Methodologies for Design Space Exploration

Pimentel, A.D.

DOI
10.1007/978-981-15-6401-7_23-1

Publication date
2022

Document Version
Final published version

Published in
Handbook of Computer Architecture

License
Article 25fa Dutch Copyright Act (https://www.openaccess.nl/en/in-the-netherlands/you-share-we-take-care)

Citation for published version (APA):
Methodologies for Design Space Exploration

Andy D. Pimentel

Contents

Introduction ................................................................. 2
DSE: The Basic Concepts ................................................... 3
   Two Basic Ingredients of DSE ......................................... 5
Y-Chart-Based DSE ....................................................... 6
Evaluation of a Single Design Point ........................................ 8
   Simulative Fitness Evaluation ....................................... 8
   Analytical Fitness Evaluation ...................................... 12
Searching the Design Space ............................................... 13
   GA-Based DSE .......................................................... 14
   Optimizing GA-Based DSE .......................................... 17
Multi-application Workload Models ......................................... 18
   Scenario-Based DSE ..................................................... 19
Application Exploration .................................................... 23
   NAS by Means of Evolutionary Piecemeal Training (EPT) .................. 23
Conclusion and Outlook .................................................... 26
References ................................................................... 28

Abstract

In this chapter, an overview of techniques and methods for the design space exploration (DSE) of embedded systems is presented. DSE is the critical design process in which system designs are modeled, evaluated, and, eventually, optimized for the various extra-functional system behaviors, such as performance, power or energy consumption, and cost. The discussion is organized along the lines of the two primary elements of DSE, namely, the evaluation of single design points and the search strategy for covering the design space.

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

© Springer Nature Singapore Pte Ltd. 2022
A. Chattopadhyay (ed.), Handbook of Computer Architecture,
https://doi.org/10.1007/978-981-15-6401-7_23-1
Keywords

Design space exploration · Multi-objective optimization · Performance modeling · Genetic algorithms

Introduction

Designers of modern embedded computer systems face several daunting challenges since these systems typically have to meet a range of stringent and often conflicting, design requirements. As many embedded systems target mass production and battery-based devices or devices that cannot use active cooling, they should be cheap and power efficient. Mission- and safety-critical embedded computer systems, like those in the avionics and space domains, usually also demand high levels of dependability, which is becoming even more important as the levels of system autonomy rise. Moreover, a great deal of these systems must, increasingly, support multiple applications and standards for which they often need to provide real-time performance. For example, mobile devices must support a variety of different standards for communication and coding of digital contents. In addition, many of these systems also need to provide a high degree of flexibility, allowing them to be easily updated and extended with future applications and standards. This calls for a high degree of programmability of these systems, whereas performance, power consumption, and cost constraints require implementing substantial parts of these systems in dedicated hardware blocks. As a result, modern embedded systems often have a heterogeneous multiprocessor system architecture. They consist of processors that range from fully programmable cores to fully dedicated hardware blocks for time-critical application tasks. Increasingly, the components in such systems are integrated onto a single chip, yielding heterogeneous multiprocessor system-on-chip (MPSoC) architectures (Wolf et al. 2008).

To cope with the design complexity of such systems, a new design methodology has emerged in the past 15 to 20 years, called system-level design (see chapter “Electronic System-Level Design”). It aims at raising the level of abstraction of the design process to improve the design productivity. Key enablers to this end are the use of MPSoC platform architectures to facilitate reuse of IP components and the concept of high-level system modeling and simulation (Keutzer et al. 2000; Sangiovanni-Vincentelli and Martin 2001). The latter allows for capturing the behavior of platform components and their interactions at a high level of abstraction. As such, these high-level models minimize the modeling effort and are optimized for execution speed and can therefore be applied during the very early design stages to perform design space exploration (DSE) (Gries 2004; Pimentel 2017). During such DSE, a large variety of different design alternatives can be explored, such as the number and type of processors deployed in the platform architecture, the type of interconnection network used to connect system components, or the spatial binding and temporal binding (i.e., scheduling) of application tasks to processor cores. It is of paramount importance to start performing such DSE as early as possible in
the design process because the considered design choices may heavily influence
the success or failure of the final product. However, the process of DSE also is
highly challenging since the design space that needs to be explored typically is
vast, especially during the early stages of design. For instance, the design space
for exploring different mappings of application tasks to processing resources – and
trying to optimize the mapping for, e.g., system performance or power consumption –
exponentially grows with the number of application tasks and processors in the
system and is known to be an NP-hard problem (Singh et al. 2013). Therefore,
the development of efficient and effective DSE methods has received significant
research attention in recent years. In this chapter, an overview will be provided of
the various aspects involved in DSE of embedded systems.

DSE: The Basic Concepts

During the DSE of embedded systems, multiple optimization objectives – such
as performance, power/energy consumption, and cost – should be considered
simultaneously. This is called multi-objective DSE (Pimentel 2017). Since the
objectives are often in conflict, there cannot be a single optimal solution that
simultaneously optimizes all objectives. Therefore, optimal decisions need to be
taken in the presence of trade-offs between design criteria.

Given a set of \( m \) decision variables, which are the degrees of freedom (e.g.,
MPSoC system parameters like the number and type of processors, application
mapping, etc.) that are explored during DSE, a so-called fitness function must
optimize the \( n \) objective values. The fitness function is defined as:

\[
f_i : \mathbb{R}^m \rightarrow \mathbb{R}^1 \quad (1)
\]

A potential solution \( x \in \mathbb{R}^m \) is an assignment of the \( m \) decision variables. The
fitness function \( f_i \) translates a point in the solution space \( X \) into the \( i \)th objective
value (where \( 1 \leq i \leq n \)). For example, a particular fitness function \( f_i \) could
assess the performance or energy efficiency of a certain solution \( x \) (representing
a specific design instance). As illustrated in Fig. 1, the combined fitness function
\( f(x) \) subsequently translates a point in the solution space into the objective space
\( Y \). Formally, a multi-objective optimization problem (MOP) that tries to identify a
solution \( x \) for the \( m \) decision variables that minimizes the \( n \) objective values using
objective functions \( f_i \) with \( 1 \leq i \leq n \):

\[
\text{minimizes } y = f(x) = (f_1(x), f_2(x), \ldots, f_n(x))
\]

\[
\text{where } x = (x_1, x_2, \ldots, x_m) \in X,
\]

\[
y = (y_1, y_2, \ldots, y_n) \in Y
\]

Here, the decision variables \( x_i \) (with \( 1 \leq i \leq m \)) usually are constrained. These
constraints make sure that the decision variables refer to valid system configurations.
(e.g., using not more than the available number of processors, using a valid mapping of application tasks to processing resources, etc.), i.e., $x_i$ are part of the so-called feasible set. In the remainder of this discussion, a minimization procedure is assumed, but without loss of generality, this minimization procedure can be converted into a maximization problem by multiplying the fitness values $y_i$ with $-1$.

