Tools and techniques for efficient system-level design space exploration
Thompson, M.

Citation for published version (APA):
Thompson, M. (2012). Tools and techniques for efficient system-level design space exploration

General rights
It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

Disclaimer/Complaints regulations
If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: http://uba.uva.nl/en/contact, or a letter to: Library of the University of Amsterdam, Secretariat, Singel 425, 1012 WP Amsterdam, The Netherlands. You will be contacted as soon as possible.
Chapter 7

Support for automatic DSE

7.1 Introduction

In the previous chapters we have demonstrated various tools and techniques to evaluate the different aspects of system modeling in the context of the Daedalus design flow. In the introduction (Chapter 1) it was stated that efficient design space exploration requires 1) a method to evaluate a single design point and, 2) a method to efficiently navigate a design space. Given that design spaces grow exponentially in the number of parameters, exhaustive search (evaluating every possible design point) is infeasible. The number of design points can easily outgrow any practical limit on execution time, given a minimum evaluation time of a single design point (limited by the evaluation algorithm or simulator). In this chapter we look at methods to navigate the design space using genetic algorithms (GAs).

A genetic algorithm is a well-known metaheuristic optimization algorithm from the evolutionary algorithm family. Genetic algorithms have previously been used successfully towards solving a wide range of combinatorial problems, such as the bin packing problem or the graph coloring problem [117, 107, 27, 109] as well as many combinatorial problems ([39, 61]). Moreover, GAs can be used in their basic (domain-independent) form, or with custom extensions that incorporate domain-dependent knowledge in order to improve search performance even further. For these reasons, GAs are a promising solution for the traversal of the combinatorial exponentially growing design spaces in the field of embedded system design.

Many attempts in this direction have already been made (these will be discussed in more detail in section 7.2). Although the authors of these works generally report positive results, no authoritative literature on this topic exists yet and—as far as we know—adoption of the methods in commercial tools is currently non-existent. ¹ There may be several reasons why this may be the case. First of all there may be practical reasons that restrict the use of GAs, for example due to the previously mentioned limit on evaluation time, or because it is difficult to find a suitable “genetic” representation for the design problem. Secondly, it may

¹Here we refer explicitly to the use of evolutionary algorithms for the purpose of design space exploration. For example, evolutionary algorithms have been used successfully in commercial place-and-route tools.
be the case that the effectiveness of evolutionary heuristic search algorithms simply has not been sufficiently proven (at least not for generic design problems). Unfortunately, the efforts in this area have been rather fragmented and there is no consensus over which methods are best for certain DSE problems, or alternatively, which has the best generic performance for a wide range of different problems. In any case, further research in this area is required to enhance and solidify our knowledge and understanding in this area.

In this chapter we report our efforts to mitigate some of the common problems and find solutions by evaluating both standard, well-known genetic algorithm techniques as well as some newly developed ones. We present the results of applying genetic algorithm techniques to a few typical Sesame case studies. The focus is on evaluating and improving the exploration of the application-to-architecture mapping. We consider this to be one of the most challenging dimensions of the design space, since it is a non-trivial design parameter with no apparent internal structure and its effects on the design objectives are hard to predict using for example analytical models. Furthermore, as we will explain in Section 7.3, the mapping parameter can be viewed as a “meta-parameter” that incorporates a range of other important design parameters such as the number and type of processors in a system. Although the mapping design space exploration parameter is important, it has not received as much attention as other, more straightforward design parameters (such as strictly architectural parameters related to component properties: component types, clock-timings and memory and buffer sizes). Based on some observations about the mapping design sub-space, we propose and evaluate an extension to a standard genetic algorithm search method. As an additional contribution, we propose, motivate and apply a simple, yet rigorous graphical method to compare different heuristic DSE search methods.

7.2 Related work

The combined DSE search problem that uses simulation or analytical estimation models (to evaluate individual design points) together with a heuristic search method (to traverse the design space) is a relatively new area of research. The need to address this problem is reinforced by the current state-of-the-art of implementation technologies that makes manual system design infeasible (the aforementioned implementation-gap). Heuristic search methods typically do not guarantee finding global optima in the design space, but they are nevertheless able to prune the design space: to reduce the design space to a set of design candidates that meet certain requirements (or are close to the optimum with respect to certain objectives). Evolutionary algorithms are already being used in many different areas in the field of system design, e.g. for place-and-route, netlist generation and for reliability testing and validation purposes. In the following however, we limit ourselves to applications for the purpose of system-level design space exploration. Moreover, we focus on those approaches that 1) make a clear separation between the parameter and objective spaces, 2) use analytical estimation methods or simulators to evaluate design points and 3) traverse the design space by means of an evolutionary algorithm or a related method. Works that fall in this category still have a striking variety of contexts, application domains and types of architectures.
Moreover, there is a large disparity in the types of design problems, the design parameters and objective metrics, design space sizes, evaluation mechanisms and optimization methods. And finally, many different qualitative evaluation methods are used to evaluate and compare proposed search methods. In the following we will not give an exhaustive comparison of related work, but rather emphasize some of the most important differences and similarities.

To our knowledge, the earliest work that meets the previously mentioned criteria is that of [101]. A method is proposed to optimize the system-level design (although referred to as synthesis) of a dataflow-based application implemented by an MPSoC architecture. Application and architecture are represented as specification graphs, which are tied together with edges representing expert knowledge indicating the (in)feasibility of mappings. A multi-objective fitness function (performance and cost) is defined that adds extra penalties to steer the evolutionary algorithm away from infeasible population individuals. A case study reports that the proposed method is able to efficiently find pareto-optimal points.

We also refer to earlier work from our research group ([25]), proposing a solution towards solving the design space search problem of mapping Kahn process networks (KPNs) onto heterogeneous MPSoCs. A mathematical model expresses the three objectives (performance, power, cost) of a design point as an analytically solvable non-linear mixed integer programming problem. This is then used as the fitness function for two elitist evolutionary algorithms (SPEA2 [120] and NSGA-II [21]). A case study shows that (according to three newly defined metrics) they perform similarly, but that NSGA-II is preferred due to its less computationally intensive fitness assignment scheme.

Also closely related to the work presented in this chapter is [77], where a standard simulated annealing algorithm is extended with automatic parameter selection. The design problem consists of mapping tasks from a synthetically generated KPN network onto a 2-4 processor architecture: with typical design spaces consisting of 10s of millions of candidates. The proposed method is compared against random search, group migration and standard simulated annealing by looking at the relative quality of results in terms of the objective space (performance). It shows that for a given class of KPNs, the proposed method finds better results for the same number of mapping evaluations (if that number is sufficiently high).

In the work of [37], the design space search problem is described as a Markov Decision Problem (MDP) and design space traversal is defined as a sequence of movement vectors between states. Movement vectors change states in parameter space (number of processors, I/D cache size) and approximate analytically the impact on the objectives (power and performance). A major advantage of this approach is that simulation only needs to be applied when repeated application of movement vectors exceeds a predefined level of estimation error. A case study is presented where MJPEG4 and Ogg-Vorbis applications are mapped onto a 2-8 ARM-processor MPSoC with varying I/D cache sizes. It shows an 80% reduction in evaluation time compared to tabu search or simulated annealing, or conversely, it finds better results given the same evaluation time. Note that exploration of explicit task-to-architecture mappings is not supported, probably because it would be hard to define a sufficiently accurate movement vector for this design space parameter.
In the work of [29] a method based on Ant Colony Optimization (ACO) is proposed to optimize mapping and scheduling of an application onto a 4-processor MPSoC platform. The application is specified as a parallel task graph and mapping concerns both task mapping and communication mapping. The proposed method is compared against other metaheuristic algorithms such as simulated annealing, taboo search and genetic algorithms. Case studies consisting of both synthetic and a JPEG application show that the proposed ACO method finds better average results and reaches optimal solutions faster (though at the cost of a higher execution time) compared to the other methods.

While these previous works typically focused on specific DSE methods that were very much tied to a specific context (simulation tools, architectural platforms or application domain), the work in [48] makes a commendable effort to propose a generic infrastructure for system-level DSE. The NASA framework aims to be modular, extensible, flexible and reusable. For example: the search method(s), feasibility checker, platform generation, evaluation (e.g. a simulator) and fitness functions have been de-coupled as separate parts and use well-defined interfaces so that substitute components can be used in a plug-and-play fashion. Components that conform to the interfaces are much easier to compare and code for setting up and evaluating experiments can be reused. An example of the flexibility and modularity of the system is illustrated by the framework’s proposed dimension-oriented DSE approach, where different design dimensions are co-explored using the same or (optionally) different search algorithms. A further innovation is that instead of using a fixed architectural topology, NASA constructs the topology from so-called Basic Topology Units (BTUs), which consist of a network container and a number of element containers. According to a custom user specification, these BTU building blocks will be automatically connected and the containers will be instantiated with components such as buses, bridges and links (network containers) or processors and memories (element containers). This allows for automatic exploration of a much wider range architectural topologies than is commonly the case.

