Contents lists available at SciVerse ScienceDirect

Journal of Systems Architecture

journal homepage: www.elsevier.com/locate/sysarc

Exploiting domain knowledge in system-level MPSoC design space exploration

Mark Thompson, Andy D. Pimentel *

Informatics Institute, University of Amsterdam, Science Park 904, 1098 XH Amsterdam, The Netherlands

ARTICLE INFO

Article history:
Received 7 December 2012
Received in revised form 16 April 2013
Accepted 26 May 2013
Available online 7 June 2013

Keywords:
System-level design space exploration
Design space pruning
Genetic algorithms
MPSoC design

ABSTRACT

System-level design space exploration (DSE), which is performed early in the design process, is of eminent importance to the design of complex multi-processor embedded multimedia systems. During system-level DSE, system parameters like, e.g., the number and type of processors, and the mapping of application tasks to architectural resources, are considered. The number of design instances that need to be evaluated during such DSE to find good design candidates is large, making the DSE process time consuming. Therefore, pruning techniques are needed to optimize the DSE process, allowing the DSE search algorithms to either find the design candidates quicker or to spend the same amount of time to evaluate more design points and thus improve the chance of finding even better candidates. In this article, we study several novel approaches that exploit domain knowledge to optimize the DSE search process. To this end, we focus on DSE techniques based on genetic algorithms (GA) and introduce two new extensions to a GA to optimize its search behavior. Experimental results demonstrate that the extended GAs perform at least as well, but typically significantly better than a reference (non-optimized) GA.

1. Introduction

Advances in chip technology according to Moore's Law, allowing more and more functionality to be integrated on a single chip, have led to the emergence of Multi-Processor Systems on Chip (MPSoCs). Nowadays, these MPSoCs are key to the development of advanced embedded multimedia systems, such as set-top boxes, digital televisions, and 3G/4G smart phones. Designers of MPSoC-based embedded systems are typically faced with conflicting design requirements regarding performance, flexibility, power consumption, and cost. As a result, MPSoC-based embedded systems often have heterogeneous system architectures, consisting of components that range from fully programmable processor cores to fully dedicated hardware blocks. Programmable processor technology is used for realizing flexibility (to support, e.g., multiple applications and future extensions), while dedicated hardware is used to optimize designs in time-critical areas and for power and cost minimization.

The increasing design complexity of these MPSoC-based embedded systems has led to the emergence of system-level design. A key ingredient of system-level design is the notion of high-level modeling and simulation in which the models allow for capturing the behavior of system components and their interactions at a high level of abstraction. As these high-level models minimize the modeling effort and are optimized for execution speed, they can be applied at the early design stages to perform, for example, design space exploration (DSE). Such early DSE is of eminent importance as early design choices heavily influence the success or failure of the final product.

Current DSE efforts typically use simulation or analytical models to evaluate single design points together with a heuristic search method to search the design space. These DSE methods usually search the design space using only a finite number of design-point evaluations, not guaranteeing to find the absolute optimum in the design space, but they 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. However, the number of design points that need to be evaluated to find these design candidates is still large, making the DSE process time consuming. Therefore, it is important to develop pruning techniques that can further optimize the DSE process, allowing the DSE search algorithms to either find the design candidates quicker or to spend the same amount of time to evaluate more design points. The latter can be used to enable the search of larger design spaces or to improve the chance of finding better design candidates.

In this article, we study several novel approaches that exploit domain knowledge to optimize the DSE search process. To this end, we focus on DSE techniques based on genetic algorithms (GA) [2,4] and introduce two new extensions to a GA to optimize its search behavior. One extension aims at reducing the redundancy present in chromosome representations of a GA, while the

* Corresponding author. Tel.: +31 205257578.
E-mail address: a.d.pimentel@uva.nl (A.D. Pimentel).

other extension introduces a new crossover operator based on a so-called mapping distance metric. In addition, we have also investigated the combination of the two extensions. Using a range of experiments, we demonstrate that the extended GAs perform at least as well, but typically significantly better than a reference (non-optimized) GA.