With an optimization of a single objective, the comparison of solutions is trivial. A better fitness (i.e., objective value) means a better solution. With multiple objectives, however, the comparison becomes nontrivial. Take, for example, two different MPSoC designs: a high-performance MPSoC and a slower but much cheaper one. In case there is no preference defined with respect to the objectives and there are also no restrictions for the objectives, one cannot say if the high-performance MPSoC is better or the low-cost MPSoC. A MOP can have even more different objectives, like the performance, energy consumption, cost, and reliability of an MPSoC-based embedded system. To compare different solutions in the case of multiple objectives, the Pareto dominance relation is generally used. Here, a solution $x_a \in X$ is said to dominate solution $x_b \in X$ if and only if $x_a < x_b$:

$$x_a < x_b \iff \forall i \in \{1, 2, \ldots, n\} : f_i(x_a) \leq f_i(x_b) \land \exists i \in \{1, 2, \ldots, n\} : f_i(x_a) < f_i(x_b)$$

Hence, a solution $x_a$ dominates $x_b$ if its objective values are superior to the objective values of $x_b$. For all of the objectives, $x_a$ must not have a worse objective value than solution $x_b$. Additionally, there must be at least one objective in which solution $x_a$ is better (otherwise they are equal).

An example of the dominance relation is given in Fig. 2, which illustrates a two-dimensional MOP. For solution $H$, the dominance relations are shown. Solution $H$ is dominated by solutions $B$, $C$, and $D$ as all of them have a lower value for both $f_1$ and $f_2$. On the other hand, solution $H$ is superior to solutions $M$, $N$, and $O$. Finally,
some of the solutions are not comparable to $H$. These solutions are better for one objective but worse for another.

The Pareto dominance relation only provides a partial ordering. For example, the solutions $A$ to $F$ of the example in Fig. 2 cannot be ordered using the ordering relation. Since not all solutions $x \in X$ can be ordered, the result of a MOP is not a single solution but a front of non-dominated solutions, called the Pareto front. A set $X'$ is defined to be a Pareto front of the set of solutions $X$ as follows:

$$\{x \in X' \mid \nexists x_a \in X : x_a < x\}$$

The Pareto front of Fig. 2 contains six solutions: $A$ to $F$. Each of these solutions does not dominate the other. An improvement on objective $f_1$ is matched by a worse value for $f_2$. Generally, it is up to the designer to decide which of the solutions provide the best trade-off.

**Two Basic Ingredients of DSE**

The search for Pareto optimal design points with respect to multiple design criteria as targeted by DSE entails two distinct elements (Gries 2004; Pimentel 2017):

1. The evaluation of a single design point using the fitness function(s) regarding all the objectives in question like system performance, power/energy consumption, and so on
2. The search strategy for covering and navigating through the design space, spanned by the decision variables $x_i$ (with $1 \leq i \leq m$), during the DSE process

Figure 3 shows a simple taxonomy for DSE approaches, recognizing the above two DSE elements as well as different properties of these DSE elements. Please note that these properties typically cannot be considered in pure isolation as they can be interdependent and even conflicting with each other. As will be discussed in more detail in the following sections.
detail later on, there usually exists a trade-off between the accuracy and speed with which the fitness of single design points can be evaluated. In addition to this, the various fitness evaluation techniques also differ with respect to the implementation effort and the capability of evaluating the fitness for a wide range of systems, involving issues such as modularity, reusability of models, etc.

Regarding the search strategy aspect of DSE, the confidence property denotes the degree of certainty that the design points returned by the DSE include the true optimum or, alternatively, how close they are to the true optimum. In many search algorithms, confidence is obtained by avoiding local optima and ensuring sufficient design space coverage. Clearly, an exhaustive search in which every single point in the design space is evaluated and compared would provide a 100% confidence. However, such exhaustive search is usually prohibited due to the sheer size of the design space. In those cases, as will be discussed later on, search techniques based on metaheuristics can be used to search the design space for optimal solutions using only a finite number of design point evaluations. The convergence property denotes the speed of evaluating a range of design points and, more specifically, the rate at which the DSE search algorithm manages to converge to an optimum. Finally, analogous with the effort property in the case of evaluating a single design point, the effort for searching the design space refers to the implementation of the search method and setting its parameters, as well as setting up, running, and evaluating the results of the exploration experiment.

**Y-Chart-Based DSE**

Many system-level fitness evaluation and DSE methods and tools in the embedded systems domain are based on the Y-chart methodology (Kienhuis et al. 2002; Balarin et al. 1997), which is illustrated in Fig. 4. This implies that these DSE methods separate application models (or workload models) and architecture models
while also recognizing an explicit mapping step to map application tasks onto architecture resources (i.e., bind tasks to processing elements in space and time). In this approach, an application model – derived from a specific application domain – describes the functional behavior of an application workload in a timing and architecture independent manner. An MPSoC (platform) architecture model – which usually has been defined with the application domain in mind – defines architecture resources and captures their extra-functional behavior, i.e., behavior in terms of performance, power consumption, cost, etc. To perform quantitative analysis of the fitness of a design point, application models are mapped onto the architecture model under investigation, after which the fitness of each application-architecture combination can be evaluated. Subsequently, the resulting fitness numbers may be used by the search algorithm of a DSE process to change the architecture, restructure/adapt the application(s), or modify the mapping of the application(s). These actions are illustrated by the light bulbs in Fig. 4.

Essential in this methodology is that an application model is independent from architectural specifics, assumptions on hardware/software partitioning, and timing characteristics. As a result, application models can be reused in the exploration cycle. For example, a single-application model can be used to exercise different hardware/software partitionings or can be mapped onto a range of architecture models, possibly representing different MPSoC architecture designs or modeling the same architecture design at various levels of abstraction. The latter refers to the gradual refinement of architecture models (e.g., Pimentel et al. 2006; Thompson et al. 2006). As design decisions are made, a designer typically wants to descend in abstraction level by disclosing more and more implementation details in an architecture model. Eventually, such refinement can bring an initially abstract

---

**Fig. 4** Y-chart-based DSE (Kienhuis et al. 2002; Balarin et al. 1997)
architecture model closer to the level of detail where it is possible to synthesize an implementation (Nikolov et al. 2008; Thompson et al. 2007; Stefanov et al. 2017).