To conclude, we will shortly clarify the position of the work presented in this chapter relative to the mentioned related works. As in the previous chapters, we consider streaming multimedia applications (as in [25, 77, 101, 17]), and more specifically we use Kahn process networks (as do [25, 77]). Since the focus of this chapter is to study DSE search algorithm behavior rather than optimization of a specific application or platform instance, we use synthetically generated KPN workloads (using a modified version of the KPN workload generator of [77]). This is also the reason why (in contrast to most other works [17, 101, 37, 25, 48]), we propose a way to compare and evaluate different algorithms independently of objective space metrics. Design points are evaluated using our Sesame system-level simulator: [17, 48, 77] also use simulation-based evaluation, whereas other approaches use analytical methods ([25, 101]) or a mix of both ([37]). Simulation-based evaluation can typically more accurately determine characteristics of actual design points, but may be slower than analytical approximations. Although we use the Sesame simulator, we aim for our methods to be equally applicable to other contexts (possibly using different simulators) while still being able to provide a useful comparison (e.g. by measuring search method efficiency in the number of required evaluations instead of time).
7.3. DSE AS A GA SEARCH PROBLEM

Our method of choice for searching the design space is a genetic algorithm (like [101, 25, 48]) using a mapping based genetic representation (like [101, 25]). However, we propose methods to optimize the standard GA using newly developed techniques that (as far as we know) have not been used previously in the field of system-level DSE (see Section 7.8). Our methods for optimization are not only relevant since the mapping-based representation is commonly used (e.g. [25, 101, 48]), but also because they are highly compatible with many other optimizations (so that different optimizations can be applied simultaneously). The work in [101] and [17] report that their genetic algorithms manage to find the true optimum in the design space. In both cases, however, the identification of the true optimum seems to be given by means of some (not further specified) analytical method. However, it is not addressed if and how the presence of such an analytical solution influences their respective evolutionary search algorithms. For example, it is possible that a genetic algorithm is able to converge more easily if there are true-optimal solutions which are (apparently) not too hard to derive. We clarify in advance that the search problems from this chapter have been set-up in such a way that there is no known method to derive the true optimum of the design space, other than by exhaustive search (which, as we will see, is often infeasible).

7.3 DSE as a GA search problem

In this section we discuss the representation of a DSE problem as a search problem which is suitable for solving by a GA. GAs operate by searching through the solution space where each possible solution has an encoding as a string-like representation (often referred to as the chromosome). A (randomly initialized) population of these chromosomes will be iteratively modified by performing a fixed sequence of actions that are inspired on their counterparts from biology: evaluation and selection, crossover and mutation. This sequence of actions is in fact the only common denominator between GA implementations. There are many different implementation options for each step in the GA and choosing the right implementation is “the art of evolutionary algorithms”, since (unfortunately) it is mostly a matter of designer’s intuition and trial and error. A fundamental design choice is the genetic representation of the solution space, because each of the selection, crossover and mutation steps depends on it. Furthermore, it is known from literature that for some search problems the representation can influence GA performance. However, from a practical perspective, one typically chooses a representation that naturally fits the problem at hand, as this simplifies the implementation of the crossover and mutation operators.

Design point representation

In our case the problem is finding an optimal design candidate in a large space of possible design candidates that can be evaluated within Sesame. In Chapter 3 we saw that a design space in Sesame often consists of parameters that are related to the application-to-architecture mapping. For example, the number processors of a design can be completely determined by the mapping: processors to which no processes have been mapped can simply
be discarded. Similarly, if there is a choice between $N$ different types of processors and a maximum of $M$ processors, then the meta-platform (as introduced in Section 3.4) simply contains all $M \times N$ processors and the mapping determines the final configuration. If a suitable meta-platform can be found for a given DSE experiment, then the mapping specification provides a complete representation of a design point. However, creating a meta-platform can be hard for some architectural parameters, especially if the architectural topology is one of the design parameters. Furthermore, it doesn’t work with “active” components (which are not dependent on input) or components that start simulation with an initialization routine, since these components may affect the simulation even if nothing is mapped onto it. Finally, one has to consider that the meta-platform specification grows combinatorially with the number of parameters. This can cause practical problems, considering that even unused model components have to be stored on disk (before simulation) or in memory (at runtime). However, as we will see below, a mapping-based representation using a meta-platform enables a very straightforward GA representation. We will use this representation for the case-studies throughout this chapter, since they are designed such that they do not suffer from the previously mentioned disadvantages.

![Figure 7.1: A simple homogeneous crossbar model](image)

For the experiments in this chapter we will use a Sesame model that maps an application onto a crossbar-based meta-platform (Figure 7.1). A GA search algorithm will be used to find the best application-to-architecture mapping. Here we discuss the representation of the mapping as a GA chromosome. Firstly, we assume that there are no functional restrictions on the processors: all processors can execute all of the tasks (this is generally true for programmable processors). Secondly, we assume that each pair of processors can communicate so that there are no topological restrictions. Indeed, the crossbar in the proposed platform fully connects all processors, so processes can communicate regardless on which processor they are mapped. The result is that any task can be mapped onto any processor so that we do not have to make special provisions for infeasible mappings. As we saw in Chapter 3, KPN communication channels in Sesame are mapped explicitly onto communication structures in the architecture. However, the crossbar uses a local-write, remote-read paradigm, so that outgoing application channels are always mapped to the FIFO memory that is associated with the processor. This means that the channel mapping does not need to be specified, since it can be trivially derived from the task-mapping. So finally, we can conclude that the
7.3. DSE AS A GA SEARCH PROBLEM

platform can be completely specified by simply associating application tasks to processors:

Task 1 $\Rightarrow$ Processor 1
Task 2 $\Rightarrow$ Processor 2
Task 3 $\Rightarrow$ Processor 1

\ldots

Task $n$ $\Rightarrow$ Processor $k$

As a more convenient mapping description, we can use a vector of $n$ processor identifiers, where the $i$-th index indicates the mapping target of task $i$:

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

Note that this commonly used description is very suitable to serve as the chromosome representation (or genotype) for a genetic algorithm. All possible combinations of integers will result in valid mappings, as long as $\forall i : p_i < k$. This means that implementing crossover and mutation operators will be relatively easy, since any recombination of such chromosomes results in a valid specification and the GA can be implemented without special consideration for “broken” chromosomes or repair mechanisms.

As a subsequent simplification, we will assume a homogeneous architecture where all processors are the same and a task incurs the same delay on each processor (except for additional delays caused by interfering tasks mapped onto the same processor). This condition also includes that the architecture is symmetrical: other delays from the system that affect the processor (e.g., network delays due to communication, memory delays, task scheduling overhead, etc.) should also be the same for each processor. This condition holds for many homogeneous architectures, for example, a multi-processor system where all processors are connected to a central bus with a non-prioritizing arbiter or indeed the crossbar system used in our model. A valid mapping specification is a partitioning of all $n$ processes, where partitions may be empty or contain all $n$ processes. Note that partitions may be empty (processor not in use) or contain all $n$ processes (a single processor system). A processor that is not assigned any tasks (having an empty task partition) can be considered idle or non-existent.

Note that many of the techniques presented in this chapter can be quite easily extended to more heterogeneous systems (this will be discussed as future work in Section 7.10).

Size of the design space

For a maximum of $p$ processors in the platform, there are a total of $p^n$ possible mapping descriptions. We need to be aware however, that multiple mapping specifications can describe the same mapping. For example, mapping $A : \{0, 1, 0, 0, 2, 3\}$ and $B : \{2, 0, 2, 2, 3, 1\}$ denote the same (duplicate) partitioning, and therefore an equivalent mapping on a homogeneous platforms. This can be solved by using a normal form notation of mappings, but here we introduce the mapping distance metric which can distinguish duplicate mappings and which will prove a useful concept later on. This metric is similar to the Hamming distance, or edit distance, except that it is independent of the symbols that make up the word.
We define the mapping distance $\delta(p,q)$ as the minimum number of task re-mappings that is required to transform mapping $p$ in mapping $q$. For example, $\delta(A,B) = 0$ and $\delta(A,C) = 4$ if $C : \{0,1,2,3,1,1\}$. In general we have that for any $p$ and $q$: $\delta(p,q) = \delta(q,p)$ and for duplicate mappings: $\delta(p,q) = 0$. Note that with minor modifications, the distance metric can also be defined for heterogeneous platforms (where duplicate mappings can still occur on groups of processors with the same type). A more detailed explanation of the distance metric is given in Section 7.6.2.

We can quantify the size of the design space as the number of partitionings (see Table 7.2). We can see that the design space is orders of magnitude smaller when we discard duplicate mappings, which is therefore an important consideration for the search algorithm. In the following we will show results of experiments where a genetic algorithm has been applied to our mapping design space using the vector mapping representation. Then we will give some suggestions on how the mapping distance technique can be used to improve the search algorithm. From the table we immediately see the exponential growth of the design space: slight increases in the problem size ($n$ or $k$) show orders of magnitude increase of the design space. We also observe that the number of unique mappings grows much slower than the number of chromosome representations (though it still grows exponentially). The difference between the number of all and unique representations is approximately $k!$. In the next Section we investigate the GA behavior for one of these problems sizes.

