO channels (see also refer to the section "Overview of Basic Data Flow Models" in the ▶ Chap. 3, "SysteMoC: A Data-Flow Programming Language for Codesign"). Next to the KPN application models, a scenario database [29] explicitly stores all the possible multi-application workload behaviors in terms of application scenarios (i.e., intra- and inter-application scenarios).

An important problem that needs to be solved by scenario-based DSE is the fact that the number of possible application scenarios is too large for an exhaustive evaluation of *all* – or even a restricted set of – the design points with *all* the scenarios during the MPSoC DSE. Therefore, a small but *representative subset of scenarios* must be selected for the evaluation of MPSoC design points. This representative

subset must compare mappings and should lead to the same performance ordering as would have been produced when the complete set of the application scenarios would have been used. However, the selection of such a representative subset is not trivial, as was already explained in Sect. 9.2 and studied in more detail in [27]. This is because the representative subset is dependent on the current set of mappings that are explored. Depending on the set of mappings, a different subset of application scenarios may reflect the relative mapping qualities of the majority of the application scenarios.

As a result, the representative subset cannot statically be selected. For a static selection, one would need to have a large fraction of the mappings that are going to be explored during the MPSoC DSE. However, since these mappings are only available during DSE, a dynamic selection method must be used. Thus, both the set of optimal mappings and the representative subset of scenarios need to be *coexplored simultaneously* such that the representative subset is able to adapt to the set of mappings that are currently being explored.

In the scenario-based exploration framework (see Fig. 9.3), two separate components are shown that simultaneously perform the coexploration tasks: the design explorer searches for the set of optimal mappings, while the subset selector tries to select a representative subset of scenarios. As these components are running asynchronously, a shared-memory interface is present to exchange data. For the design explorer, a sample of the current mapping population is stored in the shared memory, whereas the subset selector makes the most representative subset available for the fitness prediction in the design explorer. One of the main advantages of the strict separation of the execution of the design explorer and the subset selector is that the running time of the design explorer becomes more transparent. From a user perspective, this is the most important component, as it will identify the set of optimal mappings.

9.4 Design Explorer

In this section, the design explorer, the component that is responsible for identifying promising mappings, is described. First, our system model is described. This system model formally describes both the applications and the architecture. Next, the system model is used to describe the complete mapping procedure that will be applied during the search for good mappings, which has been implemented using a Genetic Algorithm (GA).

9.4.1 System Model

Our system model is based on the popular Y-chart design approach [13] which implies that we separate *application models* and *architecture (performance) models* while also recognizing an explicit *mapping step* (or layer) to map application tasks onto architecture resources [24]. The system model has been implemented in the Sesame system-level MPSoC simulation framework [3,18], as illustrated in Fig. 9.4.

Fig. 9.4 The different layers in the Sesame model and their connections
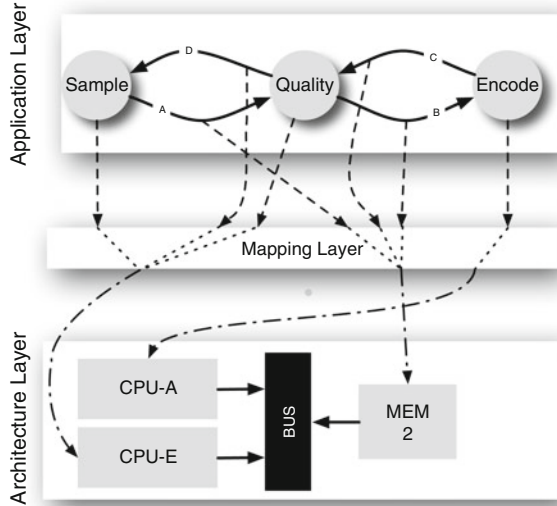
For a more detailed discussion of the Sesame framework, we also refer the interested reader to chapter ▶ Chap. 30, "DAEDALUS: System-Level Design Methodology for Streaming Multiprocessor Embedded Systems on Chips". The system model formalizes each of the application, mapping, and architecture layers:

Application (model) layer The application model describes each individual application as a Kahn Process Network (KPN) [11]. A KPN is formally defined as a directed graph $G_K(V, E_K)$. The vertexes V are the process nodes. In Fig. 9.4, an example application model is depicted in which the set V is equal to {SAMPLE, ENCODE, QUALITY}. The communication channels of the application are represented by directed edges $E_K = V \times V$. If, for example, (SAMPLE, QUALITY) is defined in E_K, it means that there is a communication channel from SAMPLE to QUALITY.

Architecture (model) layer The architecture is represented by a directed graph $G_R(R, E_R)$. In this case, the set R contains the architectural components like processors, communication buses, crossbar switches, FIFO buffers, and memories. Edges $E_R = R \times R$, on the other hand, describe the communication links in the architecture.

There are three types of architectural elements: (1) processors, (2) buffers, and (3) interconnects. The processors $R_P \subset R$ are architectural elements that are capable of executing processes. Buffers $R_B \subset R$ are the FIFO buffers/memories used for the communication between the different processors. If two communicating processes are mapped onto the same processor, the communication may also be done internally. This is only possible when a processor supports internal communication. If a processor $p \in R_P$ supports internal communication, a buffer $b \in R_B$ is added to the architecture. Additionally, the buffer is connected to enable

reading and writing of data: $(p, b) \in E_R \wedge (b, p) \in E_R$. Finally, the set of interconnects $R_I \subset R$ is purely meant to connect the various system components.

In the example of Fig. 9.4, the architectural processors R_P consist of CPU-A and CPU-E. Next, the FIFO buffers in the architecture are MEM-2 and CPU-E. This means that CPU-E supports internal communication. Finally, the BUS belongs to the set of interconnects. There are eight edges in the architecture. Six of these links are connected to the bus (for reading and writing). Additionally, the internal communication buffer of CPU-E is connected to the processor for reading and writing. In this example, all communication links are bidirectional.

Mapping Layer The mapping layer connects the application layer to the architecture layer. Hence, it contains only edges: computation edges and communication edges. Computation mapping edges E_X assign KPN processes to the architectural resources. To be precise, the edge $(v, p) \in E_X$ assigns KPN process $v \in V$ to processor $p \in R_P$. A KPN process can only be mapped on processing elements that are feasible of running that task:

$$(v, p) \in E_X \iff \text{Feasible}(p, v) \tag{9.1}$$

This allows for modeling processors that range from general-purpose processors (i.e., those processors $p \in R_P$ for which holds that $\forall v \in V : \text{Feasible}(p, v)$) to processors that are able to perform only a limited set of tasks like, e.g., ASICs. Here, we would like to note that it is also possible to map multiple KPN processes onto a single processor if the (modeled) processor type allows this (e.g., in the case of a general-purpose processor, ASIP, etc.). The communication is mapped using communication edges E_C. A communication edge (c, b) maps FIFO channel $c \in E_K$ to FIFO buffer $b \in R_B$.