In the next two sections, a more detailed overview will be provided of the different techniques, and their properties, applied in each of the two aforementioned elements of DSE, i.e., fitness evaluation of a single design point and searching the design space.

**Evaluation of a Single Design Point**

Methods for evaluating the fitness of a single design point in the design space roughly fall into one of three categories: (1) measurements on a (prototype) implementation, (2) simulation-based evaluations, and (3) estimations based on an analytical model. Each of these methods has different properties with regard to evaluation time and accuracy. Evaluation of prototype implementations provides the highest accuracy, but long development times prohibit evaluation of many design options. Analytical estimations are considered the fastest, but accuracy is limited since they are typically unable to sufficiently capture particular intricate system behavior. Simulation-based evaluation fills up the range in between these two extremes: both highly accurate (but slower) and fast (but less accurate) simulation techniques are available (see also chapter “Processor Simulation and Characterization”). This trade-off between accuracy and speed is very important, since successful DSE depends both on the ability to evaluate a single design point and being able to efficiently search the entire design space. As present DSE efforts in the domain of embedded systems design usually use simulation or analytical models to evaluate single design points, the remainder of this section will focus on these methods.

**Simulative Fitness Evaluation**

Simulating system components can be performed at different levels of abstraction. The higher the abstraction level, the less intricately the system components are modeled and, therefore, the higher the simulation speed is. Evidently, such efficiency improvements come at the cost of a less accurate fitness estimation because of the fact that particular system details are not taken into account. This simulation speed-accuracy trade-off is shown in Fig. 5. This figure depicts several widely used simulation abstraction levels, and it does so for both the simulation of processor components and the simulation of communication between system components.

For both the simulation of processor and communication components, the lowest level of abstraction for simulating a digital system is the register-transfer level (RTL). At this level of abstraction, the flow of digital signals between registers and combinational logic is explicitly simulated. This yields a highly accurate but also very slow simulation. As a result, the use of RTL simulation in the process of DSE is confined to only relatively small and narrow design spaces focusing on,
for example, the design of one specific system component. Performing system-level DSE is infeasible using RTL simulation.

Raising the level of abstraction, one can simulate system components at the cycle-accurate level. This means that the system components are simulated on a cycle-by-cycle basis and, as such, that the simulated system state conforms to the cycle-by-cycle behavior of the target design. This results in more efficient simulation as compared to RTL simulation at the cost of a somewhat reduced accuracy since the system state in between cycles is not accounted for. Cycle-accurate simulation is a popular technique for simulating microprocessors (see also chapter “Processor Simulation and Characterization”): so-called cycle-accurate instruction set simulation (ISS). These ISS simulators try to capture the cycle-by-cycle behavior of the micro-architectural components of a microprocessor, such as the pipeline logic, out-of-order processing, branch predictors, caches, and so on. To account for power consumption behavior, ISS simulators often use activity-based power models that accumulate the power consumption of the relevant micro-architecture components based on their activity ratio. A good example is the widely used cycle-accurate Gem5 ISS (Binkert and et al.: 2011), which can be extended to also support area and power predictions using activity-based modeling frameworks such as CACTI (Thoziyoor et al. 2008) and McPAT (Li et al. 2013). Although these ISS simulators can be deployed to perform micro-architectural DSE for processor components, they are generally still too slow for performing full system-scale DSE of multicore-based embedded systems.

In cycle-accurate ISS simulators, the fetching, decoding, and execution of instructions are explicitly simulated. To further optimize the speed of such simulators, one could translate the instructions from the target binary to be simulated to an equivalent sequence of instructions (using static or dynamic just-in-time translation) that can be executed on the simulation host computer. This so-called binary translation technique, which is, for example, deployed in the widely used QEMU
simulator (Bellard 2005), aims at reducing the overhead of explicitly simulating the instruction fetch and decode stages. The translated instruction sequences are often instrumented with additional code to keep track of the extra-functional behavior such as timing and power consumption, of the original code as it would have been executed on the target processor. In some cases, however, ISS simulators and especially binary translation-based simulators only focus on mimicking the functional behavior and do not capture the extra-functional behavior of the target processor. In these cases, they are usually referred to as emulators rather than simulators.

For simulating communication between system components, one could use so-called bus-cycle-accurate simulation (Cai and Gajski 2003) to speed up the simulation process. In this type of simulation, all signals of the communication bus are modeled explicitly in a cycle-accurate fashion, but this accuracy is only maintained for the signals on the communication bus and not for the logic around it. The surrounding components can thus use more abstract timing models.

Raising the abstraction level even further for processor simulation yields so-called host-compiled simulation (Ceng et al. 2009; Bringmann et al. 2015). In this technique, the source code of the target program is directly compiled into a binary program that can run on the host computer. In addition, and similar to the binary translation technique, the source code can be instrumented with a timing and power consumption model based on the target architecture. Since this type of simulation is efficient as it directly executes target programs on the host computer, it is very suitable for system-level DSE. However, at this level of abstraction, it is difficult to accurately capture intricate micro-architectural behavior, like pipeline and caching behavior. Another drawback of this simulation approach is that one needs to have access to the source code of a target program.

For simulating communications, transaction-level modeling (TLM) (Cai and Gajski 2003) provides the highest level of abstraction. In TLM, communication details at the level of signals and protocols are abstracted away by means of encapsulation into entire transactions between system components. At this level, the emphasis is more on the functionality of the data transfers, i.e., what data are transferred to and from what locations, rather than on their actual implementation. Evidently, the extra-functional behavior in TLM simulation models is also captured at the level of entire transactions.

The above processor simulation techniques are all execution-driven simulation methods as they are directly driven by the execution of a program. Alternatively, there are also trace-driven simulation techniques in which the simulation is driven by event traces that have been collected through the execution of a program (e.g., Butko et al. 2015; Castrillon et al. 2010). These trace events can focus on the evaluation of specific system elements such as memory access address traces for cache simulation (Uhlig and Mudge 1997). However, an event trace may also consist of the full sequence of executed instructions, thereby allowing full, trace-driven microprocessor simulation for the purpose of performance and/or power estimation. To optimize for simulation speed, the trace events may also represent computations
Methodologies for Design Space Exploration

(and, if needed, communications) at a higher level of abstraction than the level of
machine instructions, like at the level of the execution of basic blocks or even entire
functions. Another advantage of trace-driven simulation is the fact that the event
traces often only need to be generated once (i.e., executing the program to collect
the traces once), after which they can be reused in the DSE process. Drawbacks of
trace-driven simulation evidently are the need for storing the event traces which can
become extremely large in size and the fact that trace-driven simulation does not
allow for simulating all intricate system behavior, such as the effects of speculative
instruction execution in microprocessors.

