

# Multi-criteria Decision Making in Embedded System Design

Simon Künzli, Lothar Thiele and Eckart Zitzler \*

## Abstract

This chapter introduces design space exploration as one of the major tasks in embedded system design. After reviewing existing exploration methods at various layers of abstraction, a generic approach is described based on multi-objective criteria, black-box optimisation and randomised search strategies. The interface between problem-specific and generic parts of the exploration framework is made explicit by defining an interface called PISA. This specification and implementation interface and the availability of a wide range of randomised multi-objective search methods makes the proposed framework accessible to a wide range of exploration problems. It resolves the problem that existing optimisation methods can not be coupled easily to the problem-specific part of a design exploration tool.

## 1 Introduction

Embedded systems are usually evaluated according to a large variety of criteria such as performance, cost, flexibility, power and energy consumption, size and weight. As these kinds of non-functional objectives are very often conflicting, there is no single optimal design but a variety of choices that represent different design trade-offs. As a result, a designer is not only interested in one implementation choice but in a well chosen set that best explores these trade-offs.

In addition, embedded systems are often complex in that they consist of heterogeneous subcomponents such as dedicated processing units, application-specific instruction set processors, general purpose computing units, memory structures and communication means like buses or networks. Therefore, the designer is faced with a huge design space.

Embedded systems are resource constrained because of tight cost bounds. Therefore, there is resource sharing on almost all levels of abstraction and resource types that makes it difficult for a designer to assess the quality of a design and the final effect of design choices. This combination of a huge design space on the

---

\*Department Information Technology and Electrical Engineering, Computer Engineering and Networks Lab, ETH Zürich, Switzerland, {kuenzli,thiele,zitzler}@tik.ee.ethz.ch

one hand and the complexity in interactions on the other hand makes automatic or semi-automatic (interactive) methods for exploring different designs important.

Besides the above mentioned multiple objectives in the design of embedded systems, there are tight constraints on the design time. One possibility to accommodate late design changes and a short time to market is to choose a very flexible design, close to a general purpose computing system. On the other hand, this approach sacrifices almost all other quality criteria of a design. As a consequence, embedded systems are usually domain-specific and try to use the characteristics of the particular application domain in order to arrive at competitive implementations. In order to achieve acceptable design times though, there is a need for automatic or semi-automatic (interactive) exploration methods that take into account the application domain, the level of abstraction on which the exploration takes place and that can cope with conflicting criteria.

Following the usual hierarchical approach to embedded system design, there are several layers of abstraction on which design choices must be taken. Above the technology layer one may define the abstraction levels 'logic design and high level synthesis', 'programmable architecture', 'software compilation', 'task level' and 'distributed operation'. These terms are explained in more detail in Section 2. Design space exploration takes place on all of these layers and is a generic tool within the whole design trajectory of embedded systems.

A simplified view on the integration into an abstraction layer is shown in Fig. 1. For example, if the layer of abstraction is the 'programmable architecture', then the generation of a new design point may involve the choice of a cache architecture. The estimation of non-functional properties may be concerned with performance of task execution on the underlying processor architecture, the size of the cache or the total energy consumption. The estimation may either be done using analytic methods or by a suitable simulator by use of suitable input stimuli, e.g. memory access traces. In any case, properties of the sub-components (from logic design) are necessary, e.g. the relations between area, power consumption, structure and size of the cache. The generation of new design points has to satisfy various constraints, e.g. in terms of feasible cache sizes or structures. The choice of a cache will then lead to refined constraints for the design of its sub-components (digital design layer).

Figure 1 makes also apparent the interplay between exploration on the one hand and estimation on the other. The methods and tools applied to the estimation of non-functional properties very much depend on the particular abstraction layer and the design objectives. For example, if the average timing behaviour is of concern, very often approaches based on simulation are used. On the other hand, worst-case timing usually requires analytic methods. Estimation is particularly difficult as only a limited knowledge about the properties of sub-components and the system environment in terms of input stimuli is available. For example on the system-level, the subcomponents to be used are not designed yet and the individual tasks of the application may not be fully specified. This chapter mainly focuses on the generation of new design points and the decision process that finally leads to a

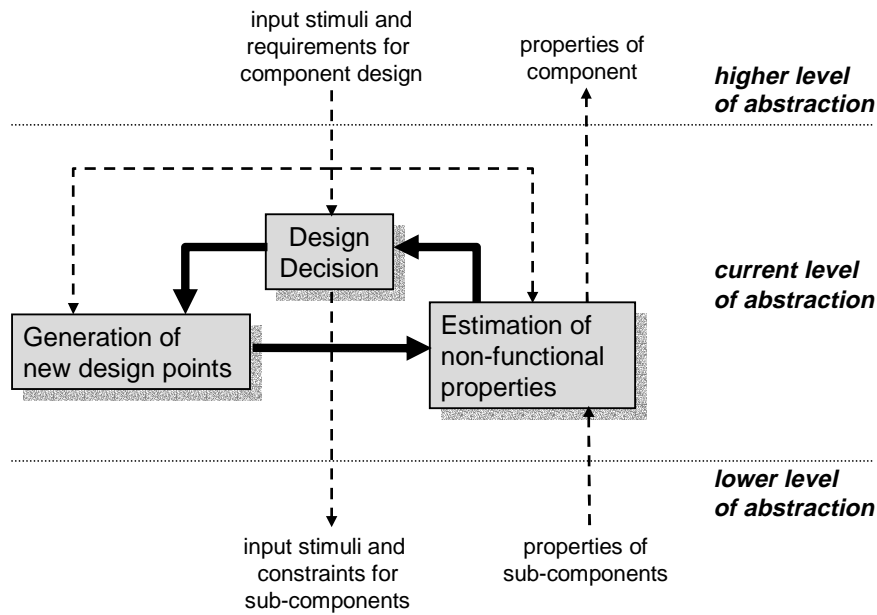


Figure 1: Embedding of exploration in a hierarchical design trajectory for embedded systems.

design decision, estimation will not be covered.

The purpose of the chapter is to review existing approaches to design space exploration of embedded systems and to describe a generic framework that is based on multi-objective decision making, black-box optimisation and randomised search strategies. The framework is based on the PISA (Platform and Programming language independent Interface for Search Algorithms) protocol that specifies a problem-independent interface between the search/selection strategies on the one hand and the domain-specific estimation and variation operators on the other. It resolves the current problem that state-of-the-art exploration and search strategies are not (easily) accessible to solve the domain-specific exploration problems in embedded systems design. The main questions the chapter would like to answer can be phrased as follows: How can one apply efficient design space exploration to a new design problem in embedded system design? How can one integrate a new estimation methodology into a complete design space exploration in a simple and efficient way?

In Section 2 existing approaches to design space exploration in embedded systems are reviewed and classified. Section 4 describes the proposed framework that is based on strategies that select promising design points (Section 4.1), implementation of domain-specific variation operators to determine a suitable neighbourhood of current set of design points (Section 4.2), and the implementation based on the PISA protocol (Section 4.3). A simple running example for cache exploration is

used to illustrate the different steps. Finally, Section 4.4 shortly describes a more complex application, i.e. the system-level exploration of a stream processing architecture.

## 2 Approaches to Design Space Exploration

There is a vast number of approaches available that make use of an automated or semi-automated design space exploration in embedded systems design. Therefore, only a representative subset will be discussed with an emphasis on the exploration strategies, whereas the different estimation methods for non-functional properties will not be discussed further.