The remainder of this article is organized as follows. In the next section, we provide more background on system-level DSE of MPSoCs and describe a number of prerequisites for our work. Section 3 presents the two extensions to a GA, exploiting domain knowledge, to optimize its search behavior. In Section 4, we evaluate these GA extensions by comparing their results to those from a non-optimized reference GA. Section 5 describes related work, after which Section 6 concludes the article.

2. System-level DSE of MPSoCs

In this article, we focus on *application mapping DSE* for MPSoCs, where mapping involves two aspects: (1) allocation and (2) binding. Allocation deals with selecting the architectural components in the MPSoC platform architecture that will be involved in the execution of application workloads. Here, it is important to note that the allocation process also allows for selecting between different configurations of a certain type of architectural component, like, e.g., a specific processor type with different cache configurations. Subsequently, the binding specifies which application task or application communication is performed by which MPSoC component.

The process of mapping DSE for MPSoCs is comprised of two core ingredients [9]: (1) a method to evaluate a single design point (i.e., mapping) and, (2) a method to efficiently navigate the mapping design space. For the evaluation of single design points, we deploy the Sesame system-level simulation framework [19,7]. Sesame is targeted at the evaluation of (multimedia) MPSoCs and allows for rapid performance assessment of different MPSoC architecture designs, application to architecture mappings, and hardware/software partitionings. Given that design spaces grow exponentially in the number of parameters, exhaustive search (evaluating every possible design point) is infeasible. In this article, we therefore look at methods to navigate the design space that are based on genetic algorithms (GA). GA-based DSE has been widely studied in the domain of system-level design (e.g., [9,6,21]) and has been demonstrated to yield good results.

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* [2]. A (randomly initialized) population of these chromosomes will be iteratively modified by performing a fixed sequence of actions that are inspired by their counterparts from biology: evaluation and selection, crossover and mutation. A fundamental design choice of a GA is the genetic representation of the solution space, because each of the selection, crossover and mutation steps depends on it. In our case, the problem is finding an optimal design candidate in a large space of possible design candidates that can be evaluated with Sesame. More specifically, our design space consists of parameters that are related to the application-to-architecture mapping. As a convenient mapping description for an application with n tasks, we use a vector of size n with processor identifiers p_i, where p_i indicates the mapping target of task i:

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

This commonly used description is very suitable to serve as the chromosome representation (or genotype) for a GA. A valid mapping specification is a partitioning of all n tasks. Note that partitions may be empty (processor not in use) or contain all n tasks (a single processor system). A processor that is not assigned any tasks (having an empty task partition) can be considered idle or non-existent. Here, 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). Moreover, we assume that each pair of processors can communicate so that there are no topological communication restrictions. The result is that any task can be mapped onto any processor so that we do not have to make special provisions for repairing infeasible mappings. This means that implementing crossover and mutation operators in the GA will be relatively easy, since any recombination of such chromosomes always results in a valid design-point specification. We note, however, that the methods presented in this article can also be applied to GA-based DSE approaches that do generate infeasible mappings (and thus need repair mechanisms for this).

Moreover, in this article, we focus on homogeneous architectures or, alternatively, on the set(s) of homogeneous processing elements that are often present in heterogeneous MPSoCs. So, this means that we assume that all processors are the same and a task incurs the same delays 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, like multiprocessor systems in which all processors are connected to a central bus with a non-prioritizing arbiter. Addressing system heterogeneity is considered as future work.

As mentioned before, although GA-based DSE methods search the design space only using a finite number of design-point evaluations, the number of design points that need to be evaluated to find good design candidates is still large, making the DSE process time consuming. Therefore, pruning techniques are needed to further optimize the DSE process. In the next section, we present two of such pruning techniques by exploiting domain knowledge in a GA.