<table>
<thead>
<tr>
<th>$n$</th>
<th>$k$</th>
<th>all descriptors ($k^n$)</th>
<th>unique points ($\sum_{i=1}^{k} S(n,k)$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>3</td>
<td>243</td>
<td>41</td>
</tr>
<tr>
<td>7</td>
<td>3</td>
<td>$&gt; 2K$</td>
<td>365</td>
</tr>
<tr>
<td>9</td>
<td>4</td>
<td>$&gt; 262K$</td>
<td>$&gt; 11K$</td>
</tr>
<tr>
<td>11</td>
<td>4</td>
<td>$&gt; 4M$</td>
<td>$&gt; 175K$</td>
</tr>
<tr>
<td>13</td>
<td>4</td>
<td>$&gt; 67M$</td>
<td>$&gt; 2M$</td>
</tr>
<tr>
<td>15</td>
<td>4</td>
<td>$&gt; 1G$</td>
<td>$&gt; 44M$</td>
</tr>
<tr>
<td>15</td>
<td>9</td>
<td>$&gt; 205T$</td>
<td>$&gt; 1G$</td>
</tr>
</tbody>
</table>

Figure 7.2: Number of (unique) design points for given $n$, $k$

### 7.4 Initial case study

Here we present the results of an initial, small-scale case study where the design space consists of an 11-process application mapped onto a 4-processor crossbar architecture. We also introduce a graphical method to quickly and concisely summarize and compare the results of different GA experiments, followed by a short note on implementation issues. But first we analyze the design space by exhaustive search.
Size and shape of the design space

As we can see from Table 7.2, a design space with \( n = 11 \) and \( k = 4 \) contains more than 4 million (175K unique) design points. We use an algorithm based on Nijenhuis and Wilf ([73], Chapter 11) to generate all unique design points and iteratively evaluate the Sesame model for each point. The result of this exhaustive search is that we have a complete view of the relationship between parameter space and objective space. We can pin-point the global optimum and use it in subsequent experiments to rate the relative quality of the results found by the GA. The unique design points are shown in Figure 7.3a in the order of Nijenhuis-generation and in Figure 7.3b as sorted by performance. Note that the same 275K of points are plotted in both figures, although more of them overlap in the sorted figure. The latter

\[
\begin{bmatrix}
    q_1, r_1, s_1, t_1, q_2, r_2, s_2, t_2, q_3, r_3, s_3, t_3, q_4
\end{bmatrix}
\]

evaluates to ztyuq clock cycles. Note that we can not make general assumptions about the shape of the design space and the distribution of high-quality results, which is likely to be different for other architectural platforms and application workloads.

Evaluation method

When analyzing results of heuristic search algorithms that are based on a random initialization (e.g., the initial population in a Genetic Algorithm), we have to be aware that the result may be different each time the algorithm is run. In figure 7.4a we show the quality of the best design point found by 128 runs of our GA (we discuss the specific GA parameters later). Each result is shown as a percentile of its location in the sorted design space: this gives us an idea how “close” a design point is relative to the global optimum. In figure 7.4b the same data is shown as a histogram: although many results are very close to the global optimum (0-0.1 percentile), closer inspection of the data shows that the global optimum is not found in any of the experiments. From this we conclude that our sample design space is already quite
challenging for our GA. One can imagine that for bigger problem sizes, or more elaborate
design spaces with more parameters, the design space search problem becomes even harder.
In any case, it is critical to find search algorithms that show good convergence towards the
optimum (whether they find the global optimum or not).

![Figure 7.4: Results of repeating a GA 128 times](image)

In order to compare different search methods, we propose an evaluation method that is
not based exclusively on the use of metrics in the objective space (in our case the simulated
system runtime). Although objective space metrics are commonly used for this purpose,
they can give a distorted view of the results, since the “landscape” of the design space can
be unpredictable. For example, if we compare two search methods A and B for their per-
formance on the design space in Figure 7.3b, then a difference between the search results
of 250000 units in metric space equals a difference of 20000 design points if measured in
the beginning segment of the sorted design space (closest to the optimum), but up to 110000
points in the middle segment of the design space. In this research we are more interested in
a search method’s convergence behavior, instead of the absolute values of the result. There-
fore, we use a new metric that performs a fair comparison of different search methods (or of
the same search method in different design problems). The metric is most easily represented
as a graph, such as in Figure 7.5 where each line summarizes the results of all repetitions
of one specific search method. The horizontal axis represents the quality of the result as a
percentile towards the 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. If one
method is repeated \( r \) times, then the graph consists of \( r \) discrete dots, but a line is plotted
through for visibility. In the following we will refer to this kind of plot as a Probability-
Quality plot (PQ-plot). Since the x-axis requires the calculation of the result’s percentile
towards the optimum, PQ-pots like this are only possible for design spaces that have pre-
viously been exhaustively evaluated. In Section 7.8 we show a solution to represent larger
design spaces in a similar way.

As an example consider the PQ-plot in Figure 7.5: the middle line shows the results of
a GA with the following parameters: population size 15, 15 iterations, crossover probability
0.9, mutation probability 0.01. It shows that a result within the 0.1 percentile (belonging
to the best \( \frac{1256}{1000} \approx 175 \)) results can be achieved approximately 28% of the time (which has
7.4. INITIAL CASE STUDY

Figure 7.5: PQ-plot for repeated experiments of 2 different types of GA and one random search

been derived from repeating the method \( r \) times. The lowest, smooth line represents the theoretically derived probabilities of finding results using uniform random search (as will be discussed below). The top line represents another search method which dominates the middle line, indicating a better performing search method: it has been found that it has a higher probability of reaching search results with the indicated quality (or higher). In general it is desirable that a PQ-line that climbs rapidly to Probability=1, therefore PQ plots show only the interesting part of the x-axis (typically within 0.1-0.2 percentile). By comparing the lines for different search algorithms we can get an insight the relative (average) performance behavior.

An additional advantage of expressing the quality of an experimental result relative to the optimum is that we can make an easy performance comparison to random search. We define random search as the method that takes \( r \) random samples from the design space and returns the best result. It serves as a comparative baseline for experiments which can be plotted in a PQ-plot: each proposed algorithm should at least perform better than random search. We choose \( r \) to be equal to the number of evaluations of the other algorithm, so for a GA this would be population size \( \times \) number of iterations When comparing different population sizes, multiple random-baselines will be shown: the plot key then lists the number of evaluations in braces to identify which baseline belongs to a particular GA experiment. Instead of performing an actual random search, we can theoretically derive the random baseline (the probability of finding a result within C percentile of the optimum with \( r \) random samples):

\[
P_{\text{find}} = 1 - \left( 1 - \frac{C}{100} \right)^r
\]

Although the PQ-plots give a detailed overview of a search method’s behavior, it does not specify the statistical significance of the difference between two or more search methods. In other words: when lines are close together, does the one on top represent a truly better method, or is it just a side-effect of the non-determinism of the initial random population? In order to discriminate these cases we compute for each experiment an approximate
confidence interval of the mean value of the \( r \) repetitions of a single search method. The confidence interval indicates how certain (as specified by the confidence level) we are that the real mean (mean and variance of the distribution represented by the \( r \) samples are unknown) lies within the confidence interval. The more the confidence intervals for different experiments are non-overlapping, the more significant the difference of the mean behavior. The confidence intervals for the results in Figure 7.5 is given in Figure 7.6 for different confidence levels (the mark in the center of a confidence interval represents the sample mean).

![Figure 7.6: Confidence intervals for the experiments in Figure 7.5](image)

The boundaries of the interval are computed as follows: 
\[
\text{mean}(x) \pm t(r-1)\frac{sdev(x)}{\sqrt{r}}
\]
where 
\( t(r-1) \) is the upper \((1-C)/2\) critical value for the \( t \) distribution with \( r-1 \) degrees of freedom. Note that the confidence interval is only approximate, since there is no reason inherent to the model to assume that the samples will be normally distributed. However, we have performed several normality-tests on results of repeated GA experiments, showing they indeed follow the normal distribution quite well \(^2\).

**Implementation**

The results in this chapter are obtained using a standard genetic algorithm library. In the next section we look at the DSE results using the unmodified (standard) GA, but in Sections 7.7 and 7.8 we introduce certain additional features and GA operators. Experiments can be done using “simulator-in-the-loop” (where the fitness function consists of a call to the Sesame simulation model), or using a database of cached simulation results. A separate program is available to exhaustively search the design space by generating every (unique) design point, run it through the simulator and store the result in a local database. In that case the GA fitness function is simply a lookup in the database of cached simulation results, which is of course only possible for relatively small design spaces. Considering the large number of GA-based experiments that has been done for this research, we exploit easily available parallelism in order to speed up the runtime of experiments. We can use parallelism at two levels: for evaluating single design points (e.g., in order to create the database exhaustive results) or for repeating experiments to measure average search-performance results (the experiments will run in parallel). The program to start and evaluate these DSE experiments

\(^2\)The various distributions that are inherent to the simulation model (modeling different latencies in all parts of the model) may indeed accumulate to a normally distributed performance number, perhaps as an effect of the *central limit theorem* (CLT). However, we have no way to motivate the occurrence of normality or to prove applicability of the CLT. Therefore we emphasize that we claim (approximate) normality on the basis of the observations only.
7.5 Initial case study - parameters

In this section we perform an initial exploration of the effect of different parameters on the performance of the standard GA. We compare the results with the theoretical random-search baseline and with each other. The parameters available in typical GA implementations are: population size, crossover and mutation rate, selection strategy and selective pressure. Furthermore, we investigate the population diversity throughout the run of the GA. Although the experiments in this section give us an indication of good GA parameters, we can’t assume that these will be suitable for all case-studies in general (where design spaces and problem sizes may be different). When we consider a different design problem, we always perform sanity-checks to see that (for the given problem) we are still using a good set of GA parameters. Here we report our findings for the case study from the previous section. We discuss the issue of the generic applicability of these findings later in this chapter.

First we consider the population size parameter, which can be regarded as one of the most influential parameters. Larger populations will typically result in better GA performance, since the initial (random) population is more diverse and because the GA has more internal parallelism, so that more recombinations can occur in the same iteration (thus possibly converging faster to the optimum). However, larger populations come at the cost of more evaluations. For the current design space we can simply use pre-computed results from a complete store in memory, but when using the simulator to evaluate chromosomes, the GA execution time will increase with larger populations. The extra execution time can sometimes be mitigated by the fact that the GA converges faster and fewer GA iterations are required or because a much better optimum is found. Therefore, the best GA is a trade-off between number of evaluations (depending on population size, number of iterations, population diversity) and resulting quality. In Figure 7.7 the P-Q plot is given of 300 repeats of a GA with different population sizes and constant mutation rate (0.1), crossover probability (0.9), tournament selection and a uniform crossover strategy. We can see that the largest pop-
population size (80 chromosomes) has a better chance to find results of certain quality. Note that the figure shows multiple random-search baselines for fair comparison with each of the GAs, allowing random-search the same number of evaluations (as indicated by the last number in the plot legend). From the results we conclude that although population size 80 obtains the best result, a population of 40 is a reasonable performance-execution time trade-off. We take population size 40 as a basis for comparison in the remainder of this section.