9.4.2 Mapping Procedure

While the application and the architecture layers are predefined before a DSE is started, the mapping layer is the part of the MPSoC design that needs to be optimized. As discussed before, the mapping consists of two steps: allocation and binding. Allocation can reduce the resource usage of the MPSoC design, whereas the binding maps all processes and channels on the allocated resources. The procedure is as follows:

Allocation First, the architecture resources are selected to use in the allocation α. All types of architecture resources are selected at once: $\alpha_P = \alpha \cap R_P$, $\alpha_B = \alpha \cap R_B$, and $\alpha_I = \alpha \cap R_I$. More precisely, the allocation α contains a subset of resources such that $\alpha \subseteq R$:

$$\left(\sum_{r \in \alpha} \text{area}(r) \right) \leq \text{MAX\_AREA} \tag{9.2}$$

Hence, Eq. 9.2 implies that the total area of the allocated resources may not be larger than the maximal area of the chip. Part of the system model is also the feasibility of the mapping: for each of the processes, there must be at least one processor that is capable of executing the specific process. This is defined as follows:

$$\forall v \in V : |p \in \alpha_P : \text{Feasible}(p, v)| \geq 1 \tag{9.3}$$

Once the allocation α is known, a set of potential communication paths $\psi = (\alpha_P \times \alpha_B \times \alpha_P)$ can be defined:

$$\psi = \{(p_1, b, p_2) : \text{PATH}_\alpha(p_1, b) \wedge \text{PATH}_\alpha(b, p_2)\} \tag{9.4}$$

The set of communication paths ψ is the set of paths such that (:) there is a path from processor p_1 to buffer b and a path from buffer b to processor p_2 (Eq. 9.4). This path may span multiple resources as long as they are allocated:

$$\text{PATH}_\alpha(r_1, r_2) := (r_1, r_2) \in E_R \vee \tag{9.5a}$$

$$\exists r_i \in \alpha_I : (r_1, r_i) \in E_R \wedge \text{PATH}_\alpha(r_i, r_2) \tag{9.5b}$$

The PATH function is recursively defined. There is a path between resources r_1 *and* r_2 if there is a direct connection between them (Eq. 9.5a). An interconnect r_i can also be used as part of the path. In this case, there must be a direct connection between resource r_1 and interconnect r_i and a path between interconnect r_i and resource r_2 (Eq. 9.5b). An allocation is only valid if there is at least one communication path between each set of processors:

$$\forall p_1, p_2 \in \alpha_P : \exists (p_1, b, p_2) \in \psi \tag{9.6}$$

By enforcing at least a single communication path between each set of processors, the automatic exploration of mappings is guaranteed to find at least one valid mapping. As will be explained later, the procedure randomly picks the processors after which one of the communication paths is selected.

Binding   Binding maps all the KPN process nodes onto the allocated resources. There are two steps: (1) computational binding and (2) communication binding. Computational binding β_X maps the processes onto the processors such that $\beta_X \in E_X$:

$$\forall v \in V : |\{p : (v, p) \in \beta_X \wedge p \in \alpha_P\}| = 1 \tag{9.7}$$

Equation 9.7 enforces that each process v is mapped on exactly one allocated processor p. After the computational binding, the communication binding can be done. Recall from Eq. 9.6 that we have enforced that between each set of processors, at least one communication path is present in ψ. Therefore, for each

communication channel in the application, a communication path in the allocated architecture can be selected. More strictly, for each communication channel in the application, an architectural buffer is selected such that $\beta_C \in E_C$:

$$\forall (v_1, v_2), b \in \beta_C : (v_1, p_1) \in \beta_X \land \tag{9.8a}$$

$$(v_2, p_2) \in \beta_X \land \tag{9.8b}$$

$$(p_1, b, p_2) \in \psi \tag{9.8c}$$

$$\forall c \in E_K : |\{b : (c, b) \in \beta_C\}| = 1 \tag{9.9}$$

Multiple conditions must be enforced. First, the architectural buffer b on which the communication channel (v_1, v_2) of an application is mapped must be a valid communication path. This means that processes v_1 and v_2 must be mapped on processors p_1 and p_2 (Eq. 9.8a and 9.8b) and that (p_1, b, p_2) is within the set of communication paths ψ (Eq. 9.8c). Processor p_1 and processor p_2 do not necessarily need to be different as both of the processes in the communication link may be mapped onto the same processor. Next, all communication channels c (which is a tuple of the two communication processes) must be mapped on exactly one buffer (Eq. 9.9).

A mapping m is the combination of an allocation α and the bindings β_X and β_C. It is only valid if all the preceding constraints (Eqs. 9.2, 9.3, 9.6, 9.7, and 9.9) are fulfilled.

9.4.3 Exploring Mappings Using a Genetic Algorithm

Our aim is to optimize the mapping of an embedded system. Hence, the space of possible mappings must be explored as efficiently as possible. For this purpose, an NSGA-II-based multi-objective GA [4] is used (see also ▶ Chap. 6, "Optimization Strategies in Design Space Exploration"). Figure 9.5 shows the chromosome design for exploring the mapping space. The mapping chromosome consists of two parts: (1) a KPN process part and (2) a KPN communication channel part. Within these parts, all of the applications are encoded consecutively. The gene values encode the architecture components on which the elements of the applications are mapped: the KPN processes are mapped onto processors, and the KPN channels are mapped onto memories. A special memory is the internal memory, as was previously explained.

The example chromosome in Fig. 9.5 has 11 genes. Five genes are dedicated to the processes, and six genes are dedicated to the communicational channels. As there are three potential processors, the gene value for the KPN process part is between 0 and 2. For the memories, there are three possibilities: two memories and a reserved entry for the internal memory. In this way, the binding to architectural components is encoded for all of the processes and channels. The first process gene
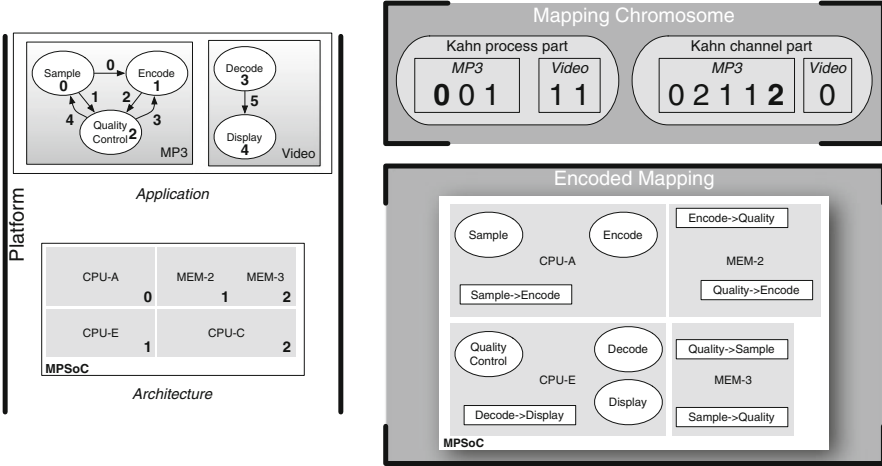
Fig. 9.5 Chromosome representation of a mapping. Both the gene sequence is shown and the mapping that is encoded by the gene sequence

of the MP3 application, for example, has gene value 0. Looking at the platform, gene 0 is the SAMPLE process. This process is mapped on the first processor: CPU-A. Similarly, the channel 4 (QUALITY $\rightarrow$ SAMPLE) is mapped on MEM-3. The complete encoded mapping is illustrated in Fig. 9.5.

The NSGA-II GA is an elitist selection algorithm that applies non-dominated sorting to select the offspring individuals. Non-dominated sorting ranks all the design points based on their dominance depth [2]. Conceptually, the dominance depth is obtained by iteratively removing the Pareto front from a set of individuals. After each iteration, the rank is incremented. An example is shown in Fig. 9.8c. The main reason for choosing an NSGA-II-based GA is because of its use of the dominance depth for optimization. As will be discussed in Sect. 9.5.2, the dominance depth can easily be used for rating the quality of the representative subset of scenarios.