3. Pruning by exploiting domain knowledge

3.1. Motivation

As described in the previous section, our DSE addresses the mapping of application tasks onto architectural resources. The performance of a single design point (i.e., mapping) 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 believe 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, after which we mutate each chromosome in x positions by randomly choosing the positions *and* its mutation value. The results are shown in Fig. 1 for a design problem where an application with 20 tasks is mapped onto a homogeneous, symmetrical architecture with eight 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

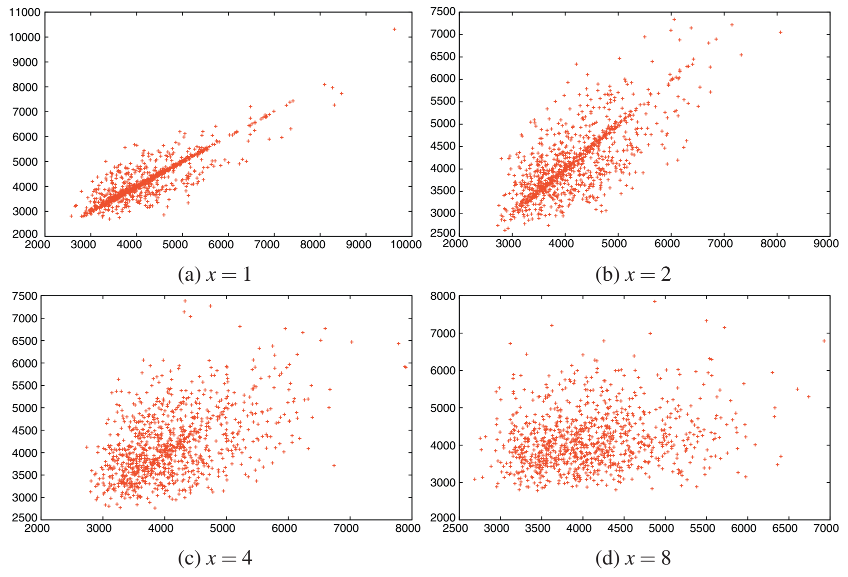


Fig. 1. Correlation between results with mutation in x gene positions.

and disappears with changes in 4 or 8 positions (bottom graphs). This is a clear indication that, at least for homogeneous systems/system parts, small changes in the design point specification typically lead to modest performance differences. In this article, we propose methods to exploit this correlation by integrating such domain knowledge into our GA-based DSE methods. To exploit domain knowledge for the purpose of design space pruning, we propose new GA-operators for crossover and mutation. These new GA-operators attempt to optimize search performance either by (1) reducing the redundancy present in chromosome representations, or (2) using a new and improved crossover operator that is based on a *mapping distance metric*. Combinations of these are also possible and we will show and analyze the results of these methods in a range of experiments.

3.2. Reducing representation redundancy

A chromosome representation of a design point with k homogeneous 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 [20]. For example, in our case, the two mapping vectors [0,0,1,1] and [1,1,0,0] would refer to symmetrical (and thus equivalent) design points. It is known that in some problem domains this symmetry in the search space can negatively affect the performance of search algorithms like genetic algorithms [23]. We therefore need to 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 search 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. We follow a naming convention where the vector A is a mapping where $A[i]$ denotes the processor number onto which application task i is mapped. Then, it holds for each base-symmetry chromosome representation A:

$$A[0] = 0$$
$$A[i+1] \leqslant \max(A[0], \ldots, A[i]) + 1$$

In the work of [23], 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, and A is the design point to be re-written.