As described in Section 1, exploration of implementation alternatives happens at various levels of abstraction in the design. These various layers are described next and existing design space exploration approaches are classified accordingly:

- *Logic Design and High Level Synthesis*: Here one is concerned with the synthesis of digital logic starting from either a register-transfer specification or a more general imperative program. Here, the manual design of dedicated computing units is also included. Typical design choices concern speed vs. implementation area vs. energy consumption, see e.g. [1, 2].
- *Programmable Architecture*: The programmable architecture layer contains all aspects below the instruction set. For example it contains the instruction set definition, the microprocessor architecture in terms of instruction level parallelism, the cache and memory structures. There are numerous examples of exploration on this level of abstraction; they concern different aspects such as caches and memories [3, 4, 5], or the whole processor architecture especially the functional unit selection [6, 7, 8].
- *Software Compilation*: This layer concerns all ingredients of the software development process for a single task such as code synthesis from a model-based design or a high level program specification. Within the corresponding compiler, possible exploration tasks are code size vs. execution speed vs. energy consumption. There are attempts to perform a cross-layer exploration with the underlying processor architecture, see e.g. [9, 10].
- *Task Level*: If the whole application is partitioned into tasks and threads. Therefore, the task level refers to operating system issues like scheduling, memory management and arbitration of shared resources. Therefore, typical trade-offs in choosing the scheduling and arbitration methods are energy consumption vs. average case vs. worst case timing behaviour, e.g. [11].
- *Distributed Operation*: Finally, there exist applications that run on distributed resources. The corresponding layer contains as well the hardware

aspects of distributed operation (such as the design of communication networks) as methods of distributed scheduling and arbitration. On this level of abstraction one is interested in the composition of the whole system that consists of various computing and communication resources. This abstraction layer is sometimes called system-level and the design choices concern the whole system composition out of components as well as the mapping of application to the architecture and the necessary (distributed) scheduling and arbitration methods. This highest level of abstraction seems to be especially suited for exploration methods, see e.g. results on the communication infrastructure [12, 13], on distributed systems [14] or multiprocessor systems and systems-on-chip, e.g. [15, 16, 17, 18, 19].

The above approaches combine several important aspects such as the integration of the exploration into the whole design process, the specific estimation method used to evaluate the properties of design points and finally, and the method that is used to perform the actual exploitation. Following the focus of the chapter, the existing approaches can be classified in a way that is orthogonal to the abstraction layers, namely the methods that are applied to perform the exploration itself. This way it becomes apparent that the exploration process is largely independent of the abstraction level. This property will be used later on in defining the new generic framework.

If only a single objective needs to be taken into account in optimisation, the design points are totally ordered by their objective value. Therefore, there is a single optimal design (if all have different objective values). The situation is different if multiple objectives are involved. In this case, design points are only partially ordered, i.e. there is a set of incomparable, optimal solutions. They reflect the trade-offs in the design. Optimality in this case is usually defined using the concept of Pareto-dominance: A design point dominates another one if it is equal or better in all criteria and strictly better in at least one. In a set of design points, those are called Pareto-optimal which are not dominated by any other.

Using this notion, available approaches to the exploration of design spaces can be characterised as follows:

1. *Exploration by hand*: The selection of design points is done by the designer himself. The major focus is on efficient estimation of the selected designs, e.g. [16].
2. *Exhaustive Search*: All design points in a specified region of the design parameters are evaluated. Very often, this approach is combined with local optimisation in one or several design parameters in order to reduce the size of the design space, see e.g. [4, 20].
3. *Reduction to a Single Objective*: For design space exploration with multiple conflicting criteria, there are several approaches available that reduce the

problem to a set of single criterion problems. To this end, manual or exhaustive sampling is done in one (or several) directions of the search space and a constraint optimisation, e.g. iterative improvement or analytic methods is done in the other, see e.g. [2, 3, 12, 8].

4. *Black-box Randomised Search*: The design space is sampled and searched via a black-box optimisation approach, i.e. new design points are generated based on the information gathered so far and by defining an appropriate neighbourhood function (variation operator). The properties of these new design points are estimated which increases the available information about the design space. Examples of sampling and search strategies used are Pareto Simulated Annealing [21] and Pareto Tabu Search, e.g. [10, 7], evolutionary multi-objective optimisation [14, 18, 13, 22], or Monte Carlo methods improved by statistical estimation of bounds. e.g. [1]. These black box optimisations are often combined with local search methods that optimise certain design parameters or structures, e.g. [11].
5. *Problem-dependent Approaches*: In addition to the above classification, one can find also a close integration of the exploration with a problem-dependent characterisation of the design space. Several possibilities have been investigated so far:
  - Use the parameter independence in order to prune the design space, e.g. [17, 23].
  - Restrict the search to promising regions of design space, e.g. [6].
  - Investigate the structure of the Pareto-optimal set of design points, for example using hierarchical composition of sub-component exploration and filtering [15, 5].
  - Explicitly model the design space, use an appropriate abstraction, derive a formal characterisation by symbolic techniques and use pruning techniques, e.g. [24].

Finally, usually an exhaustive search or a black-box randomised search is carried out for those parts of the optimisation that are inaccessible for tailored techniques.

From the above classification, one can state that most of the above approaches use randomised search techniques one way or the other, at least for the solution of subproblems. This observation does not hold for the exploration by hand or the exhaustive search, but these methods are only feasible for small design spaces with a few choices of the design parameters. Even in case of a reduction to a single objective or in the case of problem-dependent approaches, sub-optimisation tasks need to be solved, either single objective or multi-objective and randomised (black-box) search techniques are applied.

While constructing tools that perform design space exploration of embedded systems at a certain level of abstraction, the question arises, how to apply exploration to a new design problem: How does one connect the problem-specific parts of the exploration with a randomised black-box search engine? What is an appropriate interface between the generic and problem-dependent aspects? Which search strategy should one use? How can one achieve a simple implementation structure that leads to a reliable exploration tool? Section 4 of the chapter is devoted to this problem.

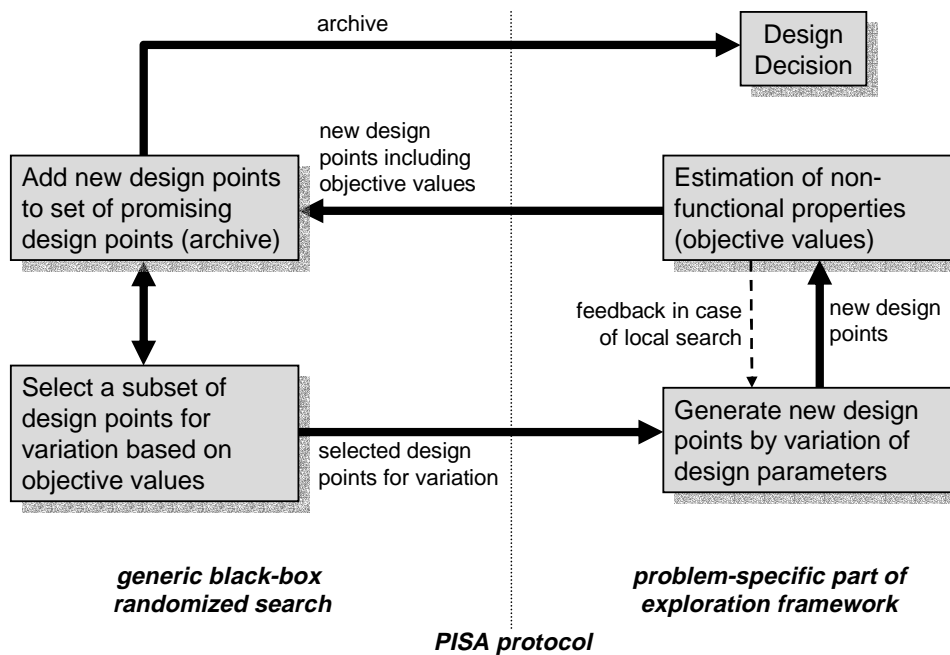


Figure 2: Overview of the proposed framework for design space exploration based on the PISA protocol.

The basis of the proposed solution is the protocol PISA (Platform and Programming language independent Interface for Search Algorithms), see [25]. It is tailored towards black-box randomised search algorithms and is characterised by the following properties: (1) The problem-specific and the generic parts of the exploration method are largely independent from each other, i.e. the generic search and selection should be treated as a black-box (separation of concerns). (2) The framework itself should not depend on the machine types, operating systems or programming languages used (portability). (3) The protocol and framework should be tailored towards a reliable exploration. The main components of the proposed framework in Fig. 2 are a refinement of Fig. 1. It shows the separation into the problem-specific variation and estimation part on the one hand and generic black-box search on the other.