An example of a high-level, trace-driven MPSoC simulation environment is the
Sesame system-level modeling and simulation framework (Pimentel et al. 2006;
Erbas et al. 2007). Sesame is based on the aforementioned Y-chart methodol-
ogy (Kienhuis et al. 2002), and accordingly it recognizes separate application
and architecture models. The application models are explicitly mapped onto the
architecture models by means of trace-driven simulation. The workload of an
application is captured by instrumenting the application model – which is a parallel
specification of the application – with annotations that describe the application’s
computational and communication actions at a coarse-grained level (typically at
the level of the execution of entire functions). By executing this instrumented
application model, these annotations cause the generation of traces of application
events that subsequently drive the underlying architecture model. This architecture
model – capturing the system resources and their constraints – then simulates
the consequences of the consumed computation and communication events in
terms of extra-functional system behavior (performance, power consumption, etc.).
Figure 6 depicts Sesame’s layered organization, illustrating the mapping of two
multimedia applications (an MP3 encoder and video decoder) onto a bus-based
MPSoC platform. A special mapping layer in Sesame, which can be seen as an
abstract (real-time) operating system (RTOS) model, provides the scheduling of
application events in the case multiple application processes are mapped onto a
single processing resource.

Orthogonal to most of the (processor) simulation methods described above, there
are additional techniques to further improve the simulation speed (Eeckhout 2010).
In sampled simulation, for example, the simulation does not cover the execution
of an entire program but only simulates relatively small samples of the program’s
execution. Here, the challenge is to select the samples in such a manner that they
sufficiently represent the behavior as if the entire program was simulated. Another
technique for speeding up simulation is statistical simulation. Rather than using
real (benchmark) programs for simulation, it uses a statistical program profile. This
profile captures the distributions of important program characteristics and is used for
generating a synthetic instruction trace that drives a simple trace-driven simulator.
As the synthetic trace is randomly generated from a statistical profile, this type of
simulation can converge to a set of performance predictions fairly quickly.
Analytical Fitness Evaluation

In comparison to simulation, analytical models allow for much more efficient prediction of the extra-functional system behavior at the expense of a reduced accuracy. This makes analytical models very suitable for exploring large design spaces and to rapidly identify regions of interest that can be later explored in more detail using simulation. Another advantage of analytical models is that they can provide direct insight into the relationship between model parameters (representing design choices) and the predicted extra-functional behavior. For simulative methods, such understanding would require a large number of simulation runs.

Analytical models can roughly be divided into three classes (Eeckhout 2010): (1) mechanistic (or whitebox) models, (2) empirical (or blackbox) models, and (3) a hybrid combination of mechanistic and empirical modeling. Mechanistic models are based on first principles, which implies that they are built in a bottom-up fashion starting from a basic understanding of the mechanics of the modeled system. For example, in a mechanistic microprocessor performance model, penalties due to cache misses, branch mispredictions, the execution of instructions with different latencies, etc., are explicitly captured in the model.
In empirical models, statistical inference and machine learning techniques, like regression models or neural networks, are used to automatically synthesize a model through the process of learning from training data. For example, using a set of micro-architectural parameters such as pipeline depth, issue width, cache sizes, etc., one could train a model that predicts the instructions per cycle (IPC) or cycles per instruction (CPI) of a microprocessor. Inferring a model by means of automatic training typically is easier than developing a mechanistic model because it does not require intimate understanding of the mechanics of the modeled system. Evidently, the latter is also an immediate drawback as empirical models also tend to provide less insight than mechanistic models.

In hybrid mechanistic-empirical modeling, which is sometimes referred to as greybox modeling, extra-functional system aspects are captured using a formula that has been derived from insights in the underlying system. However, this formula includes a number of unknown parameters, which are then inferred through fitting (e.g., using regression), similarly to empirical modeling. Such hybrid mechanistic-empirical modeling is motivated by the fact that it provides insight (like mechanistic modeling) while easing the construction of the model (like empirical modeling).

Searching the Design Space

As explained before, searching a design space is a multi-objective optimization process. This process will evidently benefit from a good trade-off between speed, accuracy, and effort of the method for evaluating the fitness of a single design point, as discussed in the previous section. But, even if this trade-off is ideal, it should still be ensured that each evaluation of a design point contributes as much as possible to an effective and efficient search of the design space. A crucial component toward this goal is the search algorithm that navigates through the design space toward areas of interest by proposing which design points to evaluate next. Regardless of the specific type of search method that is used for such a design space traversal, its success depends on three major concerns, as was shown in Fig. 3: confidence, convergence, and effort. As was already mentioned earlier, these concerns typically cannot be considered in isolation, as they are highly interdependent, contradictory, and sometimes overlapping. The state of the art in DSE can be summarized as finding a good trade-off between these concerns.

DSE search algorithms can be divided into exact and non-exact methods. In exact DSE methods, like those implemented using integer linear programming (ILP) solutions (e.g., Niemann and Marwedel 1997; Lukasiewycz et al. 2008) or branch and bound algorithms (e.g., Padmanabhan et al. 2011), the optimum is guaranteed to be found. As such methods generally are compute intensive, they typically use design space pruning (i.e., discarding unsuitable design points) to optimize the efficiency of the search, thereby allowing them to handle larger design spaces. However, for realistic design problems with design spaces that are vast, these methods may still not scale and thus be less suited. Alternatively, in non-exact methods, metaheuristics are typically used to find a design point in the
known design space that meets the design requirements as best as possible. To this end, these methods search the design space for optimal solutions using only a finite number of design point evaluations and can thus handle larger design spaces. However, there is no guarantee that the global optimum will be found using meta-heuristics, and therefore the result can be a local optimum within the design space. Examples of metaheuristics are hill climbing, tabu search, simulated annealing, ant colony optimization, particle swarm optimization, and genetic algorithms (GA) (Panerati et al. 2017). In this chapter, the focus will be on methods to navigate the design space that are based on GA. GA-based DSE has been widely studied in the domain of system-level embedded design (e.g., Palesi and Givargis 2002; Madsen et al. 2006; Erbas et al. 2006; Quan and Pimentel 2014; Goens et al. 2016) and has demonstrated to yield good results. Moreover, GAs can be used in their basic (domain-independent) form or, as will also be explained later on, with custom extensions that incorporate domain-dependent knowledge in order to improve search performance even further.