The dominance of the individuals is based on their fitness. As discussed before, the predicted fitness is used instead of the real fitness. Let S be the total set of scenarios and $\tilde{S}_j$ the representative subset of scenarios at time step j. The fitness objectives of a mapping m are as follows:

$$F(m) = \frac{1}{|S|} \sum_{s \in S} (\text{time}(m, s), \text{energy}(m, s), \text{cost}(m)) \qquad (9.10)$$

$$\tilde{F}_{\tilde{S}_j}(m) = \frac{1}{|\tilde{S}_j|} \sum_{s \in \tilde{S}_j} (\text{time}(m, s), \text{energy}(m, s), \text{cost}(m)) \qquad (9.11)$$
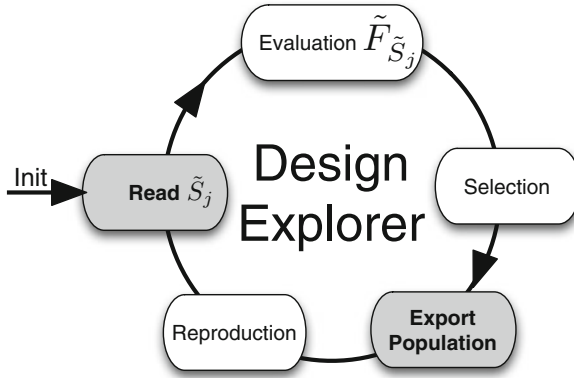
Fig. 9.6 The genetic algorithm for the design explorer extended with the required steps to communicate with the subset selector. The steps that are emphasized involve communication

Given the mapping m and the application scenario s, the functions $\text{time}(m, s)$ for execution time and $\text{energy}(m, s)$ for energy consumption are evaluated using the Sesame system-level MPSoC simulation framework. The cost of a mapping is independent of the scenario and can be determined statically by adding up the costs of the individual elements within the allocation α. There is an important difference between the *real* fitness F and the *estimated* fitness $\tilde{F}_{\tilde{S}_j}$. The real fitness uses all possible application scenarios to determine the fitness, whereas the estimated fitness only uses a (representative) subset of the scenarios ($\tilde{S}_j \subseteq S$). As a result, the real fitness is independent of the current generation. The predicted fitness, on the other hand, may vary over the different generations. The fitness $\tilde{F}_{\tilde{S}_j}$ is only valid for generation j as the representative subset $\tilde{S}_{j+1}$ may change over time.

In order to update the representative subset of scenarios between the generations, the GA of the design explorer must be extended to support the communication between the design explorer and the subset selector. This extension is shown in Fig. 9.6. Before any individual (i.e., mapping) can be evaluated, the currently most representative subset of scenarios $\tilde{S}_j$ must be acquired. Using the representative subset of scenarios, the design explorer can quickly predict the fitness of all the individuals in the population. This means that depending on the number of changed scenarios in the representative subset of scenarios since the previous generation, the parent population also must be partially reevaluated. This predicted fitness is used to select the individuals for the next generation. In case the scenario subset is representative, the decisions made by the NSGA-II selector are similar to those where the real fitness would have been used. If this is not the case, the scenario subset should be improved. For this purpose, the selected population is exported to the subset selector. Finally, reproduction is performed with the selected individuals. During reproduction, a new population of individuals is generated for usage in the next generation.

9.5   Subset Selector

To work properly, the design explorer requires a representative subset of application scenarios. The better the fitness prediction in the design explorer, the better the outcome of the scenario-based DSE is. Therefore, the subset selector is responsible for selecting a subset of scenarios. This subset selection is not trivial. First of all, there are a potentially large number of scenarios to pick from. This leads to a huge number of possible scenario subsets. On top of that, the scenario subset cannot be selected statically as the representativeness of the scenario subset is dependent of the current set of mappings. This set of mappings is only available at run time. Therefore, the scenario subset is selected dynamically.

At the end of the previous section, it was already explained that the design explorer communicates its current mapping population to the subset selector. This set of mappings can be used to *train* the scenario subset such that it is representative for the current population in the design explorer. As the population of the design explorer slowly changes over time, the subset will change accordingly. The overview of the scenario-based DSE (see Fig. 9.3) shows that the subset selector contains two threads of execution: the selector thread and the updater thread. Figure 9.7 shows these threads in more detail. The updater thread obtains the mapping individuals from the design explorer and updates the training set T_i of application mappings. This set of training mappings is used by the selector thread for selecting a representative subset of scenarios. The most representative subset of scenarios is exported to the design explorer.

In the remainder of this section, we provide a detailed overview of the subset selector. Before doing so, however, we will first describe the updater thread that is responsible for updating the trainer. Next, the metric used to judge the quality of the
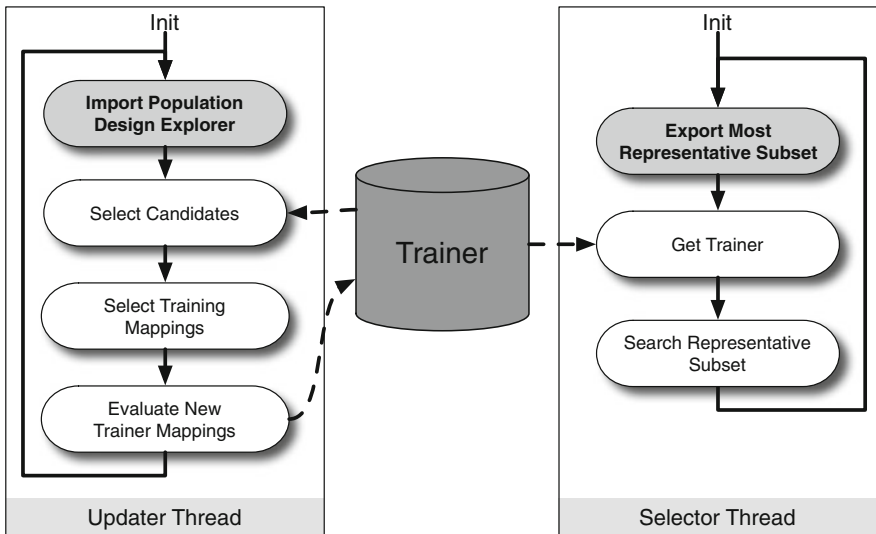

Fig. 9.7   The design of the subset selector

scenario subsets is described. The final subsection will show how the subset quality metric is used within the selector thread to select scenario subsets.

9.5.1 The Updater Thread

During the search of a representative subset of scenarios, it is crucial to have a set of training mappings T_i. Without a set of exhaustively evaluated mappings, one cannot judge if the predicted fitness of a scenario subset makes a correct prediction. As a training mapping is evaluated for all scenarios, it is relatively expensive to evaluate a mapping that needs to be added to the trainer. Therefore, it is important that the training mappings are selected carefully. Figure 9.7 illustrates the trainer update from T_i to T_{i+1}. The steps are as follows:

Import Population Design Explorer: To keep the training set T_i up to date with the mapping population in the design explorer, the current design explorer population g_j is imported.

Select Candidates: The current population is used to update the list of candidate mappings C_{i+1}:

$$\underset{C_{i+1}}{\text{maximize}} \sum_{m \in C_{i+1}} \text{last\_gen}(m)$$

$$\text{subject to (1) } C_{i+1} \subseteq C_i \cup g_j$$

$$\text{(2) } C_{i+1} \cap T_i = \emptyset$$

$$\text{(3) } |C_{i+1}| \leq \text{C\_SIZE}$$

While updating the candidate mappings, there are three conditions. First, the new set of candidate mappings is the union of the previous set of candidate mappings and the population g_j that was just received from the design explorer. Secondly, condition (2) makes sure that all the candidate mappings are not yet in the trainer. Using these two conditions, the procedure selects a set of candidate mappings that is new to the trainer. Still, the first two conditions do not provide any control on the size of the set of candidate mappings. As the selection of training mappings involves computational overhead, the size of the set of candidate mappings must be limited as well. Therefore, condition (3) makes sure that the size of the set of candidate mappings is not larger than the predefined constant C_SIZE. As the optimization goal is to maximize the sum of the last generation that each of the training mappings was used (as returned by the function last_gen), the most recently used mappings will be kept in the set of candidate mappings (these have the highest value for last used generation). Additionally, the optimization of the total sum tries to get the number of candidate mappings as large as possible: the least recently used candidate mappings will be removed until the set of candidate mappings is smaller or equal to C_SIZE. In this way, the representative subset of scenarios can be optimized to predict the fitness of the current population of the design explorer.