### 3 A Simple Example: Design Space Exploration of Cache Architectures

Before the PISA-framework is described in more detail (in Sec. 4.3), this section introduces a simple example application that will be used throughout the remainder of this chapter for illustration purposes. Note, that it is not the purpose of the example to present any new results in cache optimisation.

The example problem to solve is to optimise the architecture of a cache for a predefined benchmark application. The solution space for the problem is restricted to L1 data caches only, i.e. the design choices include the cache size, the associativity level, the block size, and the replacement strategy. The goal is to identify a cache architecture that (i) maximises the overall computing performance with respect to the benchmark under consideration and (ii) minimises the chip area needed to implement the cache in silicon.

Nr.	Parameter	Range
1	# of Cache lines	$2^k$ , with $k = 6 \dots 14$
2	Block size	$2^k$ Bytes, with $k = 3 \dots 7$
3	Associativity	$2^k$ , with $k = 0 \dots 5$
4	Replacement strategy	LRU or FIFO

Table 1: Parameters determining a cache architecture

In Table 1, all parameters and possible values for the cache architecture are given. A design point is therefore determined by three integer values and a Boolean value. The integers denote the number of cache lines, the cache block size and the cache associativity; the Boolean value encodes the replacement strategy: *false* denotes FIFO (first-in-first-out), *true* denotes LRU (least recently used). Figure 3 graphically depicts the design parameters. The values for the number of cache lines, block size and associativity have to be powers of 2, due to restrictions in the tools used for evaluation of the caches.

The first objective according to which the cache parameters are to be optimised is the CPI (cycles per instruction) achieved for a sample benchmark application and the second objective is the chip area needed to implement the cache on silicon. To estimate the corresponding objective values, two tools were used, namely `sim-outorder` of SimpleScalar [26] and CACTI [27] provided by Compaq. The first tool served to estimate the CPI for the benchmark `compress95` running on the plain text version of the GNU public license as application workload. The smaller the CPI for `compress95` for a particular solution, the better is this solution for this objective. The second tool calculated an estimate for the silicon area needed to implement the cache. The smaller the area, the better is the cache for the area objective.

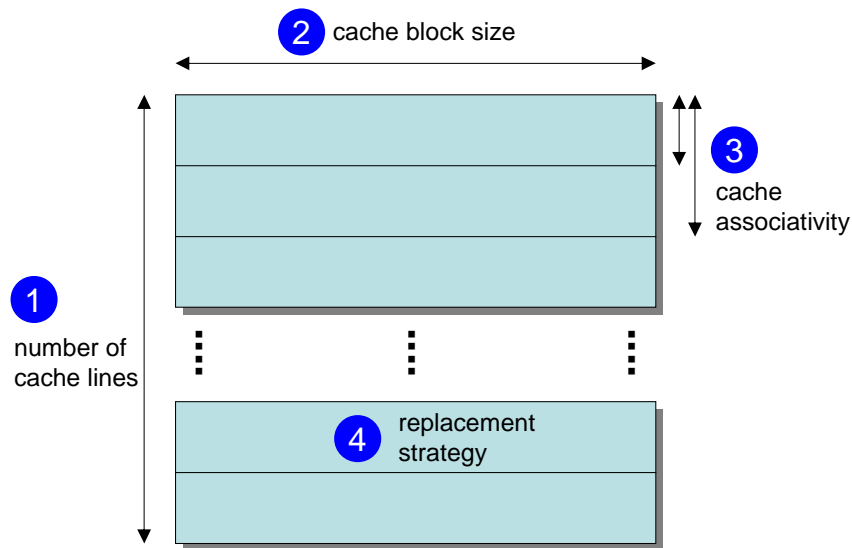


Figure 3: Illustration of the considered design choices for an L1 data cache architecture.

## 4 A General Framework for Design Space Exploration

As discussed in Section 2, the proposed general framework for design space exploration separates application-specific aspects from the optimisation strategy. The resulting two parts are implemented as independent processes communicating via text files, as will be detailed in Section 4.3. This concept (cf. Fig. 2) reflects the working principle of black-box randomised search algorithms.

Black-box methods are characterised by the fact that they do not make any assumptions about the objective functions, and in this sense they treat the design criteria as black boxes which can contain arbitrarily complex functionalities. Initially, they create one or several designs at random, which are then evaluated with respect to the objective functions under consideration. Afterwards, the information about the already considered design(s) is used in order to generate one or several different designs that are then evaluated as well. This process is repeated until a certain number of iterations has been carried out or another stopping condition is fulfilled. The goal here is to exploit structural properties of the design space such that only a fraction of the design space needs to be sampled to identify optimal resp. nearly optimal solutions. This implies that different search space characteristics require different search strategies, and accordingly various black-box optimisers such as randomised local search, simulated annealing, evolutionary algorithms, etc. and variants thereof are available, see e.g. [28].

Two principles form the basis for all randomised search algorithms: selection and variation. On the one hand, selection aims at focusing the search on promising

regions of the search space as will be discussed in Section 4.1. This part is usually problem independent. On the other hand, variation means generating new designs by slightly modifying or combining previously generated ones. Although standard variation schemes exist—details can be found in Section 4.2—the generation of new designs based on existing ones is strongly application dependent, similarly to the internal representation and the evaluation of designs.

## 4.1 Selection

The selection module implements two distinct phases: selection for variation and selection for survival. The former type of selection chooses the most promising designs from the set of previously generated designs that will be varied in order to create new designs. For practical reasons, though, not all of the generated designs will be kept in memory. While, e.g., simulated annealing and tabu search only store one solution in the working memory (in this case, selection for variation simply returns the single, stored solution), evolutionary algorithms operate on a population of solutions, which is usually of fixed size. As a consequence, another selection phase is necessary in order to decide which of the currently stored designs and the newly created ones will remain in the working memory. This phase is often called selection for survival or environmental selection, in analogy to the biological terminology used in the context of evolutionary algorithms.

### 4.1.1 Selection for Variation

Selection for variation is usually implemented in a randomised fashion. One possibility to choose  $N$  out of  $M$  designs is to hold tournaments between two solutions that are picked at random from the working memory based on a uniform probability distribution. For each tournament, the better design is copied to a temporary set which is also denoted as mating pool—again a term mainly used within the field of evolutionary computation. By repeating this procedure, several designs can be selected for variation, where high-quality designs are more likely to have one or multiple copies in the mating pool. This selection method is known as binary tournament selection; many alternative schemes exist as well (see [29]).

Most of these selection algorithms assume that the usefulness or quality of a solution is represented by a scalar value, the so-called fitness value. While fitness assignment is straight forward in case of a single objective function, the situation is more complex in a multiobjective scenario. Here, one can distinguish between three conceptually different approaches:

- *Aggregation*: Traditionally, several optimisation criteria are aggregated into a single objective by, e.g., summing up the distinct objective function values, where weight coefficients are used to control the influence of each criterion. The difficulty with this approach, though, is the appropriate setting of the weights. This usually requires more knowledge about the design space than

is actually available. Furthermore, optimising a particular weight combination yields one Pareto-optimal solution. To obtain several optimal trade-off designs, multiple weight combinations need to be explored either in parallel or subsequently. Nevertheless, not necessarily all Pareto-optimal designs can be found as illustrated in Fig. 4. The weighted-sum approach is only able to detect all solutions if the front of Pareto-optimal solutions is convex. Similar problems occur with many other aggregation methods, see [30].

- *Objective Switching:* The first papers using evolutionary algorithms to approximate the Pareto set suggested to switch between the different objectives during the selection step. For instance, Schaffer [31] divided selection for variation into  $n$  selection steps where  $n$  corresponds to the number of optimisation criteria; in the  $i$ th step, designs in the working memory were chosen according to their  $i$ th objective function value.
- *Dominance-based Ranking:* Nowadays, most popular schemes use fitness assignments that directly make use of the dominance relation or extensions of it. By pairwise comparing all the designs in the working memory, different types of information can be extracted. The dominance rank gives the number of solutions by which a specific solution is dominated, the dominance count represents the number of designs that a particular design dominates, and the dominance depth denotes the level of dominance when the set of designs is divided into non-overlapping nondominated fronts (see [28] for details).