Next we look at the influence of the mutation parameter, the left side of Figure 7.8a shows the result for population size 40. In both figures we see a trend that a very small mutation rate does not yield a good performance (mutation 0.01, 0.02), nor does a very high mutation rate (0.5). The most effective mutation rate seems to be between 0.05 and 0.2. The existence of an optimal mutation range can be explained by the fact that mutation enables diversification in the population. This can result in new optimal solutions to be found and to avoid sticking to local optima in the design space. However, when there’s too much mutation, the GA becomes more and more like a random search and is unable to converge towards any optima (local or global). It seems that the mutation rate has an even greater influence on a GA with a smaller population (compare e.g. the difference between mutation 0.05 and 0.20 in both figures). A reasonable explanation seems that the larger population has more randomness from its initial population, whereas a smaller population does not. For now we select mutation rate 0.10 as the basis for our future experiments.

In Figure 7.9 we see performance for different crossover probabilities. A higher crossover probability indicates that two genes exchange more genetic material once they have been selected to create offspring. We can see that the results are close together for significantly different crossover rates, suggesting that the crossover rate has little influence on the GA. Indeed, in the wider GA research field it is a hot topic of debate how important actually is the crossover step in a GA. There is no general consensus on this topic, but including some form of crossover seems to be generally beneficial. In the following, we will continue to use
7.5. INITIAL CASE STUDY - PARAMETERS

Figure 7.8: P-Q plot for different mutation rate

the commonly applied 0.9 crossover rate.

Figure 7.9: P-Q plot for different crossover probabilities

Furthermore, the method that is used to select which chromosomes are candidates for crossover may affect GA performance: the selection strategy. A selection strategy typically selects two chromosomes from the population, which the crossover method will then combine into two new chromosomes. Here we consider some of the commonly used selection strategies: roulette-wheel (selection proportional to the chromosome fitness), tournament (select fittest chromosome from a small random subset of the population) and random (select random chromosomes from population). For each selection strategy we consider a few types of crossover mechanisms: single-point crossover (breaking a chromosome at a random point and exchange the end-pieces), two-point crossover (breaking a chromosome at two points and exchange the middle piece) and uniform crossover (a child chromosome is built-up as a sequence of genes that may come from any one of the parents). The results are shown in Figure 7.5. Each graph (7.10a,7.10b,7.10c) shows the result of a particular
selection strategy in combination with the three types of crossover. The last graph (7.10d) compares the best result of each selection type. First of all we observe that the PQ-lines in each case lie very close together: apparently the choice of crossover method does not influence GA performance in a significant way. The overlapping confidence intervals confirm this: only at a low significance level can we distinguish the use of different crossover operators. For example, in Figure 7.10b, only the single-point crossover has a non-overlapping confidence interval, but only for confidence levels 0.85 or lower. In Figure 7.10d the best result for each selection type is plotted, and we see that tournament selection (with two-point crossover) shows a measurable improvement over roulette and random selection.

One final GA parameter is selective pressure, which can be defined as the preference of the selection method to choose chromosomes with a better fitness over chromosomes with worse fitness. A high selective pressure can allow a GA to converge quicker, but can also mean that the GA is likely to get stuck in local optima. With selective pressure too high, only successful chromosomes make it into the next population and successful combinations with lesser chromosomes may be missed. This can also be translated into a diversity problem: only genetic material from chromosomes with high fitness make it into the new population, so the diversity of the new population may be low. On the other hand, when selective pressure is too low, a GA will have problems converging to any optimum (thus resembling a random walk). Finding a good balance between convergence and diversity is considered an
important issue in the field of evolutionary algorithms. We will revisit this topic later in this section and throughout this chapter.

In our current experiment we can most easily try out different selective pressures for the tournament selection method. Selective pressure can be increased by increasing the size of the random subset of the population (from which the chromosome with highest fitness is selected) as weak individuals have to compete with more other chromosomes. Vice versa, selective pressure goes down for a smaller set (note that set-size 1 is equal to random selection). The results in Figure 7.11 show that varying subset sizes between 2 and 16 has very little influence on GA performance for the current problem.

![Graph showing P-Q plot for varying selective pressure](image)

**Figure 7.11: P-Q plot for varying selective pressure**

Next we take a look at the diversity of the populations during the run of the GA. For this purpose we measure in every generation of the GA how many of the population individuals are new (have not appeared in any previous generation). Since the results show a similar trend in the different experiments discussed so far, we only show the diversity for the experiment with different mutation rates. The left side of Figure 7.12 shows the number of new individuals per generation, whereas the right shows the same data cumulatively. The points shown are averages for 300 repetitions of each GA; the initial population is shown as generation 0. Not surprisingly, the highest mutation rate (0.20) introduces the most new individuals per generation. In all cases, the number of new individuals drops quickly even after the second generation, and the last generations barely introduce more than 1 new individual. In the right figure the last data point on each graph shows the total number of unique values throughout the run of the GA. We observe that increasing the mutation rate four-fold (from 0.05 to 0.20), the diversity increases by only 23%. Moreover, we know from our previous experiments that increasing mutation rate beyond 0.20, has an adverse affect on the quality of the found optima. Indeed, it is generally known that there is a delicate balance between
Figure 7.12: Diversity per generation (left) and cumulative (right)

diversity (which can be improved by higher mutation rate) and convergence (which may be disrupted by a too high mutation rate): a higher mutation rate may prevent the GA from getting stuck in local optima, but may also prevent convergence towards the best solution. We revisit this topic in Section 7.7, where we propose a new method of managing population diversity based on the distance metric.

The diversity analysis of Figure 7.12 suggests a particular performance optimization of the GA: individuals that occur multiple times do not really need to be evaluated again. By storing the evaluation result of each individual, the number of calls to the simulator can be reduced. We have not applied this optimization in our case, but the performance of the GA could be increased significantly this way. This is particularly true for DSE case studies that perhaps use a simulator that takes orders of magnitude more time to evaluate a single design point (e.g., instruction set simulators). As an example, the GA with mutation rate 0.2 calls the simulator $40 \times 16 = 640$ times (including the initial population), out of which only 173 on average are unique individuals. By using the proposed optimization, a speedup of approximately 3.6 can be obtained over a non-distributed GA implementation (where evaluations are performed sequentially).

Finally, we note that the baseline in the PQ plots is based on uniform random search using the number of design point evaluations as given by the non-optimized number of evaluations:

$$(\text{number of generations} + 1) \times \text{population size}$$

If we would allow random search to evaluate a number of individuals that is equal to the accumulated number of unique individuals in each of a GA’s populations, then the PQ-plot of the GA would show even better performance relative to the random search baseline.

7.6 Distance metric

In this section we expand on the topic of the distance metric, which was shortly mentioned before in Section 7.3. First we discuss some properties of the design space that motivates the possible usefulness of such a metric. Subsequently, the distance metric will be described in
7.6. **DISTANCE METRIC**

detail and possible implementations are discussed. In the remainder of this chapter we will consider how the standard GA methods can be improved using the distance metric.

### 7.6.1 Observations on the design space

As we have described previously, our chosen design space focuses on the mapping between application tasks and architectural resources. The performance of a single design point is heavily influenced by the (communication) dependencies between nodes in the task-graph and the dependencies that are introduced by sharing of architectural resources. The Sesame simulator captures these dependencies in its models and the resulting performance evaluation. A small change in the dependencies can, in theory, result in a completely different performance result. For example this is the case when we change the mapping of a single task such that it is added to or removed from the dependency chain that is part of a performance bottleneck in that design point. However, intuitively, we think that in general most small changes will not result in a hugely different performance result. We confirm this hypothesis by checking the correlation of performance results for pairs of design points. First a random set of design points is created, then we mutate each chromosome in $x$ positions by randomly choosing the positions and its mutation value. The results are shown in Figure 7.13 for a design problem where an application with 20 nodes is mapped onto a homogeneous, symmetrical architecture with 8 processors. Each dot represents a pair of design points with the performance of the first design point along the $x$-axis and the performance of the second along the $y$-axis. We can see a clear correlation between points that have mutated in 1 or 2 positions (top graphs), but this relation fades and disappears with changes in 4 or 8 positions (bottom graphs). Apparently, it is indeed true that (on average) small changes in the design point specification lead to small performance differences. We propose methods to exploit this correlation by integrating this piece of domain knowledge into our GA methods in Section 7.8.