Algorithm 1. The *baseform* function

```
$cnt \leftarrow 0$
$base \leftarrow [-1, \ldots, -1]$
**for** $i = 0 \rightarrow (n-1)$ **do**
$idx \leftarrow A[i]$
**if** $base[idx] < 0$ **then**
$base[idx] \leftarrow cnt$
$cnt \leftarrow cnt + 1$
**end if**
$A[i] \leftarrow base[idx]$
$i \leftarrow i + 1$
**end for**
```

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'':

$$\begin{aligned}
\text{before crossover}: \ & [0,0,0,0,\underline{0,0,1},2] = A \\
& [0,1,1,1,\underline{2,2,2},2] = B \\
\text{after crossover}: \ & [0,0,0,0,\underline{2,2,2},2] = A' \\
& [0,1,1,1,\underline{0,0,1},2] = B' \\
\text{after baseform}: \ & [0,0,0,0,1,1,1,1] = A'' \\
& [0,1,1,1,0,0,1,2] = B''
\end{aligned}$$

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.

3.3. A metric for mapping distance

To further exploit domain knowledge in our GA-based DSE, we first introduce a new *distance metric* that provides a measure of similarity between design points (i.e., mappings). In the next section, we subsequently explain how we deploy this distance metric in our GA-based DSE. To calculate the distance metric, we use an algorithm that for any pair of mappings (A, B) can perform a 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*: tasks mapped onto the same processor are in the same task group. In each recursive iteration of the algorithm, a pair of task groups $(\mathrm{tg}_A, \mathrm{tg}_B)$ will be selected where tg_A is a task group from mapping A and tg_B is a task group from mapping B. Next, certain tasks will be reassigned such that group tg_B becomes the equivalent of task group tg_A. In each level of recursion another pair of task groups will be selected and again reassignment of tasks takes place. This continues until the task groups in both mappings 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 B into A is minimal, which then yields the mapping distance between A and B.

Let **map** be a function that maps n tasks onto a k-processor system.

$$\mathbf{map} : \{0 \ldots (n-1)\} \mapsto \{0 \ldots (k-1)\}$$

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

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

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

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

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 describe the three important stages of the recursive algorithm:

3.3.1. Step 1: group selection

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

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

such that:

1. they share the maximum number of tasks: i,j with $\max(|\mathrm{tg}_{A,i} \cap \mathrm{tg}_{B,j}|)$, *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 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 B to A, as will be explained in the next section). Note that there may be more than one sequence that transforms B to A with the same number of reassignments, but finding one such sequence is sufficient for our purpose.

3.3.2. 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 is no other option than to try all of those pairs. To this end, a copy B' of B is created for each of these pairs 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

3.3.3. Reassignment function

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

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

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

3.3.4. Example

To illustrate the above algorithm, consider the following two mappings A and B for an application with 6 tasks 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:

$$\begin{array}{ll} \mathrm{tg}_{A,0} = \{0\} & \mathrm{tg}_{B,0} = \{0, 3, 4, 5\} \\ \mathrm{tg}_{A,1} = \{1\} & \mathrm{tg}_{B,1} = \{1, 2\} \\ \mathrm{tg}_{A,2} = \{2, 3, 4\} & \mathrm{tg}_{B,2} = \{\} \\ \mathrm{tg}_{A,3} = \{5\} & \mathrm{tg}_{B,3} = \{\} \end{array}$$

We find that the pair $(\mathrm{tg}_{A,2}, \mathrm{tg}_{B,0})$ has the largest overlap (tasks 3 and 4). Therefore: $B'[y] := 0$ 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 taskgroups are:

$$\begin{array}{ll} \mathrm{tg}_{A,0} = \{0\} & \mathrm{tg}_{B',1} = \{1, 2\} \\ \mathrm{tg}_{A,1} = \{1\} & \mathrm{tg}_{B',2} = \{\} \\ \mathrm{tg}_{A,3} = \{5\} & \mathrm{tg}_{B',3} = \{\} \end{array}$$

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

$$\begin{array}{ll} \mathrm{tg}_{A,0} = \{0\} & \mathrm{tg}_{B'',2} = \{\} \\ \mathrm{tg}_{A,3} = \{5\} & \mathrm{tg}_{B'',3} = \{\} \end{array}$$