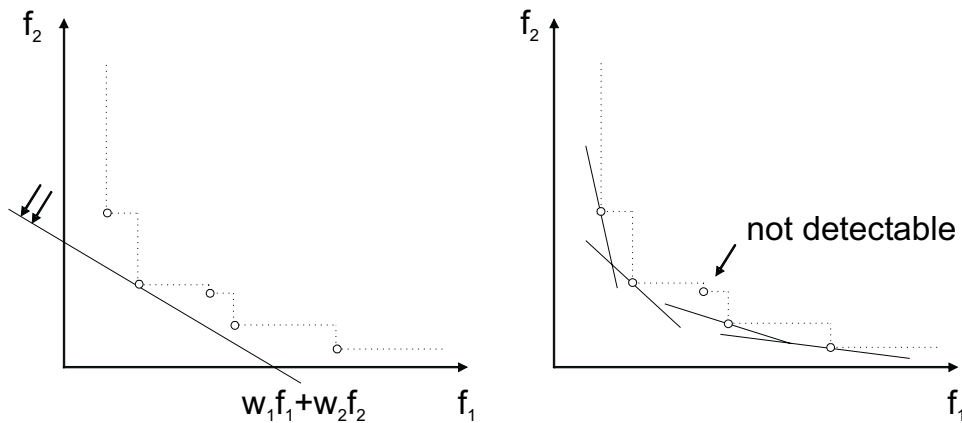


Figure 4: Illustration of the weighted-sum approach for two objectives. The left hand side shows how a particular weight combination  $(w_1, w_2)$  uniquely identifies one Pareto-optimal design. The right hand side demonstrates that not for all Pareto-optimal designs such a weight combination exists.

These fitness assignment schemes can also be extended to handle design constraints. For dominance-based approaches, the dominance relation can be modified

such that feasible solutions by definition dominate infeasible ones, while among infeasible designs the one with the lower constraint violation is superior—for feasible solutions, the definition of dominance remains unchanged (cf. [28]). An alternative is the penalty approach which can be used with all of the above schemes. Here, the overall constraint violation is calculated and summarised by a real value. This value is then added to the original fitness value (assuming that fitness is to be minimised); thereby, infeasible are penalised.

Finally, another issue that is especially important in the presence of multiple objectives is maintaining diversity among the designs stored. If the goal is to identify a set of Pareto optima, special techniques are necessary in order to prevent the search algorithm from converging to a single trade-off solution. Most modern multiobjective optimisers integrate some diversity preservation technique that estimates the density of solutions in the space defined by the objective functions. For instance, the density around a solution can be estimated by calculating the Euclidean distance to the next closest solution. This density information can then be incorporated into the fitness, e.g., by adding original fitness value and density estimate. Again, there is variety of different methods that cannot be discussed here in detail.

#### **4.1.2 Selection for Survival**

When approximating the Pareto set, it is desirable not to lose promising designs due to random effects. Therefore, selection for survival is usually realized by a deterministic algorithm. Similar issues as with selection for variation come into play here; however, almost all search methods make sure that designs not dominated among those in the working memory are preferred over dominated ones with respect to environmental selection. If there are too many nondominated solutions, then additional diversity information is used to further discriminate between these designs. Furthermore, as many randomised search algorithms only keep a single solution in the working memory, often a secondary memory, a so-called archive (see also Fig. 2), is maintained that stores the current approximation of the Pareto set. For instance, PAES [32], a randomised local search method for multiobjective optimisation, checks for every generated design whether it should be added to the archive, i.e., whether it is dominated by any other archive member. If the design was inserted, dominated designs are removed. If the archive size is exceeded after insertion, a design with the highest density estimate is deleted.

A theoretical issue that has been investigated recently by different researchers [33, 34] addresses the loss in quality per iteration. Optimally, the current set of designs represents the best Pareto set approximation among all solutions ever considered during the optimisation run—given the actual memory constraints. This goal is difficult to achieve in general, but Laumanns et al. [33] proposed an archiving method by which the loss can be bound and kept arbitrarily small by adjusting the memory usage accordingly.

### 4.1.3 Multiobjective Optimisers

The above discussion could only touch the aspects involved in the design of the selection process for a multi-objective randomised search algorithm. In fact, a variety of methods exist, cf. [28], which over the time have become more complex, see e.g. [35]. In the evolutionary computation field, even a rapidly growing sub-discipline emerged focusing on the design of evolutionary algorithms for multiple criteria optimisation [36]. However, an application engineer who would like to carry out a design space exploration is not necessarily an expert in the optimisation field. He is rather interested in using state-of-art multi-objective optimisers. For this reason, the proposed design space exploration framework separates the general search strategy from the application-specific aspects such as variation. Thereby, it is possible to use precompiled search engines without any implementation effort.

## 4.2 Variation

In this subsection the application-specific part of the proposed design space exploration framework is described. In particular, the variation module encapsulates the representation of a design point and the variation operators, see also the overview in Fig. 2. It is the purpose of this component in the design space exploration framework to generate suitable new design points from a given set of selected ones. Therefore, the variation is problem-specific to a large extent and provides a major opportunity to including domain-knowledge.

### 4.2.1 Representation

A formal description of a design needs to be appropriately encoded in the optimisation algorithm. The main objectives for suitable design representations are as follows:

- The encoding should be designed in a way that enables an efficient generation of design points in an appropriate neighbourhood, see also the next subsection on variation operators.
- The representation should be able to encode all relevant design points of the design space. In particular, if the design space has been pruned using problem-dependent approaches, the chosen representation should reflect these constraints in a way that enables efficient variation for the determination of a neighbourhood.
- The design parameters should be independent of each other as much as possible in order to enable a suitable definition of variation operators.

A representation of a solution can, e.g., consist of real or integer values, or vectors thereof to encode clock speeds, memory size, cache size, etc. Bit vectors can be used to describe the allocation of different resources. Another class of

representations could be the permutation of a vector with fix elements to represent, e.g., a certain task scheduling. Furthermore, variable length data structures such as trees or lists can be used for the representation of, e.g., graphs (see [29] for an overview).

All parameters for the representation have to lie inside the problem specification that spans the design space of possible solutions. A solution parameter could therefore be, e.g., a real value in the range given in the specification, an integer value in a list of possible integers, or a selected edge in a problem specification graph.

The cache example uses integer values to represent the number of cache lines in the solution cache, the block size and the associativity. The integer values are the actual cache parameters, such that these lie in the range specified in Table 1. The cache line replacement strategy is represented by a Boolean value.

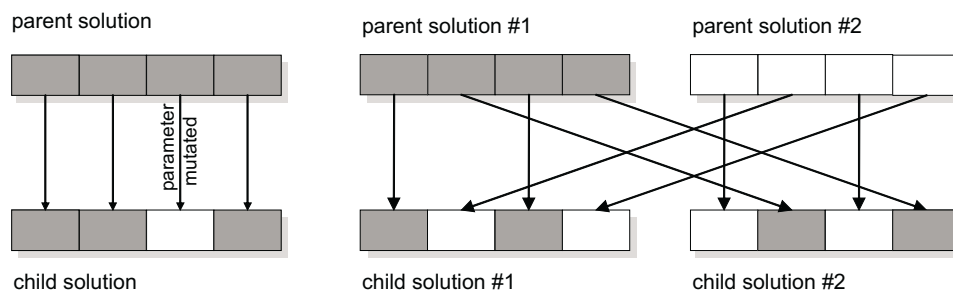


Figure 5: Variation operators used in the cache example: mutation (left), recombination (right).