### 7.6.2 Distance metric details

Here we present an implementation of the distance metric as defined in Section 7.3. For any pair of mappings $(A, B)$, the algorithm will perform stepwise reassignment of tasks in $B$ such that the result is equivalent to $A$ and that the number of required reassignments is minimal. The algorithm considers the mapping as a partitioning of a set of task groups: processes mapped onto the same processor are in the same task group. In each recursive iteration of the algorithm, a pair of task groups $(t_{gA}, t_{gB})$ will be selected where $t_{gA}$ is a task group from mapping $A$ and $t_{gB}$ is a task group from mapping $B$. Next, certain tasks will be reassigned such that group $t_{gB}$ becomes the equivalent of task group $t_{gA}$. This continues until the task groups in each mapping are equivalent: this means that mappings $A$ and $B$ are now equivalent. The essence of the algorithm is to find a sequence of task group pairs such that the accumulated number of reassignments to turn $A$ into $B$ is minimal.
Let $\text{map}$ be a function that maps $n$ tasks onto a $k$-processor system.

$$\text{map} : \{0 \ldots (n - 1)\} \rightarrow \{0 \ldots (k - 1)\}$$

Let $A$ and $B$ be two mappings where the $i$-th element in $A$ is denoted as $A[i]$:

$$A = [A[0], \ldots, A[n - 1]] = [\text{map}_A(0), \ldots, \text{map}_A(n - 1)]$$
$$B = [B[0], \ldots, B[n - 1]] = [\text{map}_B(0), \ldots, \text{map}_B(n - 1)]$$

Then a task group of a mapping $A$ is defined as a set

$$\text{tg}_{A,x} = \{t \in \{0 \ldots n - 1\} \mid \text{map}_A(t) = x\}$$

Note that when mappings $A$ and $B$ do not map to the same number of processors, then one of the mappings has some empty task groups (which does not influence the working of the mapping distance algorithm).

Next we list the three important stages from the recursive algorithm:

**Step 1: group selection**

Find a task group from each mapping to form a pair:

$$(\text{tg}_{A,i}, \text{tg}_{B,j}) \quad (i, j \in \{0 \ldots k - 1\})$$

such that:
7.6. **DISTANCE METRIC**

1. they share the maximum number of tasks: \(i, j\) with \(\max(|\text{tg}_{A,i} \cap \text{tg}_{B,j}|)\), and

2. \(i\) and \(j\) have not been part of a task group pair in a previous iteration of the algorithm

If there are no more task groups that meet the requirements, then \(B\) is now equivalent to \(A\) and the algorithm has finished. The accumulated value of the distance counter is returned as well as the sequence of task group pairs that was used to rewrite \(B\) to \(A\) (the latter will be used later to generate a “minimum path” from \(A\) to \(B\), see Section 7.8.2. Note that there may be more than one sequence that transforms \(A\) to \(B\) with the same number of reassignments, but finding one such sequence is sufficient for our purpose.

**Step 2: recursion**

The pair found by the previous step will be used for task reassignment. However, there may be multiple pairs that have intersections of the same (maximum) size. In this case it is unknown which pair should be used for reassignment, so there’s no other option than to try all of those pairs. To this end, a copy \(B'\) of \(B\) is created of every pair and the task reassignment function (see below) is applied. Next, the distance algorithm will be called recursively to calculate the distance between \(A\) and every \(B'\). When the recursion has finished, we select and return:

- the minimum found distance between \(A\) and \(B'\)
- the corresponding sequence of task pairs

**Reassignment Function**

This function takes as input a task group pair \((\text{tg}_{A,i}, \text{tg}_{B',j})\) from mappings \(A\) and \(B'\) respectively. The pair will now be used to modify mapping \(B'\) such that \(\text{tg}_{B',j}\) includes at least those tasks that are in \(\text{tg}_{A,i}\):

\[
\forall y \in (0 \ldots k - 1) : \quad B'[y]^{\text{reassign}} := j \quad \text{if} \quad A[y] = i \quad \text{and} \quad B'[y] \neq j
\]

This results in: \(\text{tg}_{A,i} \subseteq \text{tg}_{B',j}\). Note that the additional tasks \(\{\text{tg}_{B',j} - \text{tg}_{A,i}\}\) (if any) will be reassigned in a later iteration such that finally \(\text{tg}_{A,i} = \text{tg}_{B',j}\). Note that the number of reassignments may be 0, in which case the distance counter is not increased.

**7.6.3 Example**

To illustrate the above algorithm, consider the following two mappings \(A\) and \(B\) for a 6 process application and any 4 processor (homogeneous and symmetrical) architecture.

\[
A : [0, 1, 2, 2, 2, 3] \quad B : [0, 1, 1, 0, 0, 0]
\]

In the **first iteration** of the algorithm we have the following task groups:
We find that the pair \((\text{tg}_{A,2}, \text{tg}_{B,0})\) has the largest overlap (tasks 3 and 4). Therefore: \(B'[y] = 0 \text{ if } A[y] = 2\), results in: \(B' = [0, 1, 0, 0, 0, 0]\). The distance counter is incremented by 1, because task 2 was reassigned. In the second iteration the relevant remaining task groups are:

\[
\begin{align*}
\text{tg}_{A,0} &= \{0\} & \text{tg}_{B',0} &= \{1, 2\} \\
\text{tg}_{A,1} &= \{1\} & \text{tg}_{B',1} &= \{\} \\
\text{tg}_{A,3} &= \{5\} & \text{tg}_{B',3} &= \{\}
\end{align*}
\]

The largest overlap is between the pair \((\text{tg}_{A,1}, \text{tg}_{B',1})\), consisting of task 1 only. No tasks are reassigned, since \(B'[1] = 1\) is already set to 1. Therefore the distance counter is not incremented. In the third iteration the relevant remaining task groups are:

\[
\begin{align*}
\text{tg}_{A,0} &= \{0\} & \text{tg}_{B'',2} &= \{\} \\
\text{tg}_{A,3} &= \{5\} & \text{tg}_{B'',3} &= \{\}
\end{align*}
\]

All combinations of task groups from A and \(B''\) now have the same intersection (the empty set). At this point we do not know with which pair to proceed, therefore the algorithm is run recursively for each pair: \((\text{tg}_{A,0}, \text{tg}_{B'',2})\) , \((\text{tg}_{A,0}, \text{tg}_{B'',3})\) , \((\text{tg}_{A,3}, \text{tg}_{B'',2})\) , \((\text{tg}_{A,3}, \text{tg}_{B'',3})\) . In this case all 4 recursive branches will find a minimum of 2 additional reassignments. In general one of the shortest recursive branches is selected. In the following we show only the first recursive branch, so we reassign according to pair \((\text{tg}_{A,0}, \text{tg}_{B'',2})\): \(B''' = [2, 1, 0, 0, 0, 0]\) and the distance counter is incremented with one, making the total recorded number of reassignments 2. In the fourth iteration the relevant remaining task groups are:

\[
\begin{align*}
\text{tg}_{A,3} &= \{5\} & \text{tg}_{B'''3} &= \{\}
\end{align*}
\]

The only possible pair is \((\text{tg}_{A,3}, \text{tg}_{B'''3})\). We apply the reassignment rule and with one reassignment we get: \(B'''' = [2, 1, 0, 0, 0, 3]\) . The algorithm is finished and \(B''''\) is now an equivalent mapping to A. The distance counter has reached its final value of 3, which is the guaranteed minimum number of reassignments required to change mapping B into mapping A. The sequence of task group pairs used was:

\((\text{tg}_{A,2}, \text{tg}_{B,0})\), \((\text{tg}_{A,1}, \text{tg}_{B',1})\), \((\text{tg}_{A,0}, \text{tg}_{B'',2})\), \((\text{tg}_{A,3}, \text{tg}_{B'''3})\)

In this example, the following sequence would also have found the minimum:

\((\text{tg}_{A,2}, \text{tg}_{B,0})\), \((\text{tg}_{A,1}, \text{tg}_{B',1})\), \((\text{tg}_{A,3}, \text{tg}_{B'''3})\), \((\text{tg}_{A,0}, \text{tg}_{B''''2})\)
7.7. INITIAL CASE STUDY WITH DISTANCE METRIC

7.6.4 Performance improvement

As mentioned before, the essence of the distance algorithm from the previous section is to find the sequence of task groups pairs that leads to the minimum number of reassignments. It exhaustively tries all permutations of pairs, resulting in an algorithmic worst-case complexity of $O(k!n)$. Using a built-in optimization, it can perform better only when the two input mappings have overlapping task groups that do not overlap by the same number of tasks. Although the algorithmic complexity is fine for small $k$ and $n$, it does not scale well to large problem sizes. Fortunately, it is possible [100] to reduce the distance metric problem to an assignment problem, which can be solved in only $O(n^3)$ using the Munkres assignment algorithm [57].

Munkres works on an assignment-cost matrix, e.g., workers (in columns) perform a job (in rows) for a cost specified in the matrix. The Munkres algorithm then finds the minimal assignment of jobs to workers such that the total cost is minimized. We can define the matrix for two mappings $A$ and $B$ as $m_{ij} = n - |tg_{A,i} \cap tg_{B,j}|$ for each task group $i$ from $A$ and $j$ from $B$. In this way, $m_{ij}$ represents the cost of transforming task group $i$ to task group $j$: the cost is lower when the groups overlap more. Note that $m_{ij}$ is in general not equal to the required number of reassignments, but rather we choose $n$ (number of tasks) so that all $m_{ij} \geq 0$. The outcome of the Munkres algorithm is a list of pairs of task groups such that the cost is minimal. By applying the list of pair groups as reassignments to $B$, we can transform $B$ into $A$ and obtain the distance value. An example of reassignment according to a list of task groups is given in Section 7.8.2.

7.7 Initial case study with distance metric

In Section 7.4 we mentioned that the balance between diversity and convergence in a GA is delicate and that both properties are important for a successful GA. In the previous section the distance metric was proposed as a way to discriminate equivalent and similar designs. Here, we propose a few simple methods to incorporate the distance metric into the GA in order to manage (increase) population diversity in the hope that this will help the GA to sweep a broader range in the design space. We perform the same evaluation as in Section 7.4 using the same design problem.