Select Training Mappings: For each of the candidate mappings, the predicted fitness using the currently most representative subset of scenarios is obtained. For the candidate mappings that have just been imported from the design explorer, this should not require any new Sesame simulations (the population is quite likely evaluated using the same representative subset). For older candidate mappings, some computational overhead may be required for the partial reevaluation of the mapping fitness. Together with the mappings in the current trainer T_i, an estimated Pareto front $\tilde{P}_j$ is obtained.

The main goal of the representative subset is to correctly identify good mappings. Therefore, the trainer will focus on the mappings that are the closest to the Pareto front. Any mapping may have a fitness that is hard to predict (a mapping with a high quality or a mapping with a poor quality), but the scenario-based DSE only suffers from high-quality mappings that have an incorrectly predicted fitness. As long as both the real and predicted fitness of a mapping are bad, it does not really matter how bad the predicted fitness is. However, it is still an issue if the predicted quality of a mapping is poor, whereas the real quality is good. In this case, the mapping will not be added to the trainer. Although this is undesirable, without exhaustively evaluating the candidate mappings, these kinds of incorrect predictions cannot be detected. As the exhaustive evaluation is expensive, the gain in the trainer quality does not outweigh the additional computational overhead that is required to identify the high-quality mappings where the predicted mapping quality is low. Over time the predicted ordering of the mappings near the predicted Pareto front will be improved. Likely, this will also improve the prediction of other mapping individuals.

Therefore, the k new training mappings M_c are selected from the set of candidate mappings C_{i+1} by optimizing the distance to the estimated Pareto front:

$$\underset{M_c}{\text{minimize}} \sum_{m \in M_c} \underset{m_p \in \tilde{P}_j}{\min} \left(\overline{d} \left(\tilde{F}_{\tilde{S}_j}(m_p), \tilde{F}_{\tilde{S}_j}(m) \right) \right)$$

$$\text{subject to (1) } M_c \subseteq C_{i+1}$$

$$(2) |M_c| = \min(|C_{i+1}|, k)$$

The mappings are ordered on their normalized Euclidean distance to the closest mapping in the estimated Pareto front $\tilde{P}_j$. Here, the normalized Euclidean distance $\overline{d}$ between solutions x_1 and x_2 (with f being the fitness function and n the number of optimization objectives) is defined as:

$$\overline{f_i}(x) = \frac{f_i(x) - f_i^{\min}}{f_i^{\max}(x) - f_i^{\min}}$$

$$\overline{d}(x_1, x_2) = \sqrt{\sum_{i=1}^{n} (\overline{f_i}(x_1) - \overline{f_i}(x_2))^2}$$

The normalized distance translates all the objectives to a range between 0 and 1. For this purpose, the minimal ($f_i^{\min}$) and maximal value ($f_i^{\max}$) for the objective must be known. From the candidate mappings (condition 1), the k mappings are selected (condition 2) that are the closest to the estimated Pareto front.

Evaluate New Training Mappings: The mappings that are selected are exhaustively evaluated using Sesame. For this purpose, a separate pool of Sesame workers is used (just as the design explorer has a pool of Sesame workers). Once the real fitness is known, the mappings can be used to generate trainer T_{i+1} out of trainer T_i. Before the new training mappings are added, the trainer is truncated to fit the new training mappings. This is done in such a way that the trainer always contains real Pareto front P:

$$\underset{T_{i+1}}{\text{minimize}} \sum_{m \in T_{i+1}} \min_{m_p \in P} \left(\overline{d} \left(F(m_p), F(m) \right) \right)$$

subject to (1) $T_{i+1} \subseteq T_i$

(2) $|T_{i+1}| = \min(|T_i|, \text{T\_SIZE} - |M_c|)$

The truncated new trainer is a subset of the old trainer (condition 1), and it does not exceed the predefined trainer size. If trainer mappings must be discarded, the mappings that are the furthest from the real Pareto front are removed. This is done because of the second purpose of the trainer: at the end of the scenario-based DSE, it contains the best mappings that are found over time with their real fitness. Hence, we assume that the maximal trainer size is picked in such a way that it is significantly larger than the size of the Pareto front P. After truncation, the next trainer can be finalized: $T_{i+1} = T_{i+1} \cup M_c$.

9.5.2 Subset Quality Metric

Having a set of training mappings is not sufficient for judging the quality of the scenario subsets. To determine the actual quality of a subset of representative scenarios, we use the *misclassification rate* metric. The misclassification rate counts the number of ranks that is predicted incorrectly. Before we go into the definition of the misclassification rate, we first take a look into the Pareto ranking [31]. There are several approaches to rank individuals using the Pareto dominance relations, but in this chapter, we only focus on two of those: Boolean ranking and Goldberg's ranking (also called non-dominated sorting).

The ranking schemes are visualized in Fig. 9.8. Goldberg's ranking approach uses the dominance depth of the individuals. This is the same approach as the NSGA-II selector. Boolean ranking, on the other hand, follows a more simple approach: if the solution is non-dominated, the rank is one; otherwise the rank is two. As the design explorer uses an NSGA-II-based GA, it may be straightforward to use Goldberg's ranking scheme for the misclassification rate. The Boolean ranking,
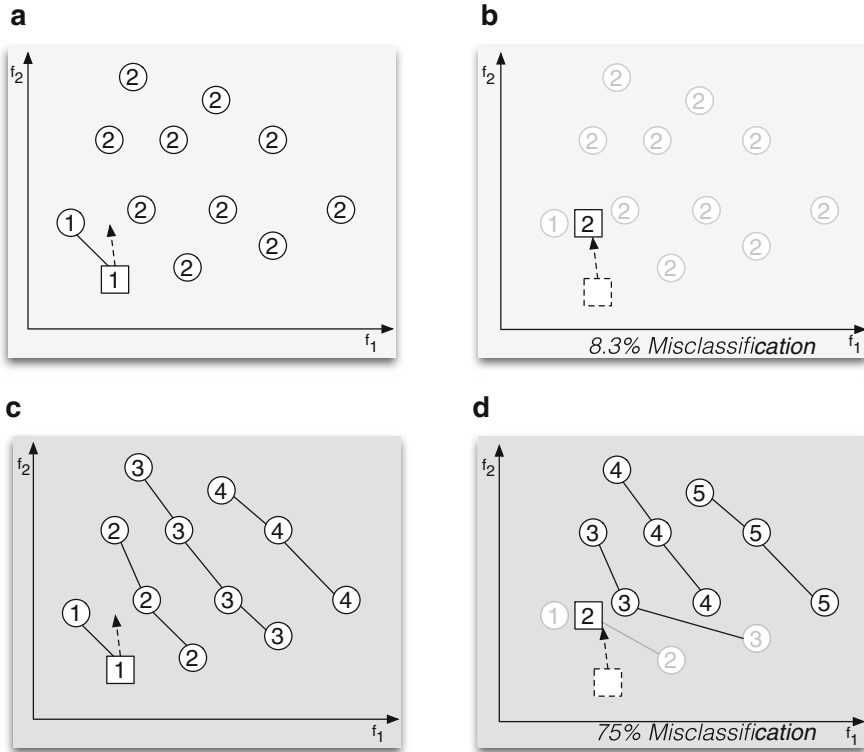
Fig. 9.8 A set of Pareto fronts showing the effect of a small error in the prediction (as shown with the dashed arrow) on the misclassification rate using the Boolean and Goldberg's ranking scheme. (**a**) Boolean ranking, real fitness. (**b**) Boolean ranking, predicted fitness. (**c**) Goldberg's ranking, real fitness. (**d**) Goldberg's ranking, predicted fitness

however, can be obtained more efficiently than the Goldberg's ranking. On top of that, the misclassification rate may be deceiving when Goldberg's ranking is used.