#### 4.2.2 Variation Operators

The purpose of the variation operators is to determine new design points given a set of selected previously evaluated design points. There are several objectives for selecting appropriate variation operators:

- The variation operators operate on the design representation and generate a local neighbourhood of the selected design points. These new design points will be evaluated by the estimation, see Fig. 1. Therefore, the construction of the variation operators is problem-dependent and a major possibility to include domain-knowledge.
- The constructed neighbourhood should not contain infeasible design points, if possible.
- In case of design points that are infeasible because non-functional properties are outside of given constraints, one may use a feedback loop shown in Fig. 1 in order to correct.

- The variation operator may also involve problem-dependent local search (e.g. by optimising certain parameters or hidden optimisation criteria) in order to relieve the randomised search from optimisation tasks that can better be handled with domain-knowledge.

In principle, different variation operators can be distinguished according to the number of solutions they operate on. Most randomised search algorithms generate a single new design point by applying a randomised operator to a known design point. For simulated annealing and randomised local search algorithms this operator is called neighbourhood function, whereas for evolutionary algorithms this operator is denoted as mutation operator. The term mutation will be used in the remainder of this section.

In the context of evolutionary algorithms there also exists a second type of variation, in addition to mutation. Since evolutionary algorithms maintain a population of solutions, it is possible to generate one or more new solutions based on two or more existing solutions. The existing designs selected for variation are often referred to as parents, whereas the newly generated designs are called children. The operator that generates  $\geq 1$  children based on  $\geq 2$  parents is denoted as recombination.

**Mutation:** The assumption behind mutation is that it is likely to find better solutions in the neighbourhood of good solutions. Therefore, mutation operators are usually designed in such a way that the probability of generating a specific solution decreases with increasing distance from the parent. There exist several approaches to implement mutation. It is, e.g., possible to always change exactly one parameter in the representation of a solution and keep all other parameters unchanged. A different mutation operator changes each of  $n$  parameters with probability  $1/n$ , which leads to the fact that one parameter is changed in expectation. This approach is also used in the cache example.

Changing a parameter means changing its value, i.e., flipping a bit in a binary representation, or choosing new parameter values according to some probability distribution for an integer- or real-valued representation. For representations based on permutations of vector elements the mutation operator changes the permutation by exchanging two elements. If the specification is based on lists of possible values, the mutation operator selects a new element according to some probability distribution.

In general, a mutation operator should on the one hand produce a new solution that is “close” to the parent solution with a high probability, but on the other hand be able to produce any solution in the design space, although with very small probability. This is to prevent the algorithm from being stuck in a local optimum.

The cache example uses the following mutation operator: Each of the design parameters is mutated with probability 0.25 (as there are 4 different parameters). The change that is applied to each of the parameters is normally distributed, i.e., the value of a parameter is increased by a value that is normally distributed around 0

inside the ranges given in Table 1; e.g. the block size parameter change is normally distributed between -4 and +4. Note, that in the example changes of size 0 are also allowed, i.e. the parameter remains unchanged.

**Recombination:** Recombination takes two or more solutions as input, and then generates new solutions that represent combinations of the parents. The idea behind recombination is to take advantage of the good properties of each of the parent to produce even better children. In analogy to the mutation operator, a good recombination vector should produce solutions that lie “between” the parents either with respect to the parameter space or to the objective space.

For vectors in general, recombination of two parents can be accomplished by cutting both solutions at randomly chosen positions and rearranging the resulting pieces. For instance, one-point crossover creates a child by copying the first half from the first parent and the second half from the second parent. If the cut is made at every position, i.e., at each position randomly either the value from the first or the second parent is copied, the operator is called uniform recombination.

A further approach for real-valued parameters is to use the average of the two parents’ parameter values, or some value between the parents’ parameter values. A detailed overview of various recombination operators for different representation data structures can be found in [37].

For the cache example uniform recombination was used, i.e., for each of the parameters like cache block size it was randomly decided from which parent solution the parameter for the first child solution should be used, where all unused parameters of the parent solutions are then used for the second child solution. See Fig. 5 on the right hand side for a graphical representation of uniform recombination.

**Infeasible Solutions:** It can happen that after mutation or recombination a generated solution is not feasible, i.e., the solution represented by the parameters doesn’t describe a valid system. To solve this problem there are different possibilities. First, one could ensure that the variation operators do not create infeasible solutions by controlling the construction of new solutions, one can call this approach “valid by construction”. Second, one could implement a repair method, that turns constructed solutions that are infeasible into feasible ones by fixing the infeasible parameters. The third possibility is to introduce an additional constraint and to penalise infeasible designs in the way as described in Section 4.1. Finally, one can use the concept of penalty functions in order to guide the search away from areas with infeasible design points.

### 4.3 Implementation Issues

In this section the protocol used in PISA is briefly introduced, see also [25]. It is the purpose of PISA to make state-of-the-art randomised search algorithms for multi-objective optimisation problems readily available. Therefore, for a new design space exploration task in embedded system design, one can concentrate

on the problem-dependent aspects, where the domain-knowledge comes in. The protocol has to be implemented by any design space exploration tool that would like to benefit from pre-compiled and ready-to-use search algorithms available at <http://www.tik.ee.ethz.ch/pisa>. Besides, the web-site also contains a set of application problems and benchmark applications for the development of new randomised search algorithms. The detailed protocol including file formats and data type definitions is given in [25]. In the protocol description, the application-specific part is called 'variator' and the search algorithm is denoted 'selector', according to Figure 6. The variator also contains the estimation of non-functional properties.

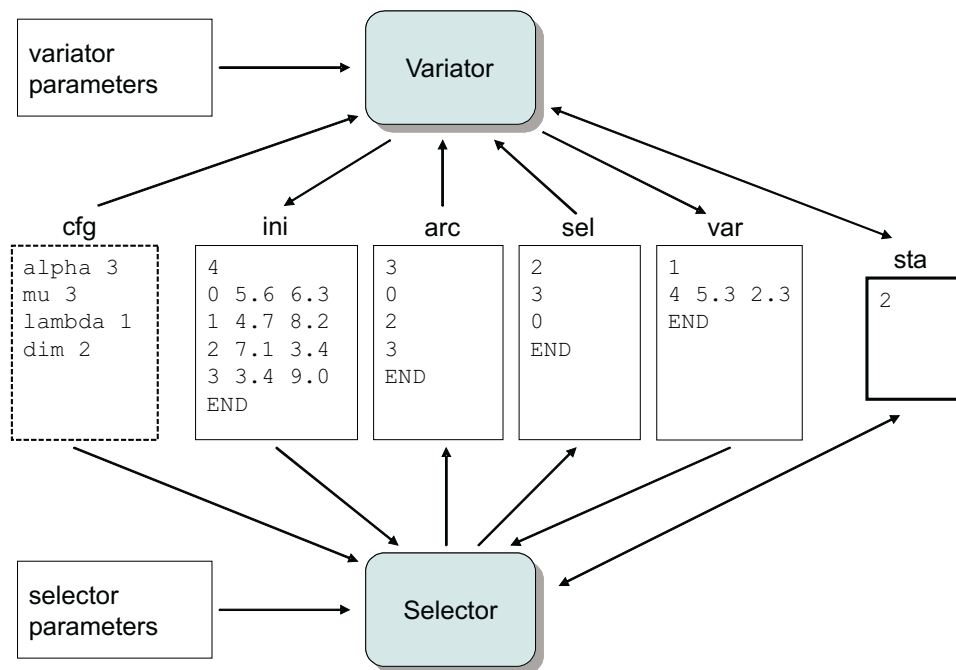


Figure 6: Communication between modules through text files as defined by the PISA protocol. The files contain sample data.

The details of the protocol have been designed with several objectives in mind:

- Small amount of data that need to be communicated between the two different processes (selector and variator).
- The communicated data should be independent of the problem domain in order to enable a generic implementation of the selector process.
- Separation into problem-independent (selector) and problem-dependent (variator) processes.

- The implementation of the protocol should be as much as possible independent of the programming languages, hardware platforms and operating systems. It should enable a reliable (delay-independent) execution of the design space exploration.