We have previously seen that our GA tends to converge towards a very homogeneous population: in some runs we saw that the final generations hardly introduce any new chromosomes. This means that at that point, the GA has converged prematurely to a local optimum, or it has simply already achieved its best possible result. Here we test what happens when we artificially introduce more diversity in the population of each GA iteration. For this purpose we analyze in each GA generation, the clustering of elements in terms of the distance metric. For each individual we calculate the average distance towards the other individuals; if the average distance is low, then we consider the individual to be very clustered. In the first of the two methods introduced here, we select a few of the most clustered individuals in the GA population for replacement by new, randomly initialized individuals. The rationale
is that the insertion of random genetic material will help the GA to continuously consider alternatives other than the solutions on which it is converging (thus increasing diversity). This is in many ways similar to the function of the GA mutation operator, except that we target specific individuals that will be completely replaced, whereas mutation randomly modifies only small parts of the chromosome (genes). In order not to override completely the natural convergence of the GA, we first replace only a percentage of population individuals. The result is shown in the left part of Figure 7.14, the line indicated with replacement 0 is the normal GA (without any replacements) as a reference. The other lines represent GAs where 5, 10, 20 or 40 percent of most clustered part of the population gets replaced. We see that none of these GAs is able to perform better than the reference GA, although, up to 20% can be replaced without much harm to the GA (40% however clearly reduces GA performance). The first of the three numbers indicates the cumulative diversity of the population (excluding initial population). It shows that a higher replacement rate does increase diversity, but clearly this does not help to improve overall GA performance. In a follow-up experiment we replace a dynamically changing proportion of the population instead of a statically fixed percentage. Here we increase the rate of replacement with 1 per generation, e.g., 1 in the first generation, 2 in the second generation, etc. However, we vary the parameter \( I \), which indicates how many generations proceed normally before the first replacement starts. So one replacement occurs in generation \( I \), two replacements in generation \( I + 1 \), etc. In our experiment we have a maximum of 15 generations, so we choose \( I = 1, 4, 9 \) and \( \text{inf} \), where \( \text{inf} \) indicates the normal GA (no replacements). The right side of Figure 7.14 shows clearly that \( I = 1 \) gives the worst results. This is perhaps not surprising since \( I = 1 \) replaces a number of individuals in the population equal to the generation number: in the last population of the GA, all elements will be replaced by random values. Apparently, this interferes too much with the normal working of the GA. Different values for \( I \) seem to do slightly better: the best result being obtained by \( I = 9 \), which starts replacing individuals in the ninth generation. However the confidence intervals show that the result of \( I = 9 \) is not significantly better than the normal GA (\( I = \text{inf} \)).

![Graph showing the performance of GAs with different replacement strategies](image)

Figure 7.14: Replacing most clustered elements (static (l) and dynamic (r) proportion)

In a follow-up experiment we replace a dynamically changing proportion of the population instead of a statically fixed percentage. Here we increase the rate of replacement with
7.7. INITIAL CASE STUDY WITH DISTANCE METRIC

1 per generation, e.g., 1 in the first generation, 2 in the second generation, etc. However, we vary the parameter $I$, which indicates how many generations proceed normally before the first replacement starts. So one replacement occurs in generation $I$, two replacements in generation $I + 1$, etc. In our experiment we have a maximum of 15 generations, so we choose $I = 1, 4, 9$ and inf, where inf indicates the normal GA (no replacements). The right side of Figure 7.14 shows clearly that $I = 1$ gives the worst results. This is perhaps not surprising since $I = 1$ replaces a number of individuals in the population equal to the generation number; in the last population of the GA, all elements will be replaced by random values. Apparently, this interferes too much with the normal working of the GA. Different values for $I$ seem to do slightly better: the best result being obtained by $I = 9$, which starts replacing individuals in the ninth generation. However the confidence intervals show that the result of $I = 9$ is not significantly better than the normal GA ($I = \text{inf}$).

![Figure 7.15: Replacing least clustered elements (static (l) and dynamic (r) proportion)](image)

Perhaps we have overlooked a possible risk in the proposed strategy: it can be expected that the most clustered elements are often also the best (fittest) elements in a given population, since the cluster is the convergence “focal point” of the GA. Moreover, these fit elements are more likely to contain fragments of genetic material that contribute to the final optimal solution. So by replacing the most clustered elements, we may well be disrupting the natural GA convergence. Therefore, we now test what happens if we replace the least clustered elements: the results are shown in Figure 7.15 for both the static and dynamic replacement schemes. We see that the results for various $I$ are very close to, but generally worse than the standard GA and that the confidence intervals are mostly overlapping, so we have to conclude again that the new strategy has no benefit.

Finally, we analyze how much diversity is actually introduced by the different methods presented in this section. The legend in each of the four plots shows three numbers measuring diversity. The first two numbers measure the average cumulative number of unique chromosomes (according to the distance metric) in the final generation. This is equivalent to the total number of unique individuals during the run of the GA. The first number measures diversity after application of the crossover and mutation operators, whereas the second number measures before. The third number is the product of number of generations and population size (the random-search baseline is calculated using this number, as we
saw in Section 7.5). Theoretically, the static method at 20% replacement introduces up-to $15 \cdot (40 \cdot 0.2) = 120$ new individuals which is the same as the dynamic method for $I = 1$: $1 + 2 + \ldots + 15 = 120$. For example, in Figure 7.15, comparing the second diversity number between 0% and 20%, we indeed see (approximately) the expected diversity increase (127 vs. 245). If we compare the first diversity number for the static and dynamic experiments, then we see that the static method has a higher diversity (142) than the static method (124). A similar observation can be made for the experiments in Figure 7.14. Apparently the dynamically increasing rate of replacement has a smaller effect on diversity than the dynamic method. Moreover, we observe the difference between the first and the second diversity number. Many of the new individuals that are inserted at the start of a generation (the second number) have disappeared by the end of the generation (the first number). The difference can be explained by the fact that randomly inserted individuals are lost from the population after selection, crossover and mutation. This suggests that the randomly inserted individuals are – on average – of insufficient quality to survive to the next generation.

We conclude that the methods presented in this section are able to increase diversity in the GA populations. However this is clearly shown to have no positive impact on the overall performance of the GA. In the next section, a different approach proves to be more successful.

### 7.8 GA with integrated distance metric

As we have seen in Section 7.7, the methods to control population diversity based on the distance metric did not contribute to improving GA performance. It is possible that the problem with those methods was cause by the fact that they were artificially layered on top of the normal GA operation. After each generation (or similarly: at the beginning of the next GA generation), the population would be artificially changed. Although this increased overall diversity in the populations, there was no measurable and predictable improvement in the GA performance. Another disadvantage of the proposed GA-extensions was the computational overhead of computing the average cluster distance in each generation, which required $n^2$ distance calculations for population size $n$. Therefore, scalability would be a problem for larger population sizes. To avoid such problems, we continue the search for generic, scalable improvements of the standard GA solutions in this section. However, we now propose more integrated approaches based on our previous observations about the design space in the context of the distance metric. This has resulted in the definition of new GA-operators for crossover and mutation. These new GA-operators attempt to optimize GA performance either by

- reducing the redundancy present in the chromosome representation
- using a distance metric-based crossover

Combinations of these are also possible and we will show and analyze the experimental results in a way similar to the previous sections. In this case, however, we scale up the
7.8. GA WITH INTEGRATED DISTANCE METRIC

reference design problem to a size that is no longer practical for exhaustive evaluation, but which is more realistic for practical design problems.

7.8.1 Reducing representation redundancy

A chromosome representation of a design point with \( k \) processors can be represented in \( k! \) different ways by permutating the \( k \) processor labels. This is sometimes referred to as the “symmetry” of the search space ([100]). Search algorithms may be affected by symmetry in the search space, and is known to have a negative impact on the performance of genetic algorithms in some problem domains, such as for example in the case of the graph coloring problem ([106]). We investigate whether the same is true for the search spaces of our design problem. For this purpose we propose a set of genetic operators that enable the GA to traverse the design space without symmetry. Intuitively, removing the symmetry from the design space should result in a more efficient search, as it effectively makes the design space smaller. But it may equally be the case that the GA, which is optimized for combinatorial problems, searches through the symmetrical subspaces with ease. Therefore, whether symmetry is a limiting factor on performance is yet to be determined.

We observe that for our chosen chromosome representation, it is easy to convert a set of chromosomes to equivalent chromosomes with a representation from the same, symmetry subspace. Note that the Nijhuis-Wilf algorithm (that we used in Section 7.4) to exhaustively generate all design points) produces all design points in base-symmetry form. Following the naming convention from Section 7.6.2, it holds for each base-symmetry chromosome representation \( A \):

\[
\begin{align*}
A[0] &= 0 \\
A[i + 1] &\leq \max(A[0], \ldots, A[i]) + 1
\end{align*}
\]

In the work of [106] the assignment function represented by \( A \) is called a Restricted Growth Function (RGF): from left-to-right (starting with 0), the mapping target is identified using the lowest possible number. This leads to a simple (order \( O(N) \)) re-assignment function to change a mapping to its unique equivalent in the base-symmetry space. This is shown in Algorithm 1; \( base \) is an array initialized to \(-1\) for all elements, \( A \) is the design point to be re-written.