**GA-Based DSE**

GAs operate by searching through the solution space (spanned by the design variables/decisions being explored) where each possible solution is encoded as a string-like representation, often referred to as the chromosome (Beasley et al. 1993). A (randomly initialized) population of these chromosomes is then iteratively modified by performing a fixed sequence of actions that are inspired by their counterparts from biology: fitness evaluation and selection, crossover, and mutation. A fundamental design choice of a GA is the genetic representation of the solution space, because each of the crossover and mutation steps depends on it. To illustrate how such a genetic representation could look like, let us use a widely studied DSE problem in the domain of system-level embedded systems design as an example: optimizing the mapping of a (set of) concurrent application(s) onto an underlying (heterogeneous) MPSoC platform architecture (Singh et al. 2013). As a convenient mapping description for an application with $n$ tasks, a vector of size $n$ is used with processor identifiers $p_i$, where $p_i$ indicates the mapping target of task $i$:

$$[p_0, \ldots, p_i, \ldots, p_{n-1}]$$

This commonly used description is very suitable to serve as the chromosome representation for a GA. A valid mapping specification is a feasible partitioning of all $n$ tasks. In this context, “feasible” means that tasks are mapped onto processing elements that can execute those tasks (i.e., there are no functional restrictions of the processing element in question, like an ASIC component which only allows the execution of one particular piece of functionality) and that communicating tasks are mapped onto processing elements that can actually communicate with each other (i.e., there are no topological communication restrictions). In case an infeasible mapping is created by the genetic operators of a GA (crossover and mutation), a
Fig. 7 GA-based mapping DSE: (a) general overview of the GA steps and (b) crossover and mutation operators

mechanism is required that either discards or repairs such a chromosome. Repairing a chromosome implies that it is transformed into a valid chromosome (mapping) that is “as close as possible” to the original, invalid one. Moreover, note that task partitions specifying a mapping may also be empty (i.e., particular processor(s) not in use) or contain all $n$ tasks (i.e., a single processor system). A processor that is not assigned any tasks (having an empty task partition) can be considered idle or nonexistent.

In Fig. 7a, the different steps of a GA are shown. This figure also illustrates the mapping representation of a chromosome for an application with six tasks and a four-processor bus-based MPSoC platform. Starting from a (randomly initialized) population of chromosomes, representing the different mapping design instances, the fitness of the mapping solutions in the population is first evaluated. To this end, any of the analytical or simulative techniques discussed in section “Evaluation of a Single Design Point” can be used. Subsequently, based on the fitness evaluation, a selection of chromosomes is made that will be used to create offspring. This offspring is created by combining genetic material from two parents using a crossover operation, as illustrated in the top part of Fig. 7b. There exist various forms of this crossover operator, of which the uniform, one-point, and two-point crossovers are the most popular. Next, new genetic material is introduced in the offspring by means of a mutation operator as illustrated at the bottom of Fig. 7b. Such a mutation randomly changes one or more genes within chromosomes. Finally, the newly created offspring is used to update the population by either replacing it
or by deploying so-called elitism. Such elitism involves the combination of the new offspring with a small number of the best solutions from the original population to avoid losing strong solutions.

To provide a small example of the results a GA-based DSE could obtain, some results are presented of a small-scale case study where the design space consists of an application with 11 tasks that is to be mapped onto a four-core MPSoC architecture with a crossbar interconnect (Thompson 2012). The mapping design space contains more than 4 million design points. Of these design points, 175K are unique ones since the target platform is a homogeneous, symmetric MPSoC and, as a consequence, exhibits mapping symmetries. Because of the relatively small design space, in this particular case, it was also possible to perform an exhaustive search, allowing a quality evaluation of the GA-based search results. To account for the stochastic behavior of GAs, all results are averages over 300 GA runs. The fitness of mapping solutions has been evaluated using the Sesame MPSoC simulation framework (Pimentel et al. 2006; Erbas et al. 2007) (see also section “Simulative Fitness Evaluation”). Figure 8 shows the results of the GA-based DSE with different population sizes (10, 15, 40, or 80 chromosomes), a constant mutation rate (0.1) and crossover probability (0.9), and a uniform crossover in a so-called P-Q (probability-quality) plot. Regarding the top part of this plot, the horizontal axis (top x-axis) represents the quality of the result as a percentile toward the true optimum (a lower percentile indicates a result closer to the optimum), and the vertical axis represents the probability of achieving a result with that quality. The straight lines in the graph represent the theoretically derived probabilities of finding results using a simple, uniform random search. The 80–95% confidence intervals of the mean fitness value (execution time in cycles, in this case) of mapping solutions found by the GA were also computed, averaged over the 300 runs of each GA search. These

![Fig. 8 P-Q plot for GA-based DSE with different population sizes](image-url)
confidence intervals, shown at the bottom of the graph in Fig. 8, indicate the degree of certainty (as specified by the confidence level) that the real mean lies within the confidence interval. The more the confidence intervals for different experiments are nonoverlapping, the more significant the difference of the mean behavior (which is clearly the case in the example of Fig. 8). The results from this particular case study show that the GA-based DSE with the largest population size can find mapping solutions that are always very close to the real optimum: within the 0.1 percentile, implying that they belong to the best 175K/1000 = 175 solutions. A larger population size, however, comes with a higher number of fitness evaluations during the search and thus requires a longer search time (assuming the number of search iterations remains constant). According to Fig. 8, a population size of 40 may therefore provide a good compromise.

**Optimizing GA-Based DSE**

There are various methods for making the search process of a GA-based DSE more efficient. This allows the DSE process to either find the design candidates quicker (i.e., improve the convergence behavior of the DSE) or to spend the same amount of time to evaluate more design points. The latter can be used to enable the search of larger design spaces or to improve the chance of finding better design candidates (i.e., improve the confidence property of the DSE). One approach for optimizing the GA-based search is to enrich the genetic operators of the GA with domain knowledge such that they produce more diverse offspring or offspring with a higher probability of being closer to the optimum. For example, in Thompson and Pimentel (2013), new GA operators have been proposed that optimize the search performance by (1) reducing the redundancy present in chromosome representations (e.g., mapping symmetries (Goens et al. 2017) in the case of homogeneous, symmetrical MPSoC platforms) or (2) using a new crossover operator that is based on a mapping distance metric that provides a measure of similarity between mappings. Using this mapping distance information, the new crossover operator aims at retaining the strong chromosome parts of both of the parents. In Quan and Pimentel (2014), a new mutation operator has been proposed that considers the affinity of tasks with respect to processors, the communication cost between tasks, and the differences of processor workloads to steer the mutation in such a way that offspring is produced with a higher probability of being (near-)optimal.