The protocol defines the sequence of actions performed by the selector and variator processes. The communication between the two processes is done by exchange of text files over a common file system. The handshake protocol is based on states and ensures that during the optimisation process only one module is active at any time. During the inactive period a process polls the state file for changes. Whenever a module reads a state that requires some action on its part, the operations are performed and the next state is set.

The core of the optimisation process consists of state 2 and state 3: In each iteration the selector chooses a set of parent individuals and passes them to the variator. The variator generates new child solutions on the basis of the parents, computes the objective function values of the new individuals, and passes them back to the selector.

In addition to the core states two more states are necessary for normal operation. State 0 and state 1 trigger the initialisation of the variator and the selector, respectively. In state 0 the variator reads the necessary parameters. Then, the variator creates an initial population, determines the objective values of the individuals and passes the initial population to the selector. In state 1, the selector also reads the required parameters, then selects a sample of parent individuals and passes them to the variator.

The four states 0–3 provide the basic functionality of the PISA-protocol. To add some flexibility the PISA-protocol defines a few more states which are mainly used to terminate or reset both the variator process and the selector process. Table 2 gives an overview over all defined states. The additional states 4–11 are not mandatory for a basic implementation of the protocol.

State	Action	Next State
State 0	Variator reads parameters and creates initial solutions	State 1
State 1	Selector reads parameters and selects parent solutions	State 2
<b>State 2</b>	<b>Variator generates and evaluates new solutions</b>	<b>State 3</b>
<b>State 3</b>	<b>Selector selects solutions for variation</b>	<b>State 2</b>
State 4	Variator terminates.	State 5
State 6	Selector terminates.	State 7
State 8	Variator resets. (Getting ready to start in state 0)	State 9
State 10	Selector resets. (Getting ready to start in state 0)	State 11

Table 2: States for the PISA-protocol. The main states of the protocol are printed in bold face.

The data transfer between the two modules introduces some overhead com-

pared to a traditional monolithic implementation. Thus, the amount of data exchange for each individual should be minimised. Since all representation-specific operators are located in the variator, the selector does not have to know the representation of the individuals. Therefore, it is sufficient to convey only the following data to the selector for each individual: an identifier and its objective vector. In return, the selector only needs to communicate the identifiers of the parent individuals to the variator. The proposed scheme allows to restrict the amount of data exchange between the two modules to a minimum.

For PISA-compliant search algorithms to work correctly, a designer has to ensure, that all objectives are to be *minimised*. In addition the variator and selector have to agree on a few common parameters: (i) the population size  $\alpha$ , (ii) the number of parent solutions  $\mu$ , (iii) the number of child solutions  $\lambda$  and (iv) the number of objectives  $d.im$ . These parameters are specified in the parameter file with suffix *c.f.g.*, an example file is shown in Figure 6.

The selector and the variator are normally implemented as two separate processes. These two processes can be located on different machines with possibly different operating systems. This complicates the implementation of a synchronisation method. Most common methods for interprocess communication are therefore not applicable.

In PISA, the synchronisation problem is solved using a common state variable which both modules can read and write. The two processes regularly read this state variable and perform the corresponding actions. If no action is required in a certain state, the respective process sleeps for a specified amount of time and then rereads the state variable. The state variable is an integer number stored to a text file with suffix *st.a.* The protocol uses text files instead of, e.g., sockets, because file access is completely portable between different platforms and familiar to all programmers.

All other data transfers between the two processes besides the state are also performed using text files. The initial population is written by the variator to the file with suffix *ini*, the population is written by the selector to a file with suffix *arc*. In a text file with suffix *sel* the selector stores the parent solutions that are selected for variation. The newly generated solutions are passed from the variator to the selector through a file with suffix *var*. All text files for data transfer have to begin with the number of elements that follow and to end with the keyword *END*.

Once the receiving process has completely read a text file, it has to overwrite the file with 0, to indicate that it successfully read the data.

#### **4.4 Application of PISA for Design Space Exploration**

For the cache example presented in Section 3, the variator part was written in Java. The mutation and recombination operator were implemented as described in 4.2, and the combination with a selector is PISA-compliant as described in Section 4.3. The selector was downloaded from the PISA website. The design space exploration for L1 data caches was performed using SPEA2, an evolutionary multiobjective

optimiser described in [35].

The solutions selected by SPEA2 for variation were pairwise recombined with probability 0.8, and the resulting solutions were then mutated with probability 0.8. Afterwards the generated solutions were added to the population and passed to the search algorithm for selection.

The design space with all solutions is shown in Fig. 7. These design points have been generated using exhaustive search in order to compare the heuristic search with the Pareto front of optimal solutions. The front of non-dominated solutions found for the cache example with SPEA2 after a typical optimisation run with 40 generation for a population size of 6 solutions is marked with circles. The details of the solutions in the population after 40 generations are represented in Table 3.

No.	CPI	Area	Design Parameters			
1	0.5774	0.001311	LRU	$2^7$ cache lines	block size 8	d. m.
2	0.5743	0.001362	LRU	$2^7$ cache lines	block size 8	2 sets
3	0.5622	0.022509	FIFO	$2^8$ cache lines	block size 64	8 sets
4	0.5725	0.002344	LRU	$2^7$ cache lines	block size 16	2 sets
5	0.5488	0.024018	LRU	$2^{10}$ cache lines	block size 32	8 sets
6	0.5319	0.027122	LRU	$2^{10}$ cache lines	block size 32	16 sets
7	0.5666	0.002898	LRU	$2^6$ cache lines	block size 32	2 sets
8	0.5653	0.003629	FIFO	$2^6$ cache lines	block size 64	d. m.
9	0.5307	0.044902	FIFO	$2^{10}$ cache lines	block size 64	8 sets
10	0.5626	0.004907	LRU	$2^6$ cache lines	block size 64	2 sets

Table 3: Details of 10 non-dominated solutions for the simple example of a cache exploration found after a typical design space exploration run. These solutions are marked with circles in Fig. 7.

Although the cache design space exploration problem is simple in nature, one can make some observations which also hold for more involved exploration problems. The two objectives, namely the minimisation of the silicon area and the minimisation of the CPI, are conflicting, resulting in an area vs. performance trade-off. This results in the fact that there is not a single optimal solution, but a front of Pareto-optimal solutions. All points on this front represent different promising designs, leaving the final choice for the design of the cache up to the designer's preference. Further, one can observe in Fig. 7 that the evolutionary algorithm found solutions close to the Pareto-optimal front.

The reduction of the problem to a single-objective optimisation problem, e.g. using a weighted-sum approach is difficult already for this simple example, because it represents a true multi-objective problem. It is not at all clear how to relate area to performance, which would be needed for the weighted-sum approach.

As a more involved example, the design space exploration of complex stream processor architectures on the system level has been performed using the PISA framework. To this end, a variator process 'EXPO' has been implemented which

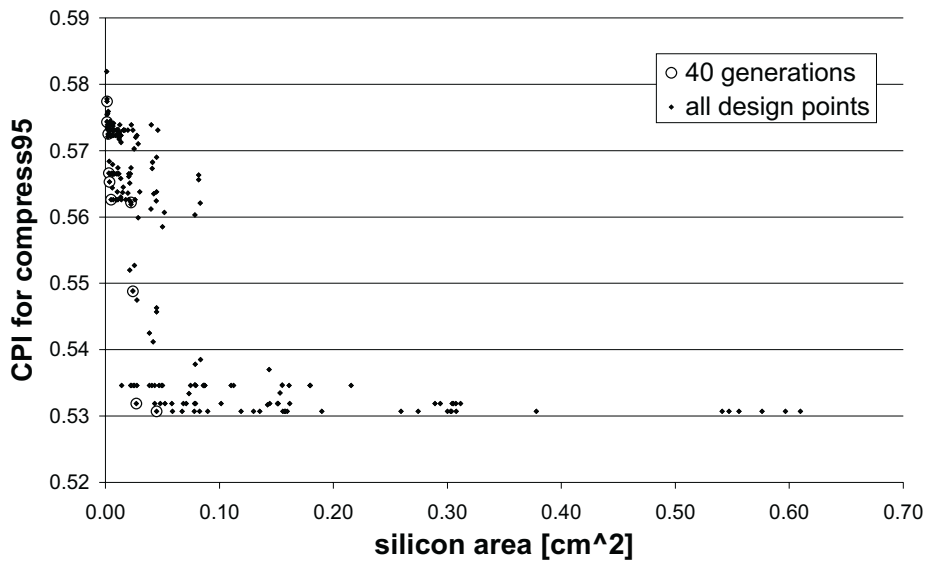


Figure 7: All 540 possible design points determined using exhaustive search and the design points found by the multi-objective search algorithm SPEA2 after 40 generations.

is available on the web-site of PISA also. The representation of design points, the variation operators, the local search method to reduce the design space and the way how infeasible design points are avoided or repaired are all specific to the application domain of stream processors. These methods are based on models for stream processing tasks, a specification of the workload generated by traffic flows, and a description of the feasible space of architectures involving computation and communication resources.