We now use the baseform function to enforce that subspace boundaries are not crossed during the normal operation of the GA. The simplest way to implement this is to append the baseform function to each normal GA operator. So, for example, a normal 2-point crossover could transform parent chromosomes \( A \) and \( B \) in child chromosomes \( A' \) and \( B' \). After a subsequent application of the baseform function to both chromosomes, the result is \( A'' \) and \( B'' \).
Algorithm 1 The baseform function

\begin{verbatim}
cnt ← 0
base ← [−1, ..., −1]
for i = 0 \(\rightarrow\) (n − 1) do
  idx ← A[i]
  if base[idx] < 0 then
    base[idx] ← cnt
    cnt ← cnt + 1
  end if
  A[i] ← base[idx]
i ← i + 1
end for
\end{verbatim}

before crossover: [0,0,0,0,0,1,2]=A
[0,1,1,1,2,2,2]=B
after crossover: [0,0,0,0,2,2,2]=A′
[0,1,1,1,0,0,1,2]=B′
after baseline: [0,0,0,0,1,1,1]=B″
[0,1,1,1,0,0,1,2]=B″

The mutation function is similarly appended with the baseform function. With the extended crossover and mutation operators, all chromosomes in each generated population are guaranteed to remain in the base-symmetry space. Note that it is possible, but not necessary to apply the baseform function to the initial random population, since all individuals would automatically be transformed to baseform notation after the first GA iteration.

7.8.2 A distance-metric based cross-over operator

Our initial motivation to introduce the distance metric was to provide a measure of similarity between design points. We saw in Section 7.6.2 that the distance metric can relate any two design points by finding a minimal set of atomic changes transforming design point \(A\) into design point \(B\). While the number of changes can be used to measure similarity, the resulting set of changes can also be used to provide some much-needed structure in the complex mapping design space. For this purpose we define a sequence of intermediate design points that is the result of applying one such minimal set of atomic changes to \(A\) (in unspecified order). The sequence of intermediate design points represents a “path” from \(A\) to \(B\):

\[ A, A^1, A^2, \ldots, A^{n-1}, B \] where \(n = \text{distance}(A, B)\)

The intermediate design points \(A^i\) will share a varying amount of characteristics from both \(A\) and \(B\), since every application of an atomic change helps to transform \(A\) into \(B\). We note that exchanging properties from parent chromosomes is the main purpose of the GA crossover operator. Therefore, we propose to use the constructed path as the basis for a new type of crossover operator: the distance-path crossover.
7.8. GA WITH INTEGRATED DISTANCE METRIC

Next we show how such a path can be constructed and how a crossover operator can be implemented. We already determined that the distance metric problem is equivalent to a group assignment problem: and indeed both distance algorithms from Sections 7.6.2 and 7.6.4 can provide us with both the resulting distance as well as (one of) the minimal group assignments. The minimal group assignments can be used to build the path between $A$ and $B$ by iteratively applying the re-assignment function (Section 7.6.2) for each group.

For example, the mappings $A$ and $B$:

$$A = [0, 1, 1, 0, 2, 3]$$

$$B = [0, 1, 2, 2, 1, 2]$$

have a distance 3 with matching groups: $(1^A, 1^B)$, $(2^A, 4^B)$ and $(3^A, 2^B)$ (using the notation from Section 7.6.3). We rewrite $A$ into $B$ by applying the reassignment function to each of the group pairs in order:

- $(1^A, 1^B)$: on every $i$ where $B[i] = 1$ assign 1 to $A[i]$,
  - $i = 4$, result: $[0, 1, 1, 0, 1, 3] = A'$

- $(2^A, 4^B)$: on every $i$ where $B[i] = 4$ assign 2 to $A[i]$,
  - no such $i$

- $(3^A, 2^B)$: on every $i$ where $B[i] = 2$ assign 3 to $A[i]$
  - $i = 2$, result: $[0, 1, 3, 0, 1, 3] = A''$
  - $i = 3$, result: $[0, 1, 3, 3, 1, 3] = A'''$
  - $[0, 1, 2, 2, 1, 2] = B$

The path from $A$ to $B$ is given in the right column, note that $A'''$ is symmetrical to $B$. Paths will be longer when $A$ and $B$ are less similar and when the problem space (and thus the chromosomes) are longer.

We propose that the new crossover operator creates offspring by simply selecting two random elements from the path between parents $A$ and $B$. Special provisions can easily be made if one objects against the fact that the children may be the same as each other or as one of their parents. Finally we note that the offspring created by this crossover mechanism only mixes genetic material from the parents, and that no new or random material is inserted. In other words, properties from an element $C$ that is not on the path will not occur in the offspring, since the distance metric is guaranteed to find the minimal path between $A$ and $B$.

7.8.3 Combination of approaches

The baseform and crossover techniques that were presented in the previous sections can also be used in combination. In that case we first perform the baseform method after the modified crossover operator. In the example of the previous section this would mean that if $A''$ was the result of the crossover, then application of the baseform function would convert it to $[0, 1, 2, 0, 1, 2]$. We note that it is possible to delay the baseform function until after the mutation operator, so that the baseform conversion needs to be performed only once.
Finally we summarize in Figure 7.16 that the proposed new approaches give rise to 3 new GA methods. These will be evaluated in the next section.

### 7.8.4 Experiments

A set of experiments has been performed to determine the impact of the two approaches that were described previously. We look both at the impact of the crossover and baseform extensions separately as well in the combined approach. As the basis for comparison we use a GA with tournament selection and uniform crossover. To challenge the search algorithm, we use a different, and (compared to the previous experiments) much larger stochastic application model consisting of 20 processes. They are mapped onto a homogeneous 8-processor architecture using a single shared bus for communication. Note that this significantly increases the design space to approximately $3\cdot10^{13}$ unique design points.

#### Part 1

This results in a series of P-Q plots (Figure 7.17 and Figure 7.18) that show the results of the four types of GA: regular (reference) GA, GA with baseform, GA with new crossover and finally a GA with baseform and new crossover. In the following we will refer to these simply as: reference, baseform, crossover and combined GA. In the first experiment (Figure 7.17), we test three different mutation rates (top-to-bottom: 0.15, 0.1 and 0.05) as well as for two different population sizes (left: 40, right: 80). In all other aspects, the GAs use the same parameters (e.g., crossover rate 0.9) and they all run for 30 generations.

Note that in contrast to previous P-Q plots, the quality-axis of the top figure is given not as a relative percentile, but as absolute values in the objective domain (simulated cycles). This change is necessary as we can not determine the true optimum anymore. The design space (20 processes onto 8 processors) is now sufficiently large that exhaustive search becomes impractical. The P-Q graphs are now effectively cumulative distribution functions of
7.8. GA WITH INTEGRATED DISTANCE METRIC

Figure 7.17: P-Q plots for GAs with combined extensions
the experimental data (known as the empirical CDF). They can be interpreted the same as the previous P-Q plots: dominating lines (towards the top-left) show a better performing GA. The confidence intervals at the bottom of each graph are the same as before. Also note that an initial exploration (not reported here) showed that for the new design problem, tournament selection with uniform crossover works best. Therefore, in the following experiments we use this as the reference GA (denoted with the label \texttt{tournamentUniform} or simply \texttt{reference}).

The first subplot (Figure 7.17a) shows that the extended GAs perform better than the reference GA. The difference between the average result (as indicated by the center of the confidence interval) of reference GA and the best performing GA (combined method) is quite large: 2470 vs. 2373 in terms of absolute performance. Moreover, the confidence intervals between the reference and combined GA are clearly disjoint, confirming the reliability of our observation. As we do not have access to the global optimum of this design space, it is not possible to quantify exactly the meaning of a reduction of approximately 100 units (cycles) in objective space. However, we can see that the best result that we found in all experiments is 2331 when using population size 80 (Figure 7.17d), which is approximately 40 units lower than our best average result in Figure 7.17a. So, as a very rough, intuitive comparison we say that a reduction of 100 is 2.5 times the improvement of doubling the population size. This is significant, considering that increasing the population size increases the number of evaluations and thereby the search cost. In relative terms we can compare the reference and the combined GA in Figure 7.17a: the probability of finding results within the range 2300-2475 differs around a factor 2 or 3.

In the same figure (7.17a) we see that the results of using only the baseform or only the path-crossover are also much better than the reference GA. The baseform-only GA seems to perform slightly worse than using only the path-crossover for mutation rate 0.15 (Figures 7.17a and 7.17b). The confidence interval plot of 7.17a clearly shows the order in performance (from high to low): the combined method, path-crossover, baseform and lastly the reference GA. The experiment with a larger population (Figure 7.17b) shows the same ordering, although the difference between path-crossover and baseform are less pronounced.

In the experiments with lower mutation rate (7.17c to 7.17f), the difference in performance between all GA methods becomes smaller. For mutation rate 0.10 (Figure 7.17c and 7.17d), the performance of the extended GA methods start to become very similar (see the confidence plots), but there is still a significant difference with the reference GA. Only for the lowest mutation rate (Figure 7.17e and 7.17f) does the reference GA catch up. For population size 40 the reference GA even performs a little better than the other methods, though not so for the larger population size of 80 (Figure 7.17f).

So we tentatively conclude that in almost all cases the extended GAs perform better, or at worst similar, to the reference GA. However, the impact seems to be inversely proportional to the mutation rate: a higher mutation rate means a larger performance difference. Therefore, a designer can use the baseform and crossover extensions with confidence, as long as he chooses a sufficiently high mutation rate ($\geq 0.10$). Moreover, the baseform, path-crossover and combined methods introduce only a small bit of computational overhead to
7.8. GA WITH INTEGRATED DISTANCE METRIC

Figure 7.18: P-Q plots for metric-based GA vs. regular GA
CHAPTER 7. SUPPORT FOR AUTOMATIC DSE

the standard GA: specifically they are much better scalable than the methods proposed in Section 7.7. From these experiments we conclude that using a sufficiently high mutation rate, the proposed GA extensions perform the same, if not better in all cases. For higher mutation rates or bigger populations, the combined method that combines the baseform and path-crossover, performs best.