In Fig. 9.8, an example of such a deceiving case is given. The used trainer consists of 12 training mappings. Figure 9.8a, c show the exact fitness of the training mappings, whereas Fig. 9.8b, d show the predicted fitness of a specific scenario subset. This scenario subset provides a relatively good prediction: 11 out of the 12 training mappings are predicted correctly (the circular mappings). The incorrectly predicted training mapping (the square mapping) is slightly off as shown by a dashed arrow. Due to the incorrect prediction, the square mapping seems to be dominated by the leftmost training mapping. For both ranking schemes, the rank of the square mapping becomes ranked second instead of first. In case of the Boolean ranking, this is the only rank that is incorrect. For Goldberg's ranking, however, all the training mappings that are dominated by the square mapping are also incremented by one. As a result, the Goldberg's ranking has a misclassification rate of $\frac{3}{4}$, whereas the

Boolean ranking has a misclassification rate of $\frac{1}{12}$. Our example clearly shows that a high-quality mapping that is incorrectly ranked can affect all of its dominated solutions. However, as the main purpose of scenario-based DSE is that the Pareto front is predicted correctly, it is not a problem that the poor mappings are incorrectly ranked.

This is exactly what is determined with Boolean ranking. Each rank is based on the correct prediction of a non-dominated individual. A formal definition of the Boolean ranking is given in the following equation:

$$\text{rel}_{F'}(m_1, m_2) := \begin{cases} 1 & F'(m_1) \text{ dominates } F'(m_2) \\ -1 & F'(m_2) \text{ dominates } F'(m_1) \\ 0 & else \end{cases} \quad (9.12)$$

$$\text{rank}_{F'}(m, T) := \begin{cases} 2 & \exists m' \in T \ (\text{rel}_{F'}(m', m) = 1) \\ 1 & else \end{cases} \quad (9.13)$$

Equation 9.12 formally defines the Pareto dominance between two mappings m_1 and m_2. The mappings are evaluated using fitness function F'. This can be the real fitness F but also the predicted fitness $\tilde{F}_{\tilde{S}}$. In this case, the scenario subset $\tilde{S}$ is used to predict the fitness of the mappings.

Based on the $\text{rel}_{F'}$ function, the $\text{rank}_{F'}$ is defined. This function ranks mapping m given trainer T (see Eq. 9.13). In case any of the mappings in the trainer dominates the mapping, the rank is equal to two. Otherwise, the mapping is Pareto optimal, and the rank is equal to one. Given the ranking function, the misclassification rate can be defined:

$$\text{r}_{\text{rank}}(\tilde{S}, T) := \frac{|\{m \in T : \text{rank}_F(m, T) \neq \text{rank}_{\tilde{F}_{\tilde{S}}}(m, T)\}|}{|T|} \quad (9.14)$$

The rate of misclassified Boolean ranks is too coarse grained to be used in isolation. In contrast to, e.g., Spearman's rank correlation [26], a lower misclassification rate is always better (the more non-dominated individuals that are correctly identified, the better). However, the probability of an equal misclassification rate is quite likely.

In this case, the *number of misclassified relations* is used as a tiebreaker. The number of misclassified relations can be defined quite straightforwardly:

$$\text{r}_{\text{rel}}(\tilde{S}, T) := \frac{|\{m_1, m_2 \in T : \text{rel}_F(m_1, m_2) \neq \text{rel}_{\tilde{F}_{\tilde{S}}}(m_1, m_2)\}|}{|T|^2} \quad (9.15)$$

By definition, when the number of misclassified relations is zero, the number of misclassified ranks is also zero. For the other cases, the number of misclassified
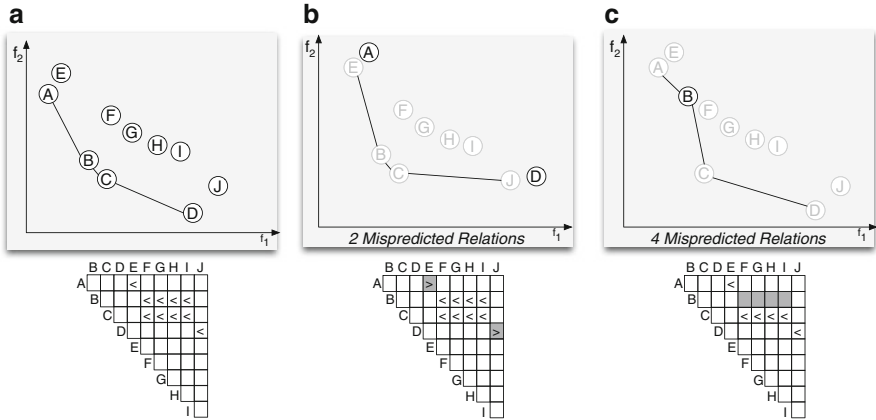
Fig. 9.9 A larger number of misclassified relations do not strictly correlate with the Pareto front quality. The Pareto front in (**c**) has more mispredicted relations than the front in (**b**), but the error ratio with respect to the real front is better. (**a**) Real. (**b**) 50% error ratio. (**c**) 0% error ratio

relations can suffer from the same problem as we showed with Goldberg's ranking. An example is shown in Fig. 9.9. Figure 9.9a shows the real Pareto front, where the fronts of Fig. 9.9b, c are obtained using a predicted fitness. The first prediction (Fig. 9.9b) only has two mispredicted relations (A ⟷ E and D ⟷ J), whereas the second prediction (Fig. 9.9c) has four mispredicted relations. Still, the Pareto front of the first prediction is only correct for 50% (E and J are not Pareto optimal). The second prediction, which is worse according to the number of misclassified relations, correctly identifies the Pareto front. As we are using the number misclassified relations as a subordinate metric, and not as a main metric, this is no problem in our case. Figure 9.9b has a misclassification rate of 20% that is worse than the misclassification rate of 0% in Fig. 9.9c.

9.5.3 The Selector Thread

The selector thread uses the subset quality metrics to select the representative subset of scenarios. More specifically, the goal of the selector thread is as follows:

$$\underset{\tilde{S}}{\text{minimize}}\ r_{\text{rank}}\left(\tilde{S}, T\right) :\ \underset{\tilde{S}}{\text{minimize}}\ r_{\text{rel}}\left(\tilde{S}, T\right) \tag{9.16}$$

As discussed in the previous subsection, the main goal is to optimize the quality of the predicted ranking. In the case of ties, the number of mispredicted relations will determine which of the scenario subsets is the best. Whenever a better representative subset is found, the subset is shared with the design explorer in order to improve its fitness prediction. The subset may be of any size, as long as it does not exceed a user-defined limit. This means that a smaller subset that has a better or equal

representativeness is preferable to a larger counterpart (the smaller the subset is, the
faster the fitness prediction is).

This leaves us with the question of how to dynamically search for the representa-
tive subset of scenarios. In [28], it was shown that a random pick of scenarios does
not result in a representative subset of scenarios. In the following subsections, we
describe three different techniques for this searching process, using (1) a genetic
algorithm, (2) a feature selection algorithm, and (3) a hybrid combination of both a
genetic algorithm and feature selection.