Another approach for optimizing GA-based DSE concerns the reduction of the time taken to evaluate the fitness of solutions during the GA’s execution. As mentioned before, DSE approaches typically use either simulation or an analytical model to evaluate the fitness of design points, where simulative approaches prohibit the evaluation of many design options due to the higher evaluation performance costs and analytical approaches suffer from accuracy issues. Therefore, in Piscitelli and Pimentel (2012a), a hybrid form of mapping DSE has been proposed that combines simulation with analytical estimations to prune the design space in terms of application mappings that need to be evaluated using simulation. To this end,
the DSE technique uses an analytical model that estimates the expected throughput of an application given a certain architectural configuration and application-to-architecture mapping. In the majority of the search iterations of the DSE process, this analytical throughput estimation avoids the use of simulation to evaluate the design points. However, since the analytical estimations may in some cases be less accurate, the analytical estimations still need to be interleaved with simulative evaluations in order to ensure that the DSE process is steered into the right direction (Piscitelli and Pimentel 2012b). A similar approach is taken in Mariani et al. (2010), where an iterative DSE methodology is proposed exploiting the statistical properties of the design space to infer, by means of an empirical analytic model, the design points to be analyzed with low-level simulation. The knowledge of a few design points is used to predict the expected improvement of unknown configurations.

Alternatively, in hierarchical DSE (e.g., Mohanty et al. 2002; Jia et al. 2013, 2014), DSE is first performed using analytical or symbolic models to quickly find the interesting parts in the design space. Hereafter, simulation-based DSE is performed on the selected sweet spots in the design space to more accurately search for the optimal design points.

**Multi-application Workload Models**

The DSE techniques discussed so far focus on the evaluation and exploration of MPSoC architectures under static, single-application workloads. Today’s MPSoC systems, however, often require supporting an increasing number of applications and standards, where multiple applications can run simultaneously and concurrently contend for system resources (Thompson and Pimentel 2007; Castrillon et al. 2013). For each single application, there may also be different execution modes (or program phases) with different computational and communication requirements. For example, in software-defined radio appliances, a radio may change its behavior according to resource availability, such as the long-term evolution (LTE) standard which uses adaptive modulation and coding to dynamically adjust modulation schemes and transport block sizes based on channel conditions. Or a video application could dynamically lower its resolution to decrease its computational demands in order to save battery life. As a consequence, the behavior of application workloads executing on the embedded system can change dramatically over time.

As illustrated in Fig. 9, there are several approaches for dealing with multi-application workloads in the context of DSE. A commonly used approach is to consider the applications in isolation, as illustrated in Fig. 9a. This implies that each of the applications in the multi-application workload will be mapped to a different, isolated part of the system. As a consequence, the DSE for each of these applications can also be performed in isolation. However, this approach typically leads to overdesigned systems since there is no or limited resource sharing between applications. Another approach, illustrated in Fig. 9b, makes the pessimistic assumption that all applications that can be executed on the system will always be active (and will thus be contending for system resources). Again,
performing DSE with such an assumption may lead to highly overdesigned systems, as in reality the concurrent activation of all possible applications may be unlikely. To address the problem of overdesigning systems and to capture the dynamism in application workload behavior during the design process, the DSE could employ the concept of application scenarios (Gheorghita et al. 2009), leading to scenario-based DSE (van Stralen and Pimentel 2010a, 2013; Pimentel and van Stralen 2017; Castrillon et al. 2013). This is illustrated in Fig. 9c. The remainder of this section will discuss the concepts of application scenarios and scenario-based DSE, again using the example of application mapping exploration for illustration purposes.

**Scenario-Based DSE**

Application scenarios are able to describe the dynamism of embedded applications and the interaction between the different applications on the embedded system. 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 for each individual application. The concept of application scenarios, inter-application scenarios, and intra-application scenarios is illustrated in Fig. 10 for

---

**Fig. 9** DSE for multi-application workloads on a three-core, bus-based MPSoC: (a) DSE with application isolation, (b) pessimistic DSE, and (c) scenario-based DSE
three multimedia applications (mp3 player, video decoder, and gsm application) with each two application modes.

The number of different application scenarios grows exponentially with the number of applications involved. So, to perform DSE with these application scenarios, scenario-based DSE needs to solve the problem that the total number of possible application scenarios is too large to exhaustively evaluate the fitness of design points with all of these scenarios. Therefore, a small but representative subset of application scenarios must be selected for the evaluation of MPSoC design points. This representative subset must be used for comparing 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. That is, if mapping $m_1$ is better than mapping $m_2$, the representative subset should be able to give a better predicted fitness to mapping $m_1$ than it assigns to mapping $m_2$. However, the selection of such a representative subset is not trivial (Pimentel and van Stralen 2017). This is because the representative subset is dependent on the current set of mappings that are being 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 be statically 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 co-explored simultaneously such that the representative subset is able to adapt to the set of mappings that are currently being explored (van Stralen and Pimentel 2010a, 2013; Pimentel and van Stralen 2017). Figure 11 shows the scenario-based DSE framework. The left part of the picture provides a general overview of the exploration flow, whereas the right part illustrates the scenario-based DSE in more detail. As input, the scenario-based DSE requires a database of application
scenarios, application models, and an MPSoC platform architecture model. The description of the application workload is split into two parts: (1) the structure and (2) the behavior. The structure of applications is described using application models (as described before), whereas a scenario database (van Stralen and Pimentel 2010b) explicitly stores all the possible multi-application workload behaviors in terms of application scenarios (i.e., intra- and inter-application scenarios). In the scenario-based DSE framework, two separate components are recognized that simultaneously perform the co-exploration tasks: the design explorer searches for the set of optimal mappings, while the subset selector tries to select a representative subset of scenarios. To this end, they exchange data in an asynchronous fashion after every search iteration. Here, the design explorer sends a sample of the current mapping population to the subset selector, whereas the subset selector makes the most representative subset available for the fitness prediction in the design explorer.