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, and therefore the algorithm is run recursively for each pair: $(\mathrm{tg}_{A,0},\mathrm{tg}_{B'',2})$, $(\mathrm{tg}_{A,0},\mathrm{tg}_{B'',3})$, $(\mathrm{tg}_{A,3},\mathrm{tg}_{B'',2})$, $(\mathrm{tg}_{A,3},\mathrm{tg}_{B'',3})$. In this case all four recursive branches will find a minimum of two 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 $(\mathrm{tg}_{A,0},\mathrm{tg}_{B'',2})$: $B''' = [2,1,0,0,0,0]$ 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:

$$\mathrm{tg}_{A,3} = \{5\} \quad \mathrm{tg}_{B''',3} = \{\}$$

The only possible pair is $(\mathrm{tg}_{A,3},\mathrm{tg}_{B''',3})$. 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:

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

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

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

The above distance algorithm exhaustively tries all permutations of pairs, resulting in an algorithmic worst-case complexity of $O(k!n)$. Although this is fine for small k and n, it does not scale well to large problem sizes. However, it is possible [20] to reduce the distance metric problem to an assignment problem, which can be solved in only $O(n^3)$ using the Munkres assignment algorithm [13]. 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 $m_{ij} = n - |\mathrm{tg}_{A,i} \cap \mathrm{tg}_{B,j}|$ 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 $m_{ij} \geqslant 0$. 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.

3.4. A distance-metric based cross-over operator

As was explained, the distance metric can relate any two design points by finding a minimal set of atomic changes transforming design point B into design point A. 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 B (in unspecified order). The sequence of intermediate design points represents a "path" from B to A:

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

For example, in the example of the previous section, the following path was obtained to make design point B symmetrical to A (underlined indices refer to the tasks being reassigned):

$$B = [0,1,\underline{1},0,0,0]$$
$$B^1 = [\underline{0},1,0,0,0,0]$$
$$B^2 = [2,1,0,0,0,\underline{0}]$$
$$B^3 = [2,1,0,0,0,3], \text{ which is symmetrical to } A = [0,1,2,2,2,3]$$

Clearly, paths will be longer when A and B are less similar and when the problem space (and thus the chromosomes) are longer. The intermediate design points B^i share a varying number of characteristics from both A and B, since every application of an atomic change helps to transform B into A. 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*.

This new crossover operator creates offspring by simply selecting two random elements from the path between parents A and B. The rationale behind such a distance-path crossover is the hypothesis that an offspring design point that is chosen from the (shortest) path between its two parents A and B, which have been selected based on their fitness, will probably retain the strong chromosome parts of both of its parents. In other words, we try to exploit the locality of good design points, like described in Section 3.1, with the aim to produce offspring design points that are close to both parents. 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 *shortest* path between A and B. To complete the GA, the *path-crossover* operator can be followed by one of the standard mutation operators in order to introduce a controlled amount of new genetic material which is needed to avoid premature GA convergence.

3.5. 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 perform the baseform method after the modified crossover operator. In the example of the previous section, this would mean that if B^2 was the result of the crossover, then application of the baseform function would convert it to [0,1,2,2,2,2]. 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. We summarize in Fig. 2 that the proposed new approaches give rise to three new GA methods. These will be evaluated in the next section.

4. Experimental results

A set of experiments has been performed to determine the impact of the two approaches that were described previously. We look at both the impact of the crossover and baseform extensions separately as well as at the combined approach. As the basis for comparison we use a GA with tournament selection and uniform crossover [2] for searching the design space. As was mentioned before, we deploy the Sesame system-level simulation framework [19,7] to evaluate MPSoC design points during the process of DSE. Sesame enables rapid performance evaluation of different MPSoC architecture designs, application to architecture mappings, and hardware/software partitioning with a typical accuracy of 5% compared to the real implementation [19,7,16]. In Sesame, MPSoC system models are comprised of separate application and architecture models. An application model describes the functional behavior of an application, where our focus is on applications from the multimedia application domain. The architecture model defines architecture resources and captures their performance constraints. After explicitly mapping an application model onto an architecture