9.5.3.1 GA-Based Search for Scenario Subset

Our first subset selection technique uses a genetic algorithm (GA) to select the
representative scenario subsets. The GA of the subset selector is somewhat similar
to the GA in the design explorer: a population of individuals is evolved over time
in order to find the individual with the highest fitness. In order to describe the
individual, a chromosome is used that enumerates the scenarios that are contained
in the scenario subset. This chromosome, as illustrated in Fig. 9.10, is simply a
sequence of integers that refer to scenarios from the scenario database. As the length
of the chromosome is equal to the limit of the scenario subset size, the scenario
subset can never become too large. Smaller scenario subset sizes are achieved in
two ways: (1) scenarios may be used more than once within the same chromosome
and (2) there is a special value for an unused slot in the scenario subset.

Scenario subsets can change size as an effect of the mutation or crossover that
is applied during the search of a GA. The evolutionary process uses mutation and
crossover to prepare individuals for the next generation of scenario subsets. Where
the mutation replaces the scenarios in the subset one by one with other scenarios,
the crossover partly exchanges the scenarios of two subsets. Only the successful
modifications will make it to the next generation, leading to a gradual improvement
of the representative subset of scenarios.

This approach has several benefits. First, the computational overhead is relatively
small. Most time is spent in judging the quality of the scenario subsets and
modifying the population of scenario subsets. Additionally, selecting scenario
subsets for the next generation is relatively cheap. Apart from the low computational
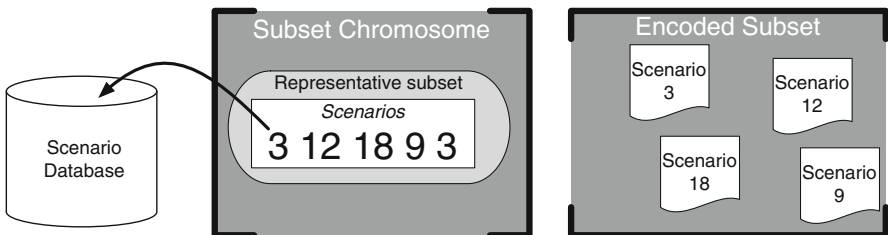overhead, the search can also quickly move through the total space of scenario

Fig. 9.10 Chromosome representation of a subset. Both the gene sequence is shown and the
representative subset of scenarios that is encoded by the gene sequence

subsets. Due to crossover and mutation, the scenario subset can quickly change, and the alternatives can be evaluated and explored. This also means that local optima can easily be avoided. A local optimum is a solution that is clearly better than all closely related solutions. A small change of a local optimum does typically not result in a better subset, but the local optimum may still be much worse than the global optimal solution. As a genetic algorithm always has a small probability that the scenario subsets are changed significantly in the next generation, there is always a probability that the search will escape from the local optimum.

Unfortunately, this is also the downside of the approach. Just when the search comes close to the optimal solution, the mutation can quickly move the search into a completely different direction. Although the likeliness of this all depends on the choice of parameters such as mutation and crossover probability, it may be quite hard to pick the parameters in such a way that the search space is completely explored in the promising regions. Elitism in GAs assures that the points close to the local optimum will be retained as part of the population, but not that the neighborhood of each solution is carefully explored.

9.5.3.2 FS-Based Search for Scenario Subset

It would be better if the approach was less dependent on the choice of the parameters of the search algorithm. The Feature Selection (FS) technique has less parameters, and it basically performs a guided search that tries to improve a single scenario subset step by step. There are many different feature selection techniques, each giving a different trade-off between computational overhead and its quality. In fact, the feature selection techniques with the lowest computational overhead actually use a GA. In our case, we have chosen to use the dynamic sequential oscillating search [25] as, in general, it provides better classifiers (i.e., the scenario subset that classifies the non-dominated mapping individuals), with a moderate computational overhead.

Figure 9.11 illustrates the dynamic oscillating search. The most fundamental part of the algorithm is the up- and downswing. These swings are named according to their effect on the size of the scenario subset. Where the upswing will modify the subset by first adding a number of scenarios to the subset and then removing the same number of scenarios, the downswing will first remove scenarios before new scenarios are added again. This explains the name of the upswing and downswing:
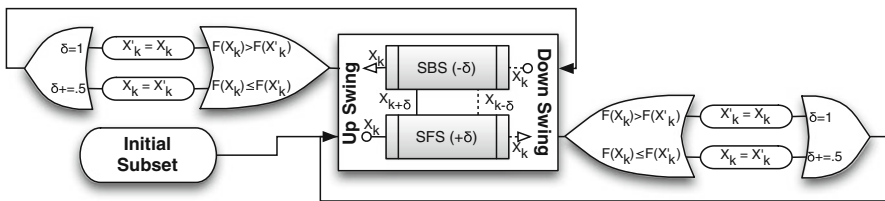

Fig. 9.11 An illustration of feature selection by means of the dynamic oscillation search procedure

in case of the upswing, the size of the subset swings upward, and for a downswing, it swings downward. For adding and removing scenarios, the *Sequential Forward Selection* (SFS) and *Sequential Backward Selection* (SBS) are used. These techniques will iteratively add or remove a scenario in the most optimal way. This means that SFS will increase the scenario subset size by iteratively adding the most optimal scenario. This most optimal scenario is determined by trying out all possible scenarios from the scenario database, and the scenario that results in the best scenario subset will be added to the larger subset. Similarly, the SBS will iteratively choose the optimal scenario to remove from the scenario subset. This means that all the scenarios that can be removed from the scenario subset are tried and the scenario removal that results in the best scenario subset will be applied.

As simple as it sounds, it makes the computational overhead of the swings largely dependent on the number of scenarios that are added or removed. The number of possibilities will grow linearly with respect with the number of scenarios that are added and removed during the swing. For each scenario that is added, all scenarios in the scenario database must be analyzed. Since this leads to a quick increase of computational overhead once the swings become larger, the size of the swing (or δ as used in Fig. 9.11) is initialized to one and slowly increased. During the search, the up- and downswing are alternated, and whenever both the up- and downswing do not result in a better scenario subset, the size of the swing is incremented by one. This can be seen in Fig. 9.11, at the cases where $F(X_k) \leq F(X'_k)$. The subset X'_k is the current best subset, and the subset X_k is the subset that is obtained after the up- or downswing. As a higher value for the function F means a better scenario subset, the case where $F(X_k) \leq F(X'_k)$ is an unsuccessful attempt to improve the scenario subset by a swing. Therefore, the currently best subset is restored, and the size of the swing is increased by 0.5. The value 0.5 is used to increment the swing by one after two unsuccessful swings: the number of scenarios that are added is the truncated integer value of δ. Of course, the swing can also be successful: in that case, the current best subset X'_k is updated, and the swing size is reset to one.

In a sense, the dynamic oscillating search is a kind of hill climbing technique. It oscillates the size of the scenario subset by exhaustively exploring all possibilities to change the scenario subset. Whenever a better subset is found, the current best representative subset is updated. Important to realize is that the current best subset can also be updated during a swing. As SFS and SBS analyze the quality of the scenario subset for each scenario that is added, it can be the case that a better representative subset is found during the swing. If the size of this subset is smaller than the maximal size, the currently most representative subset is updated and sent to the design explorer.

The FS method is more directed than the GA, and, therefore, it will only move closer to the optimal scenario subset. Unfortunately, this comes at a price: the FS is much more sensitive to local optima than the GA approach.