The design explorer performs a traditional mapping DSE using a GA, as discussed in section “Searching the Design Space”. As explained above, it uses a representative subset of scenarios to evaluate the fitness of mapping solutions. At every iteration of the GA, the design explorer reads in the most recent representative scenario subset from the subset selector and submits the current population of mapping solutions to the subset selector in order to allow the latter to select the appropriate representative subset. This subset selection is not trivial as there are many scenarios to pick from, leading to a huge number of possible scenario subsets. Therefore, the subset selector uses the set of mappings it regularly receives from the design explorer 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 representative subset will change accordingly. In van Stralen and Pimentel (2013), three different techniques for selecting a representative scenario subset are presented and evaluated: a GA-based scenario space search (which means that two GAs are running concurrently, one for the design explorer and one for the subset selector), a feature selection (FS)-based search algorithm, and a hybrid combination (HYB) of these two. The latter aims at combining the strengths of both the GA-based and FS-based searches. That is, a GA 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 FS algorithm as it more systematically explores the local neighborhood of a scenario subset.

To give a feeling of the performance of the three different fitness prediction techniques, Fig. 12 shows the results of a scenario-based DSE experiment in which the three techniques are compared for three different scenario 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 tasks 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, a DSE with a fixed duration of 100 min is performed for all three subset selector approaches. The results have been averaged over nine runs. To evaluate the fitness of mapping solutions, the Sesame MPSoC simulation framework (see section “Simulative Fitness Evaluation”) is again deployed. To determine the efficiency of the multi-objective DSE, the distance of the estimated Pareto front (execution time versus energy consumption of mapping solutions) to the optimal Pareto front is obtained. For this purpose, the execution time and energy consumption are normalized to a range from 0 to 1. As the optimal Pareto front is not exactly known since the design space is too large to be exhaustively searched, the combined Pareto front of all performed experiments is used for this.

Fig. 12 Quality of the scenario-based DSE for the different subset selection approaches. The quality is determined based on the distance between the estimated Pareto front and the optimal front.
The size of the scenario subset provides a trade-off between accuracy and convergence of the search. That is, a larger scenario subset will lead to a more accurate fitness prediction of mappings in the design explorer at the cost of a larger computational overhead to obtain the fitness of a single mapping causing a slower convergence of the search. This can be seen in Fig. 12. The GA and the FS subset selection methods have worse results when the subset becomes larger (remember that a fixed DSE duration of 100 min is used). For a subset size of 4%, the hybrid selector is, however, still 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 FS and the GA.

Application Exploration

As described in section “Y-Chart-Based DSE”, the Y-chart methodology is a popular approach for system-level DSE in the domain of embedded systems. This means that exploration can take place to investigate (the fitness of) different MPSoC architectures and different mappings of application tasks to the underlying architecture but also different application implementations. With respect to the latter, one could, for example, explore the use of different algorithms for implementing a certain piece of application functionality or vary the degree of concurrency in an application (e.g., fine-grained versus more coarse-grained concurrency) that can be exploited by an underlying MPSoC platform. So far, the discussion was limited to the first two types of exploration, i.e., exploring different MPSoC architectures and task-to-architecture mappings. In this section, an example of application exploration will be described, and this will be done for the popular application domain of deep learning. More specifically, an approach will be outlined for so-called neural architecture search (NAS) (Sapra and Pimentel 2020a, b), which automates the discovery of an efficient neural network for a given task, such as image/video recognition, classification, natural language processing, etc.

NAS by Means of Evolutionary Piecemeal Training (EPT)

The NAS approach that will be described searches for an efficient convolutional neural network (CNN) architecture. To this end, it leverages a GA, which allows a group of candidate CNNs in the GA’s population to train in parallel. In most NAS techniques, training of a neural network is considered a separate task or a performance estimation strategy to perform the neural network architecture search. However, the approach described here considers NAS from a different perspective as it aims at finding optimal CNN architectures during the training process itself.
as opposed to accuracy prediction or training as a separate performance estimation strategy. The NAS approach is called EPT (Sapra and Pimentel 2020a,b), where piecemeal training refers to training a neural network with a small “data piece” of size $\delta_k$. In this technique, a traditional continuous training is interceded by an evolutionary operator at regular intervals, and the interval of intervention is dictated by the value of $\delta_k$. The evolutionary operators applied to CNNs in the GA population lead to CNN architecture modifications and hence exploration of the search space. A new CNN architecture derived like this is always partially trained already as it was modified from another CNN undergoing training. In subsequent iterations, derived CNN architectures continue to train. Those CNN candidates that are not able to achieve high accuracy during training will be dropped from the population. This can also be seen as early training termination of candidates that are performing poorly. Toward the end of this algorithm, the best candidates are selected from the population, which can then be post processed or trained further, if needed.

The search space for the algorithm is focused on plain CNNs, which consist of convolutional, fully connected (FC) and pooling layers without residual connections, branching, etc. Batch normalization and nonlinear activations are configurable and can be added to the network. CNN architectures are defined by a group of blocks, where each block is of a specific layer type and is bounded by minimum and maximum number of layers it can have. Additionally, each layer has upper and lower bounds on each of its parameters. For example, a convolutional layer will have bounds on the number of units, kernel sizes, and stride sizes possible. These constraints are in place to make sure that CNN architectures do not become too big and limit the resource consumption of the final neural network. This is an important factor to consider when mapping the CNNs to resource-constrained embedded systems. The search space specifications along with its bounds are encoded as a collection of genes, also called a genotype. All possible combinations of parameters together form the gene pool from which individual neural networks are created and trained.

A population-based training process is used where an initial population of neural networks is randomly created from the defined gene pool. In each iteration, all candidates of the population are piecemeal-trained and then evaluated using the validation set. Depending on the available resources, all candidates can be trained in any combination of parallel and sequential manner. The size of the population is kept constant throughout the algorithm, though the candidates of the population keep changing as they are altered through the evolutionary operators applied in each iteration. The number of candidates in the population needs to be large enough to maintain enough diversity of CNN architectures in the population while still satisfying the constraints applied to it.

**Evolutionary Operators**

The crossover operator in the EPT-based NAS works with two neural networks and swaps all layers in a gene block of the same type. In this replacement, the layers being swapped are roughly in the same phase of feature extraction. The input and output feature map sizes of the layer block being swapped are also identical
in both of the selected networks. Figure 13 illustrates the crossover operator for swapping convolutional layers from two networks. Crossover is not a function preserving operator, but in experiments they were found to be important to introduce diversity in the population by changing the total number of layers in a candidate through swapping. To reduce the negative effect of training loss incurred due to the crossover, a cooling-down approach is used to the crossover rate. In earlier GA iterations, where the training loss is already high, there are more swaps happening than in the later ones, where training loss is very low.