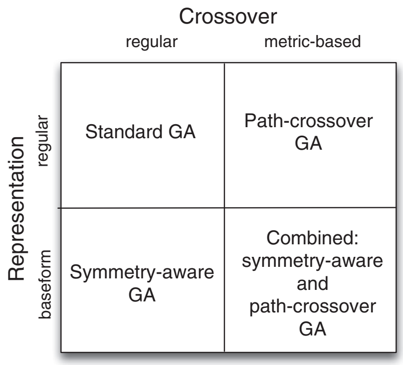


Fig. 2. Summary of different GA methods.

model, they are co-simulated via trace-driven simulation. This allows for evaluation of the system performance of a particular application, mapping, and underlying architecture.

Since the focus of this article is to study DSE search algorithm behavior rather than optimization of a specific application or platform instance, we use a synthetically generated workload that mimics a streaming (e.g., multimedia) application that has been generated with a modified version of the process-network generator of [17]. Such a synthetic workload facilitates the explicit controlling of specific application parameters, such as the number of application tasks and their connectivity, which is useful in the scope of our experiments. To challenge the search algorithm, we use a synthetic application model of considerable size: 20 tasks. The tasks are mapped onto a homogeneous, 8-processor MPSoC platform architecture using a single shared bus for communication. This results in a design space of approximately 3×10^{13} unique design points.

Fig. 3 shows the results of the four types of GA: the regular (reference) GA, symmetry-aware GA with baseform, GA with path-crossover and finally a GA with baseform and path-crossover. In the following, we will refer to these simply as: *reference*, *baseform*, *path-crossover* and *combined* GA. For the reference GA, we choose a uniform crossover operator, since it performed better than single or two-point crossover operators for our given problem space. We have tested three different mutation rates (top-to-bottom: 0.15, 0.1 and 0.05) as well as 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. All experiments are averaged over 120 runs to take the randomized, non-deterministic nature of a GA into account.

The top part of the graphs are P–Q (Probability–Quality) plots, in which the x-axis shows the quality of solutions (in terms of time units used for application execution: lower is better) and the y-axis the chance of obtaining such a solution. For a large design space with an unknown optimum, the PQ plots effectively are cumulative distribution functions of the experimental data (known as the empirical CDF) [22]. Dominating lines (towards the top-left) show a better performing GA since they indicate that a better result (lower execution time) can be obtained with a higher (average) probability. Although a PQ-plot gives 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. Therefore, to determine whether or not a method consistently outperforms another method, we also compute for each experiment the 80–95% confidence intervals of the mean value of the 120 repetitions of a single search method. The confidence intervals, shown at the bottom of the graphs in Fig. 3, indicate how certain (as specified by the confidence level) we are that the real mean 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 first subplot (Fig. 3a) 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 the 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 the design space, it is not possible to quantify exactly the meaning of a reduction of approximately 100 time units in objective space. However, we can see that the best result that we found in all experiments is 2331 when using a population size of 80 (Fig. 3d), which is approximately 40 units lower than our best average result in Fig. 3a. 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 the fact that increasing the population size increases the number of evaluations and thereby the search cost. When comparing the reference and combined GAs in relative terms in Fig. 3a: the probability of finding results within the range 2300–2475 differs around a factor 2 or 3.