9.5.3.3 A Hybrid Approach for Searching Scenario Subsets

Ideally, we want to combine the strengths of the GA and the FS approaches. The hybrid approach (as shown in Fig. 9.12) tries to achieve this by alternating the GA

Fig. 9.12 The hybrid subset selection approach that alternates between a GA and a FS algorithm
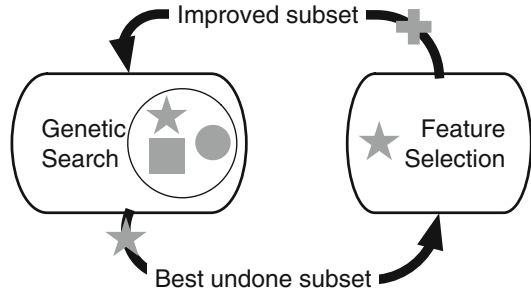
and the FS methods. During the search for the representative subset, a GA will quickly prune the design space of potential scenario subsets, whereas the FS will thoroughly search the small neighborhood around the high-quality scenario subsets that are found by the GA. The tricky point is the moment of alternation. When one of the methods starts to converge, the other method should be activated.

At first sight, the feature selection may be interpreted as a custom variation operator for the GA, but this is absolutely not the case. Both the GA and the FS will keep state over time, and, thus, if the same subset is sent to the FS more than once, the oscillating search will be continued where it stopped in the previous invocation.

As the GA keeps a population of scenario subsets and the FS only works on a single scenario subset, it must be determined which scenario subset from the GA population is sent to the FS selection method. The most obvious method is to send the most representative subset from the GA to the FS. This can, however, not be done indefinitely. If the same subset is sent to the FS too often, the hybrid approach will again be susceptible for getting stuck in local optima as all the effort of the FS will be spent on the same subset. Therefore, the amount of effort spent by the FS to improve a single scenario subset is limited. If the FS has spent sufficient time on the same scenario subset (this time can be spread over multiple invocations of the FS), the subset is done, and it will not be sent to the FS anymore. So, the subset is only sent if it is "unfinished": the size of the swing in the oscillating search does not exceed a predefined maximal margin. This margin is chosen in such a way that the computational overhead of a single swing is still acceptable.

9.6   Related Work

In recent years, much research has been performed on high-level modeling and simulation for MPSoC performance evaluation as well as on GA-based DSE [5, 8]. However, the majority of the work in this area still evaluates and explores systems under a single, fixed application workload. Some research has been initiated on recognizing workload scenarios [7, 17] and making DSE scenario aware [16, 32]. In [7], for example, different single-application scenarios are used for DSE. Another type of scenario is the use-case. A use-case can be compared with what we call

inter-application scenarios, and consequently, a use-case describes which applications are able to run concurrently. Examples of frameworks utilizing use-cases for mapping multiple applications are MAMPS [14] and the work of Benini et al. [1]. MAMPS is a system-level synthesis tool for mapping multiple applications on a FPGA, whereas Benini et al. use logic programming to reconfigure an embedded system when a new use-case is detected. Another way of describing the use of multiple applications is a multimode multimedia terminal [9], in which the inter-application behavior is captured in a single, heterogeneous Model of Computation (MoC) combining dataflow MoCs and state machine MoCs.

9.7   Discussion

Scenario-based DSE efficiently explores the mapping of dynamic multi-application workloads on an MPSoC platform. Crucial for the efficiency of such mapping exploration is the subset selector that dynamically selects the fitness predictor for the design explorer. This fitness predictor is a subset of application scenarios that is used by the design explorer to quickly identify the non-dominated set of MPSoC mappings. In this chapter, we have given a detailed description of how the representativeness of a scenario subset can be calculated and which techniques can be used to select the fitness predictor (i.e., the subset of scenarios).

The three different fitness prediction techniques that were presented are (1) a genetic algorithm (GA), (2) a feature selection (FS) algorithm, and (3) a hybrid method (HYB) combining the two aforementioned approaches. A genetic algorithm is capable of quickly exploring the space of potential scenario subsets, but due to its stochastic nature, it is susceptible to missing the optimal scenario subsets. This is not the case with the feature selection algorithm as it more systematically explores the local neighborhood of a scenario subset. Unfortunately, this approach is relatively slow and can suffer from local optima. The solution is to combine these approaches in the hybrid approach, leading to a fitness prediction technique that can quickly prune the design space, can thoroughly search the local neighborhood of scenario subsets, and is less susceptible to local optima.

To give a feeling of the performance of the three different fitness prediction techniques, Fig. 9.13 shows the results of a scenario-based DSE experiment in which the three techniques are compared for three different subset sizes (1, 4, and 8% of the total number of application scenarios). In this experiment, the mapping of ten applications with a total of 58 processes and 75 communication channels is explored. The multi-application workload consists of 4607 different application scenarios in total. The target platform is a heterogeneous MPSoC with four general-purpose processors, two ASIPs and two ASICs, all connected using a crossbar network. In this experiment, we have measured the required exploration time for the scenario-based DSE to identify a satisfying mapping. After all, the faster the DSE can provide results that match the requirement of the user, the better it is. For this purpose, a DSE of 100 min is performed for all three subset selector approaches. The results have been averaged over nine runs. To determine the efficiency of the
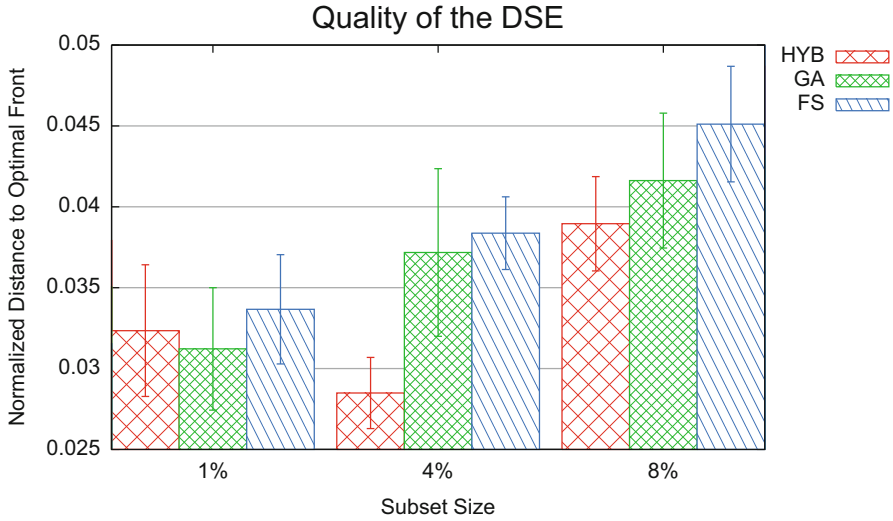
Fig. 9.13 Quality of the DSE for the different subset selection approaches. The quality is determined based on the distance between the estimated Pareto front and the optimal front

multi-objective DSE, we obtain the distance of the estimated Pareto front (execution time versus energy consumption of mapping solutions) to the optimal Pareto front. For this purpose, we normalized execution time and energy consumption to a range from 0 to 1. As the optimal Pareto front is not exactly known since the design space is too large to exhaustively search it, we have used the combined Pareto front of all our experiments for this.

When increasing the subset size, two effects will occur: (1) the larger the subset, the more accurate the fitness prediction in the design explorer is and (2) the larger the subset, the longer it takes to obtain the fitness of a single mapping causing a slower convergence of the search. This can be seen in Fig. 9.13. The GA and the FS subset selection have worse results when the subset becomes larger (the smaller the distance, the better). The hybrid selector, however, shows a somewhat different effect. With a subset size of 4%, it is able to benefit from a subset with a higher accuracy. The slower convergence only starts to effect the efficiency for the 8% subset. Comparing the different methods, the hybrid method shows the best results. The only exception is for the 1% subset. In this case, the GA is still able to search the smaller design space of possible subsets. Still, the result of the hybrid method at 4% is better than the result of the GA at 1%. With the larger subset sizes, the hybrid method can exploit both the benefits of the feature selection and the genetic algorithm.