The mutation operator changes a layer’s parameters such as the number of kernels or kernel size and is designed to be function preserving. Every mutation disrupts the ongoing training of the mutated candidates, and some additional loss is incurred in the training in process. However, due to the function preserving nature of the mutation operator, the loss incurred from this operator is as small as possible and recoverable in later piecemeal training.

**NAS Results**

To illustrate the competence and versatility of the EPT-based NAS concept, a range of experiments were performed with datasets from two different domains: CIFAR-10 for image classification (Deng et al. 2009) and PAMAP2 for human activity recognition (Reiss and Stricker 2012). For CIFAR-10, the search took
Table 1 CIFAR-10 Accuracy comparisons with evolutionary approaches

<table>
<thead>
<tr>
<th>Model</th>
<th>Search space</th>
<th>GPU-days</th>
<th>Accuracy(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>GeneticCNN (Xie and Yuille 2017)</td>
<td>Hybrid</td>
<td>17</td>
<td>92.9</td>
</tr>
<tr>
<td>EANN-Net (Chen et al. 2019)</td>
<td>Hybrid</td>
<td>–</td>
<td>92.95</td>
</tr>
<tr>
<td>AmoebaNet (Real et al. 2019)</td>
<td>Cell</td>
<td>3150</td>
<td>96.6</td>
</tr>
<tr>
<td>NSGANet (Lu et al. 2018)</td>
<td>Hybrid</td>
<td>8</td>
<td>96.15</td>
</tr>
<tr>
<td>Evolution (Real et al. 2017)</td>
<td>Hybrid</td>
<td>1000+</td>
<td>94.6</td>
</tr>
<tr>
<td>EPT</td>
<td>Plain CNN</td>
<td>2</td>
<td>92.5</td>
</tr>
</tbody>
</table>

2-GPU days, and the best prediction accuracy was found to be 92.5% on the test set. Table 1 shows comparisons with other evolutionary NAS approaches, where EPT refers to evolutionary piecemeal training. It may seem that 92.5% is relatively low compared to other published works, but this is on a very simple and plain CNN without any architectural enhancements or advance data augmentation. Other approaches use a hybrid search space where different architecture blocks or cell modules as well as arbitrary skip connections are used. Instead of stacking conventional layers, these stack different blocks. The best model found in the EPT experiments has 13 convolutional layers followed by two fully connected layers. For the PAMAP2 dataset, the EPT search took only 10 GPU-hours, and the best prediction accuracy was 94.36%. Compared to state-of-the-art neural network solutions for this particular dataset, EPT outperforms all other known efforts. The best performance was found on a neural network that has seven convolutional layers followed by three fully connected layers. The interested reader is referred to Sapra and Pimentel (2020a) for a more detailed analysis of EPT’s experimental results.

Conclusion and Outlook

In this chapter, an overview was presented of techniques and methods for DSE of embedded systems. The discussion was organized along the lines of the two primary elements of DSE: the evaluation of single design points and the search strategy for covering the design space. The overview is certainly not meant to be exhaustive. For example, the discussion mainly focused on popular GA-based DSE, optimizing system performance and, to some extent, power/energy consumption. The optimization of other important design objectives, such as system reliability (e.g., addressed in Jhumka et al. 2005; Glaß et al. 2007, 2008; van Stralen and Pimentel 2012), has not been covered.

There are still many open research challenges for this domain. For example, embedded systems more and more need to become adaptive systems due to increasingly dynamic application workload behavior (as was previously discussed); the need for quality-of-service management to dynamically trade off different system qualities such as performance, precision, and power consumption; and the
fact that a technology level is reached where digital circuits are no longer fully reliable, increasing the chances of transient and permanent faults. This calls for research to take system adaptivity, in which a system can continuously reconfigure and customize itself at run time according to the application workload at hand and the state of the system (e.g., Singh et al. 2013; Quan and Pimentel 2015, 2016a,b; Goens et al. 2017; Khasanov and Castrillon 2020), into account in the process of DSE. In the case of adaptive systems, a DSE process cannot easily compare different design choices by, e.g., simply evaluating the performance or power/energy consumption of an application workload executing on a specific platform architecture. That is, the reconfiguration behavior (i.e., when and how the system reacts to “disruptive events” that trigger system reconfigurations) of the system and the performance/power consumption consequences of such system adaptivity actions must be taken into account when comparing different design instances. This calls for efficient and effective methods that allow for evaluating and optimizing adaptive embedded systems designs such that the way the system instances and their extra-functional behavior evolve over time is also captured.

Another research direction that is worth mentioning involves the introduction of new design objectives in the process of (early) DSE, in addition to the traditional objectives such as system performance, power/energy consumption, system cost, and reliability. Arguably, a good example is the need for taking system security into account as an optimization objective (Pimentel 2020). As embedded systems are becoming increasingly ubiquitous and interconnected, they attract a worldwide attention of attackers, which makes the security aspect more important than ever during the design of those systems. Currently, system security is still mostly considered as an afterthought and is typically not taken into account during the very early design stages. However, any security measure that may eventually be taken later in the design process does affect the already established trade-offs with respect to the other system objectives such as performance, power/energy consumption, cost, etc. Thus, covering the security aspect in the earliest phases of design is necessary to design systems that are, in the end, optimal with regard to all system objectives. However, this poses great difficulties because unlike the earlier mentioned conventional system objectives like performance and power consumption, security is hard to quantify. This necessitates research on techniques that make it possible to incorporate security as an objective in early DSE.

At this moment, the integration of security aspects in the process of system-level DSE of embedded systems is still largely uncharted research ground. Only a few efforts exist that address this problem, but they typically provide only partial solutions or solutions to very specific security problems (e.g., Lin et al. 2015; Weichslgartner et al. 2016; Stierand et al. 2014; Tan et al. 2017). Moreover, in most of these works, security is modeled as a requirement in the DSE process, which does not allow for studying actual trade-offs between performance, power consumption, and cost in relationship to secureness of a design. Only a handful of research efforts, such as Ferrante et al. (2013) and Gressl et al. (2019), seem to have been aiming at incorporating security as an objective that can be traded off with other objectives during early DSE.
References


Jia ZJ, Núñez A, Bautista T, Pimentel AD (2014) A two-phase design space exploration strategy for system-level real-time application mapping onto MPSoC. Microprocessors Microsyst 38(1):9–21