For the evaluation of a single design point, the tool makes use of a new method to estimate end-to-end packet delays and queueing memory, taking task scheduling policies and bus arbitration schemes into account. The method is analytical and is based on a high level of abstraction, where the goal is to quickly identify interesting architectures, which may then be subjected to a more detailed evaluation, e.g. using simulation. The approach used in EXPO and results are described in much more detail in [22].

For the simple cache example the design space could have been explored using exhaustive search instead of employing evolutionary algorithms, which actually was done to determine all solutions shown in Fig. 7. For larger design spaces, as the one explored with EXPO, exhaustive search is prohibitive, and only randomised search algorithms can be used. It has been shown in many studies (e.g. in [28]) that evolutionary algorithms perform better on multi-objective optimisation problems than other simpler randomised search algorithms.

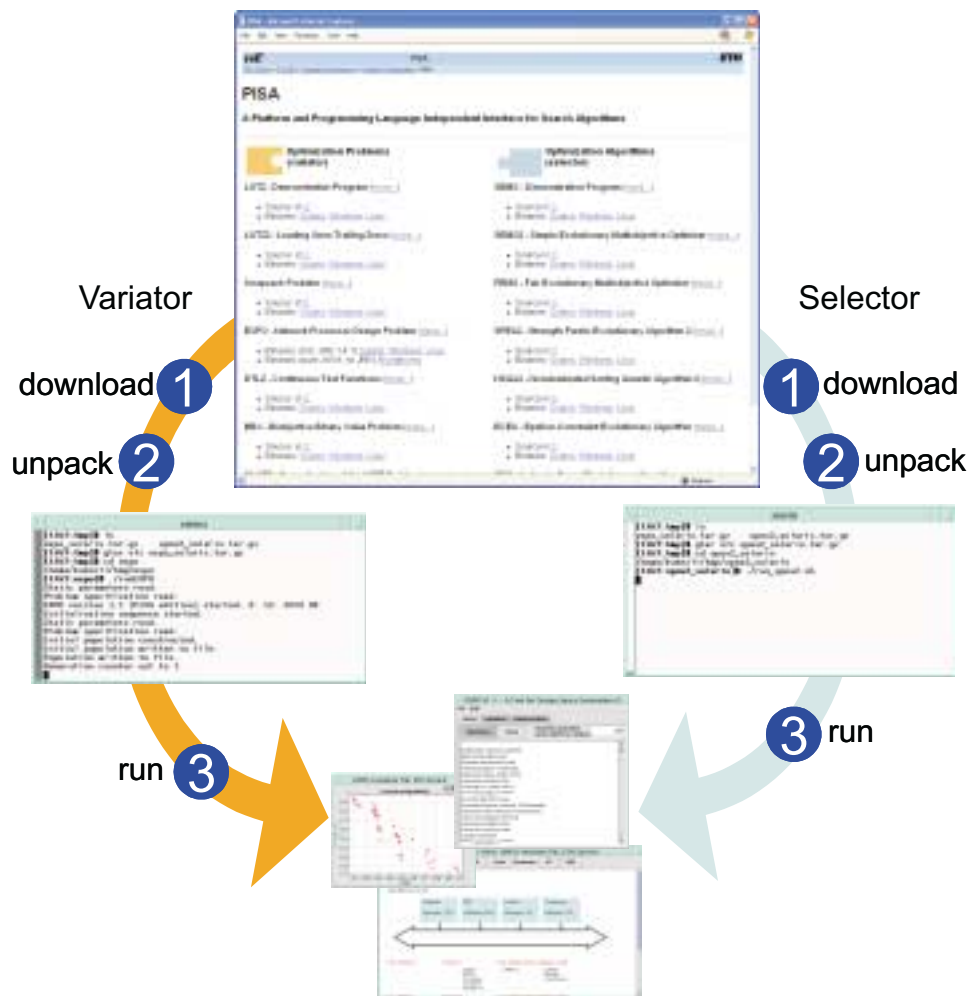


Figure 8: On the PISA website, many different ready-to-use evolutionary search algorithms can be downloaded. Additionally, a design space exploration tool is offered for download at the website. The only steps needed for a first design space exploration using the PISA framework are shown: (1) just download a variator, e.g., the exploration tool EXPO and one of the search algorithms on the right hand side of the website, (2) unpack the tools, and then (3) run them.

PISA enables the use of different evolutionary algorithms without having to change the implementation of the exploration tools. A recent study [38] has shown that for EXPO the quality of the approximation of the Pareto-optimal front may differ between different evolutionary algorithms. With a modular framework based on the PISA protocol it is possible to test the design space exploration performance of different randomised search algorithms to find the search algorithm most suitable to the exploration problem.

## 5 Conclusion

This chapter introduced a framework for design space exploration of embedded systems. It is characterised by (1) multiple optimisation criteria, (2) randomised search algorithms and (3) a software interface that clearly separates problem-dependent and problem-independent parts of an implementation. In particular, the interface PISA formally characterises this separation. It is implemented in a way that is independent of programming language used and the underlying operating system. As a result, it is easily possible to extend any existing method to estimate non-functional properties with an effective multi-objective search.

It should be pointed out, that effective automatic or semi-automatic (interactive) exploration needs deep knowledge about the specific optimisation target, i.e. the level of abstraction, the optimisation goals, efficient and accurate estimation methods. Nevertheless, the PISA framework separates the problem-dependent variation and estimation from the generic search and selection. Therefore, the user is relieved from dealing with the complex and critical selection mechanisms in multi-objective optimisation. On the other hand, his specific domain knowledge will be important when designing the variation operators that determine a promising local neighbourhood of a given search point.

Finally, it is common knowledge that the class of randomised search algorithms described in the chapter does not guarantee to find the optimal solutions. In addition, if there is domain knowledge available that allows problem-specific exploration methods to be applied, then there is little reason to use a generic approach. But usually, those analytic methods do not exist for complex optimisation scenarios as found in embedded system design.

### Acknowledgement

The work described in this paper was supported by the Swiss Government within the KTI program. The project was part of the MEDEA+ project SpeAC.

### References

- [1] D. Bruni, A. Bogliolo, and L. Benini, "Statistical design space exploration for application-specific unit synthesis," in *Proceedings of the 38th Design Automation Conference*, pp. 641–646, ACM Press, 2001.
- [2] C. Chantrapornchai, E.-M. Sha, and X. S. Hu, "Efficient acceptable design exploration based on module utility selection," *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, vol. 19, no. 1, pp. 19–29, 2000.
- [3] A. Ghosh and T. Givargis, "Analytical design space exploration of caches for embedded systems," in *Design, Automation and Test in Europe Conference and Exhibition (DATE 03)*, p. 10650, IEEE Press, March 2003.