For more extensive experimental evaluations of scenario-based DSE, and the different fitness prediction techniques in scenario-based DSE in particular, we refer the interested reader to [28, 30]. These studies compare scenario-based DSE with regular DSE in the context of multi-application workloads. Moreover, they

scrutinize the quality of the mapping solutions obtained by the different variants of scenario-based DSE as well as the efficiency with which these solutions are obtained.

The scenario-based DSE presented in this chapter aims at providing a *static mapping* of a multi-application workload onto an MPSoC. Evidently, the application dynamism as captured by application scenarios can of course also be exploited at run time to *dynamically optimize* the embedded system according to the application workload at hand. For example, in [20–23] as well as in ▶ Chap. 10, "Design Space Exploration and Run-Time Adaptation for Multicore Resource Management Under Performance and Power Constraints", various approaches are proposed for adaptive MPSoC systems that allow for such dynamic system optimization. These methods typically consist of two phases: A design-time stage performs DSE to find an optimal mapping for each application. At run time, the occurrence of different application scenarios is detected, after which the system can be reconfigured by dynamically adapting the application mapping. This could, e.g., be done by merging the pre-optimized mappings of each separate, active application in the detected application scenario to form a first-order mapping for the entire scenario. Subsequently, this first-order mapping can then be further optimized by using run-time mapping optimization heuristics [22].

References

1. Benini L, Bertozzi D, Milano M (2008) Resource management policy handling multiple use-cases in MPSoC platforms using constraint programming. In: Logic programming, Udine. Lecture notes in computer science, vol 5366, pp 470–484
2. Coello CAC, Lamont GB, Veldhuizen DA (2007) Alternative metaheuristics. In: Coello Coello CA, Lamont GB, Van Veldhuizen DA (eds) Evolutionary algorithms for solving multi-objective problems. Genetic and evolutionary computation, 2nd edn. Springer, New York
3. Coffland JE, Pimentel AD (2003) A software framework for efficient system-level performance evaluation of embedded systems. In: Proceedings of the SAC 2003, pp 666–671
4. Deb K, Pratap A, Agarwal S, Meyarivan T (2002) A fast and elitist multiobjective genetic algorithm: NSGA-II. IEEE Trans Evol Comput 6(2):182–197
5. Erbas C, Cerav-Erbas S, Pimentel AD (2006) Multiobjective optimization and evolutionary algorithms for the application mapping problem in multiprocessor system-on-chip design. IEEE Trans Evol Comput 10(3):358–374
6. Gerstlauer A, Haubelt C, Pimentel A, Stefanov T, Gajski D, Teich J (2009) Electronic system-level synthesis methodologies. IEEE Trans Comput-Aided Des Integr Circuits Syst 28(10):1517–1530
7. Gheorghita SV et al (2009) System-scenario-based design of dynamic embedded systems. ACM Trans Des Autom Electron Syst 14(1):1–45
8. Gries M (2004) Methods for evaluating and covering the design space during early design development. Integr VSLI J 38(2):131–183
9. Ha S, Lee C, Yi Y, Kwon S, Joo YP (2006) Hardware-software codesign of multimedia embedded systems: the peace approach. In: Proceedings of the IEEE international conference on embedded and real-time computing systems and applications, pp 207–214
10. Jia Z, Bautista T, Nunez A, Pimentel A, Thompson M (2013) A system-level infrastructure for multidimensional MP-SoC design space co-exploration. ACM Trans Embed Comput Syst 13(1s):27:1–27:26

11. Kahn G (1974) The semantics of a simple language for parallel programming. In: Proceedings of the IFIP congress 74. North-Holland Publishing Co.

12. Keutzer K, Newton A, Rabaey J, Sangiovanni-Vincentelli A (2000) System-level design: orthogonalization of concerns and platform-based design. IEEE Trans Comput-Aided Des Integr Circuits Syst 19(12):1523–1543

13. Kienhuis B, Deprettere EF, van der Wolf P, Vissers KA (2002) A methodology to design programmable embedded systems: the Y-chart approach. In: Embedded processor design challenges. LNCS, vol 2268. Springer, Berlin/New York, pp 18–37

14. Kumar A, Fernando S, Ha Y, Mesman B, Corporaal H: Multiprocessor systems synthesis for multiple use-cases of multiple applications on FPGA. ACM Trans Des Autom Electron Syst 13(3):1–27 (2008)

15. Martin G (2006) Overview of the MPSoC design challenge. In: Proceedings of the design automation conference (DAC'06), pp 274–279

16. Palermo G, Silvano C, Zaccaria V: Robust optimization of SoC architectures: a multi-scenario approach. In: Proceedings of the IEEE workshop on embedded systems for real-time multimedia (2008)

17. Paul JM, Thomas DE, Bobrek A (2006) Scenario-oriented design for single-chip heterogeneous multiprocessors. IEEE Trans Very Large Scale Integr Syst 14(8):868–880

18. Pimentel A, Erbas C, Polstra S (2006) A systematic approach to exploring embedded system architectures at multiple abstraction levels. IEEE Trans Comput 55(2):99–112

19. Piscitelli R, Pimentel A (2012) Design space pruning through hybrid analysis in system-level design space exploration. In: Proceedings of DATE'12, pp 781–786

20. Quan W, Pimentel AD (2013) An iterative multi-application mapping algorithm for heterogeneous MPSoCs. In: Proceedings of ESTIMedia'13, pp 115–124

21. Quan W, Pimentel AD (2013) A scenario-based run-time task mapping algorithm for MPSoCs. In: Proceedings of DAC'13, pp 131:1–131:6

22. Quan W, Pimentel AD (2015) A hybrid task mapping algorithm for heterogeneous MPSoCs. ACM Trans Embed Comput Syst 14(1):14:1–14:25

23. Schor L, Bacivarov I, Rai D, Yang H, Kang SH, Thiele L (2012) Scenario-based design flow for mapping streaming applications onto on-chip many-core systems. In: Proceedings of CASES'12, pp 71–80

24. Singh AK, Shafique M, Kumar A, Henkel J (2013) Mapping on multi/many-core systems: survey of current and emerging trends. In: Proceedings of DAC'13, pp 1:1–1:10

25. Somol P, Novovicova J, Grim J, Pudil P (2008) Dynamic oscillating search algorithm for feature selection. In: Proceedings of the international conference on pattern recognition (ICPR 2008), pp 1–4

26. Spearman C (1904) The proof and measurement of association between two things. Am J Psychol 15(1):72–101

27. van Stralen P (2014) Applications of scenarios in early embedded system design space exploration. PhD thesis, Informatics Institute, University of Amsterdam

28. van Stralen P, Pimentel AD (2010) Scenario-based design space exploration of MPSoCs. In: Proceedings of IEEE international conference on computer design (ICCD'10)

29. van Stralen P, Pimentel AD (2010) A trace-based scenario database for high-level simulation of multimedia mp-socs. In: Proceedings of SAMOS'10

30. van Stralen P, Pimentel AD (2013) Fitness prediction techniques for scenario-based design space exploration. IEEE Trans Comput-Aided Des Integr Circuits Syst 32(8):1240–1253

31. van Veldhuizen DA, Lamont GB (2000) Multiobjective evolutionary algorithms: Analyzing the state-of-the-art. Evol Comput 8(2):125–147

32. Zaccaria V, Palermo G, Castro F, Silvano C, Mariani G (2010) Multicube explorer: an open source framework for design space exploration of chip multi-processors. In: Proceedings of the international conference on architecture of computing systems (ARCS), pp 1–7