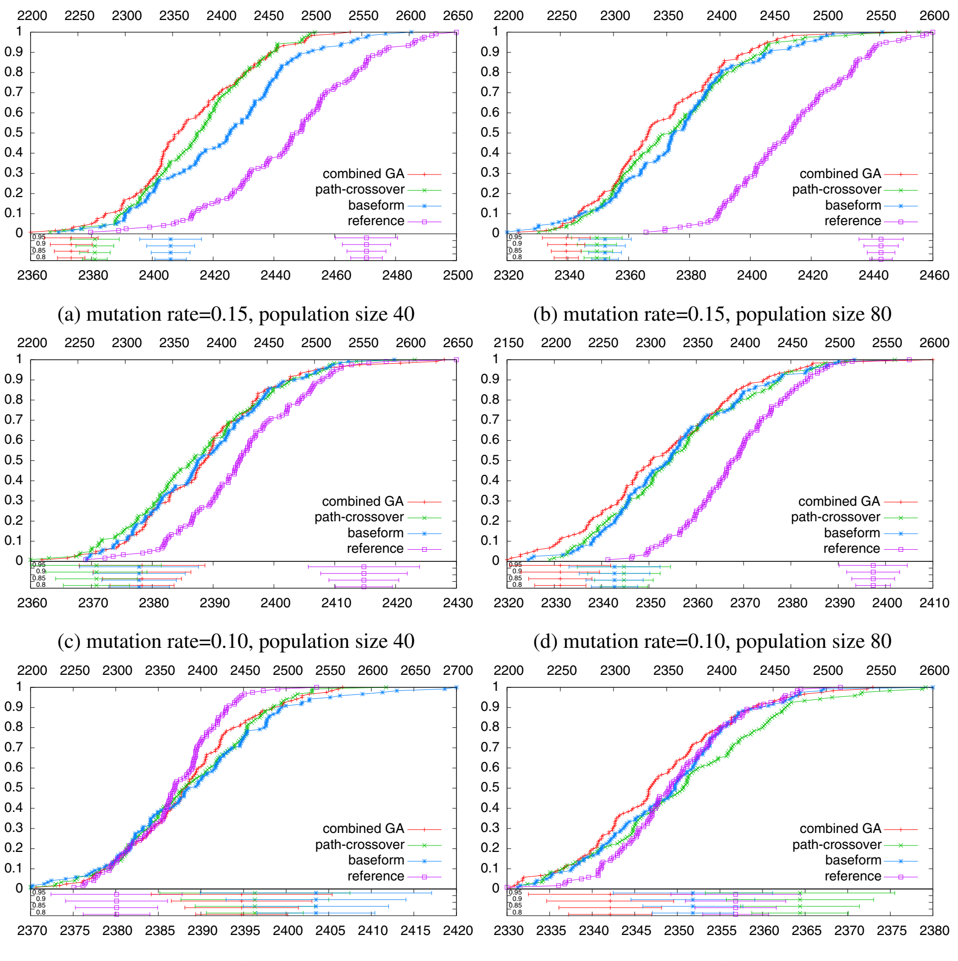
In Fig. 3a, we also see that the results of using only the baseform or only the path-crossover are also much better than the reference GA. The baseform-only GA seems to perform slightly worse than using only the path-crossover for mutation rate 0.15 (Figs. 3a and 3b). The confidence interval plot of 3a 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 (Fig. 3b) shows the same ordering, although the difference between path-crossover and baseform is less pronounced.

In the experiments with lower mutation rates (3c to f), the difference in performance between all GA methods becomes smaller. For mutation rate 0.10 (Figs. 3c and d), the performance of the extended GA methods starts 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 (Figs. 3e and f) the reference GA catches up. For a population size of 40, the reference GA even performs a little better than the other methods, though not so for the larger population size of 80 (Fig. 3f).

From these experiments, we conclude that using a sufficiently high mutation rate, the proposed GA extensions clearly perform better, or in the worst case similar, to the reference GA. For higher mutation rates or bigger populations, the combined method that combines the baseform and path-crossover performs best.

In a follow-up experiment, we check the impact of selective pressure on the performance of the GAs. Selective pressure 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 a selective pressure that is too high, only successful chromosomes make it into the next population and successful combinations with lesser chromosomes may be missed.