- [4] W.-T. Shiue and C. Chakrabarti, "Memory exploration for low power, embedded systems," in *Proceedings of the 36th ACM/IEEE Design Automation Conference*, pp. 140–145, ACM Press, 1999.
- [5] R. Szymanek, F. Catthoor, and K. Kuchcinski, "Time-energy design space exploration for multi-layer memory architectures," in *Proc. of 7th ACM/IEEE Design, Automation and Test in Europe Conference*, p. 10318, ACM Press, 2004.
- [6] G. Hekstra, G. L. Hei, P. Bingley, and F. Sijstermans, "TriMedia CPU64 Design Space Exploration," in *1999 IEEE International Conference on Computer Design*, pp. 593–598, October 1999.
- [7] G. Palermo, C. Silvano, and V. Zaccaria, "A flexible framework for fast multi-objective design space exploration of embedded systems," in *PATMOS 2003-International Workshop on Power and Timing Modeling*, vol. 2799 of *Lecture Notes in Computer Science*, (Torino, Italy), pp. 249–258, Springer, 2003.
- [8] S. Rajagopal, J. Cavallaro, and S. Rixner, "Design space exploration for real-time embedded stream processors," *IEEE Micro*, August 2004. Accepted for publication.
- [9] E. Zitzler, J. Teich, and S. Bhattacharyya, "Evolutionary algorithms for the synthesis of embedded software," *IEEE Transactions on VLSI Systems*, vol. 8, pp. 452–456, April 2000.
- [10] G. Agosta, G. Palermo, and C. Silvano, "Multi-objective co-exploration of source code transformations and design space architectures for low-power embedded systems," in *Proceedings of the 2004 ACM symposium on Applied computing*, pp. 891–896, ACM Press, 2004.
- [11] N. K. Bambha, S. Bhattacharyya, J. Teich, and E. Zitzler, "Hybrid search strategies for dynamic voltage scaling in embedded multiprocessors," in *Proceedings of the Ninth International Symposium on Hardware/Software Code-sign*, (Copenhagen, Denmark), pp. 243–248, April 2001.
- [12] K. Lahiri, A. Raghunathan, and S. Dey, "Design space exploration for optimizing on-chip communication architectures," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 23, pp. 952–961, June 2004.
- [13] M. Eisenring, L. Thiele, and E. Zitzler, "Handling conflicting criteria in embedded system design," *IEEE Design and Test of Computers*, vol. 17, pp. 51–59, April 2000.
- [14] U. Anliker, J. Beutel, M. Dyer, R. Enzler, P. Lukowicz, L. Thiele, and G. Troester, "A systematic approach to the design of distributed wearable

- systems,” *IEEE Transactions on Computers*, vol. 53, no. 8, pp. 1017–1033, 2004.
- [15] S. Abraham, B. R. Rau, and R. Schreiber, “Fast design space exploration through validity and quality filtering of subsystem designs,” Tech. Rep. HPL-2000-98, HP Labs Technical Reports, 1998.
- [16] D. D. Gajski, F. Vahid, S. Narayan, and J. Gong, “System-level exploration with SpecSyn,” in *Proceedings of the 35th Design Automation Conference*, (San Francisco, California), pp. 812 – 81, 1998.
- [17] T. Givargis, F. Vahid, and J. Henkel, “System-level exploration for Pareto-optimal configurations in parameterized system-on-a-chip,” *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 10, no. 4, pp. 416–422, 2002.
- [18] T. Blickle, J. Teich, and L. Thiele, “System-level synthesis using evolutionary algorithms,” *Journal on Design Automation for Embedded Systems*, vol. 3, pp. 23–58, Jan. 1998.
- [19] L. Thiele, S. Chakraborty, M. Gries, and S. Künzli, “Design space exploration of network processor architectures,” in *Network Processor Design: Issues and Practices*, vol. 1, ch. 4, pp. 55–90, Morgan Kaufmann Publishers, 2003.
- [20] Q. Zhuge, Z. Shao, B. Xiao, and E. H.-M. Sha, “Design space minimization with timing and code size optimization for embedded DSP,” in *Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, (Newport Beach, CA), pp. 144 – 149, ACM Press, 2003.
- [21] P. Czyzak and A. Jaszkiwicz, “Pareto-simulated annealing – a metaheuristic technique for multi-objective combinatorial optimization,” *Journal of Multi-Criteria Decision Analysis*, vol. 7, pp. 34–47, January 1998.
- [22] L. Thiele, S. Chakraborty, M. Gries, and S. Künzli, “A framework for evaluating design tradeoffs in packet processing architectures,” in *Proc. 39th Design Automation Conference (DAC)*, (New Orleans, LA), pp. 880–885, ACM Press, June 2002.
- [23] M. Palesi and T. Givargis, “Multi-objective design space exploration using genetic algorithms,” in *Proceedings of the tenth international symposium on Hardware/software codesign*, (Estes Park, Colorado), pp. 67 – 72, ACM Press, 2002.
- [24] S. Neam, J. Sztipanovits, and G. Karsai, “Design-space construction and exploration in platform-based design,” Tech. Rep. ISIS-02-301, Vanderbilt University, Institute for Software Integrated Systems, 2002.

- [25] S. Bleuler, M. Laumanns, L. Thiele, and E. Zitzler, “PISA — a platform and programming language independent interface for search algorithms,” in *Evolutionary Multi-Criterion Optimization (EMO 2003)* (C. M. Fonseca, P. J. Fleming, E. Zitzler, K. Deb, and L. Thiele, eds.), Lecture Notes in Computer Science, (Berlin), pp. 494–508, Springer, 2003.
- [26] D. Burger and T. M. Austin, “The simplescalar tool set, version 2.0,” *SIGARCH Comput. Archit. News*, vol. 25, no. 3, pp. 13–25, 1997.
- [27] P. Shivakumar and N. P. Jouppi, “Cacti 3.0: an integrated cache timing, power and area model,” Tech. Rep. WRL Research Report 2001/2, Compaq, Western Research Laboratory, August 2001.
- [28] K. Deb, *Multi-objective optimization using evolutionary algorithms*. Chichester, UK: Wiley, 2001.
- [29] T. Bäck, D. B. Fogel, and Z. Michalewicz, eds., *Handbook of Evolutionary Computation*. Bristol, UK: IOP Publishing and Oxford University Press, 1997.
- [30] K. Miettinen, *Nonlinear Multiobjective Optimization*. Boston: Kluwer, 1999.
- [31] J. D. Schaffer, “Multiple objective optimization with vector evaluated genetic algorithms,” in *Proceedings of an International Conference on Genetic Algorithms and Their Applications* (J. J. Grefenstette, ed.), pp. 93–100, 1985.
- [32] J. D. Knowles and D. W. Corne, “Approximating the non-dominated front using the Pareto Archived Evolution Strategy,” *Evolutionary Computation*, vol. 8, no. 2, pp. 149–172, 2000.
- [33] M. Laumanns, L. Thiele, K. Deb, and E. Zitzler, “Combining convergence and diversity in evolutionary multiobjective optimization,” *Evolutionary Computation*, vol. 10, no. 3, pp. 263–282, 2002.
- [34] J. D. Knowles, *Local-Search and Hybrid Evolutionary Algorithms for Pareto Optimization*. PhD thesis, University of Reading, 2002.
- [35] E. Zitzler, M. Laumanns, and L. Thiele, “SPEA2: Improving the Strength Pareto Evolutionary Algorithm for Multiobjective Optimization,” in *Evolutionary Methods for Design, Optimisation and Control with Application to Industrial Problems (EUROGEN 2001)* (K. Giannakoglou *et al.*, eds.), pp. 95–100, International Center for Numerical Methods in Engineering (CIMNE), 2002.
- [36] E. Zitzler, K. Deb, L. Thiele, C. A. C. Coello, and D. Corne, eds., *Evolutionary Multi-Criterion Optimization (EMO 2001)*, Lecture Notes in Computer Science Vol. 1993, (Berlin, Germany), Springer, March 2001.

- [37] T. Back, D. B. Fogel, and Z. Michalewicz, *Handbook of Evolutionary Computation*. IOP Publishing Ltd., 1997.
- [38] E. Zitzler and S. Künzli, “Indicator-based selection in multiobjective search,” in *Proc. 8th International Conference on Parallel Problem Solving from Nature (PPSN VIII)*, (Birmingham, UK), September 2004.