Part 2

In a follow-up experiment we check the impact of selective pressure on the performance of the GAs. In Figure 7.18 the PQ-plots are shown for a series of experiments where we vary the pressure (top-to-bottom: low, medium and high) and different mutation rates (left:0.10, 0.15 and 0.20, right:0.01 and 0.05). Population size is 40 in all experiments. Since the combined method performed consistently well in the last experiment, each plot now only compares the combined method with the reference GA. If we compare the graphs on the left with those on the right, we can immediately see that for all different selective pressure rates the combined GA methods perform significantly worse with a low mutation rate: mutation rate 0.01 has the worst performance. Mutation rate 0.05 performs better, but still worse for low pressure (Fig. 7.18b). In the case of medium and high pressure and mutation rate 0.05 (Fig. 7.18d and 7.18f), the combined and reference methods perform practically the same.

The situation in the left figures (with higher mutation rates) is very different; now the reference GA with mutation rate 0.10 is always one of the best performing methods. There is a noticeable trend related to the pressure indicating that a higher pressure increases the difference between the combined method and the reference GA. In all of the three left-hand graphs with mutation rates ≥ 0.10, the best performing reference GA is the one with mutation rate 0.10. In case of low pressure (Figure 7.18a), the combined method performs only slightly better than the reference GA with mutation rate 0.10. However, when we increase the pressure, then the distance between the two becomes much larger. For example, in case of medium pressure (Figure 7.18c), the combined method GA with mutation rate 0.15 has an average result of 2376 and the reference GA 2410. And with high pressure (Figure 7.18e), the combined method performs approximately the same, but the reference GA performs even worse, thus increasing the difference. A further observation is that when we increase pressure, the difference between the three extended GAs becomes smaller: for medium and high pressure, the results for mutation rate 0.10 and 0.15 are overlapping and the result for mutation rate 0.20 is closing in. The difference between the three reference GAs, however, seems to be constant from low to high pressure.

We conclude from these experiments that the performance of the combined GA works best for higher pressure and a mutation rate of 0.10 or 0.15. Where the combined GA method seems to benefit from higher pressure and mutation rates, the opposite is true for the reference GA. In fact, the best result with the reference GA is obtained with mutation rate 0.05 and low pressure (Figure 7.18b). This is in fact the only time that the standard GA is able to obtain a better average result (for the same pressure) than the combined GA: an average value of 2372 (Fig. 7.18b versus 2404 for the combined GA (Fig. 7.18a). However, for the medium and high pressure cases, the combined GA method always results in a better
average result, for both mutation rate 0.10 and 0.15. These mutation rates are consistent with the previous set of experiments that also identified 0.10 and 0.15 to be much better mutation rates for the combined GA than a mutation rate $< 0.10$. In the experiments of Part 1, we also saw that the extended GA types performed better compared to the reference GA when the population size was bigger. To verify that this is a consistent trend, we re-do a sample of the experiments (the ones shown in Figure 7.18c and 7.18d) for a population size of 80. The results in Figure 7.19 immediately show the increasing difference between the combined GAs and the reference GAs. In Figure 7.19a, the combined GAs are clustered to the left, and the confidence intervals show a large gap between the results of even the worst combined GA (with mutation rate 0.20) and the best reference GA (mutation rate 0.10). In Figure 7.19b we see the results for lower mutation rates and as we saw before, only the very low mutation rate of 0.01 performs badly for the combined GA. The combined method with mutation rate 0.05 (7.19b) already performs similarly to the reference GA (their confidence intervals overlapping). Moreover, even the best performing reference GA (mutation rate 0.05) only gets an average result of 2363 (7.19b), whereas all of the combined GAs in Fig. 7.19a obtain a better average result. This shows that the combined GA performs the same or better than the reference GA for a range of mutation rates 0.05-0.20. In comparison, the reference GA is only working well in the range 0.01-0.05 and even then, its results do not reach the optima that the combined method GA finds.

![Graphs showing mutation rate performance](image)

**Figure 7.19:** P-Q plots for metric-based GA vs. regular GA; population size 80

## 7.9 Conclusion

In this chapter we have investigated a DSE approach using genetic algorithms as the basis for automatically traversing the design space. A metric was introduced that can quantify the similarity between design points. Originally the metric was intended as a method to control population diversity and to apply it to the GA in order to improve GA performance. In Section 7.7 we have shown that it is indeed possible to increase population diversity by
applying a few simple, distance metric-based methods. However, these methods proved unable to actually increase GA performance. We speculate that they do not work for the reason that by replacing individuals in the GA population, they interfere too abruptly with the natural convergence process of the GA.

Therefore, in Section 7.8, we introduced two new metric-based extensions to the GA that are scalable and fit more naturally to the GA process. The combination of the two new methods results in a third, combined method, which in the experiments often showed the best performance. In the presented experimental results, the GAs with the proposed extensions perform at least as well, but often better than the reference GA. Important is the finding that we could identify a clear trend to show for which parameters the extended GA methods performed better. Once more of such trends are identified and verified, a system designer can more accurately choose a search method to fit his design problem. In particular we showed that the extended GA methods benefit from high mutation and high selective pressure. We hypothesize that the higher mutation rate keeps population diversity high (as we saw in the experiments earlier in this chapter), while the high selective pressure improves convergence to the optimum result, but more research is required to verify this. Also, we observed that the extended GA methods work better for larger population sizes. We consider this to be a desirable property, since for more complex design spaces, population sizes are usually increased to exploit more parallel search within the GA. Finally, we observed that the extended GA methods seem to be effective for a wider range of GA parameters than the regular, reference GA. The former only seemed to perform well for low mutation rates and low pressure, whereas the extended methods performed better in all other situations. We are hopeful that such knowledge can reduce the effort that is required to find the right combination of GA parameters, but more research in this area is required.

7.10 Future work

There are many interesting directions that could be explored as a logical follow-up to the contributions and case studies presented in this chapter and in general for the field of automated DSE for system-level design. First of all, it would be worth to see whether the proposed distance metric can be useful towards exploration of a wider range of system models and parameters. For example, the distance metric could be extended to heterogeneous systems by adding extra weight to the metric if a task is mapped onto a different type of processor. This metric extension would work the same for homogeneous processor sub-groups, but increases the distance value for each task mapped onto different types of resources (even if—according to the task partitioning— it is in the same task-group). Also, we intend to investigate the performance of the distance metric approach when multiple objectives are taken into account, for example considering power and cost in addition to system performance. Future experiments with the mapping distance metric could scale to even larger design spaces, e.g., mapping application workloads with hundreds of processors. In order to reduce the overhead of the simulator as the fitness function in the GA, we could also look for ways to estimate the performance based on previously evaluated design points and then in the GA we
inter leaf simulation-based evaluation with estimation. This would introduce the possibility to trade-off a lower DSE accuracy for higher DSE method execution times.

We could also explore whether a distance-like metric could be useful for design space dimensions that are not directly mapping-related. Consider for example memory or buffer size of a particular system component for which a range of (discrete integer) values need to be explored. The memory size parameter can easily be encoded in the gene, but then we can ask how to apply the distance metric to it. It may even be possible to apply the distance metric when the design point is not based on a meta-platform mapping at all, but rather for example the generator-approach as discussed in Section 3.4. In any case we recommend to keep in mind that new approaches may work in unexpected ways: in our case the distance metric was originally developed to manage diversity, but turned out to perform better as a construct to build a new type of crossover operator.

Also, recent advances in the general field of evolutionary computation may be very suitable for application in our DSE domain. We specifically note the class of niching genetic algorithms that recognizes the problem of having multiple optima of equal interest that reside in different segments of the design space. Niching GAs attempt to converge the GA population simultaneously towards multiple optima, instead of just one as with regular GAs. Interestingly, niching GAs require a distance metric to be defined on the parameter search space. The distance metric that was defined in this chapter for measuring similarity between mapping specifications could be suitable for implementing niching GAs, but further research is required.

As we have seen in our work, the GA method (although generally efficient) suffers from the problem of having itself quite a lot of parameters that effect its efficiency in complex ways. The selection of parameters is often so complex that it sometimes seems that GAs only replace the original problem by the problem of finding the right GA parameters. An interesting experiment would be to optimize the choice of parameters for a GA (which is solving the original problem) by a means of another GA. Such a recursive use of GA optimization would however be highly inefficient for DSE case studies where the computational bottleneck is by caused by the fitness function (for example if fitness is determined by a (system-level) simulator). However, it could result in new knowledge about GA parameter influence. Finally, it could be worth to study methods for automatically selecting the best GA parameters (such as was done for simulated annealing methods in [77]).

As a final point we emphasize the problem of the large diversity of evaluation and comparison methods that are currently being used in the context of research in the area of DSE for system-level design. There are likely many valuable contributions that improve DSE in one way or another, but without a common framework or methodology, their individual merits are only applied to a limited sub-domain of DSE problems. Moreover, this fragmentation in the research field may well hinder future advances, as it is not a big stretch of the imagination to expect that breakthrough approaches will come from the combination and evolution of existing methods. A more collectively orchestrated approach in the research towards DSE search problems may also provide a solution here. At the core could be a standardized benchmark of industrially-relevant real and synthetic design problems as an analog
to application benchmarks that have a longstanding tradition to provide a common performance evaluation platform for various computer architectures. It could be worth investing in the generation of large, pre-computed design spaces (relating parameter to objective values), so that DSE methods can be tested without the computational overhead of evaluating single design points (by simulation or otherwise).