In Fig. 4, 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 previous experiment, each plot now only compares the combined method with the reference GA in Fig. 4. 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 with a low mutation rate perform relatively poorly: a mutation rate of 0.01



(a) mutation rate=0.15, population size 40 (b) mutation rate=0.15, population size 80

(c) mutation rate=0.10, population size 40 (d) mutation rate=0.10, population size 80

(e) mutation rate=0.05, population size 40 (f) mutation rate=0.05, population size 80

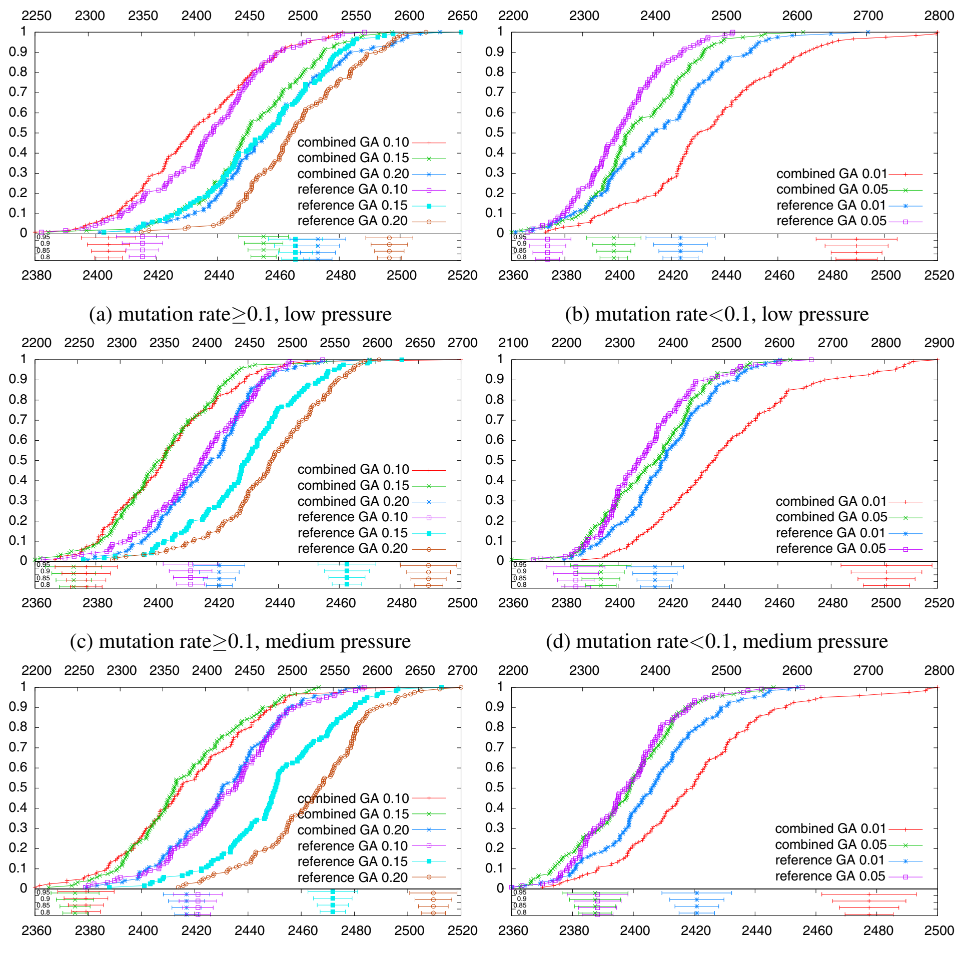
Fig. 3. P–Q plots for different GA types.

even shows the worst performance. A mutation rate of 0.05 performs better, but still worse compared to the reference GA for low pressure (Fig. 4b). In the case of medium and high pressure and a mutation rate of 0.05 (Figs. 4d and f), the combined and reference methods perform practically the same.

The situation in the left figures (with higher mutation rates) is very different. Now, the combined GA always outperforms the reference GA. 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 $\geqslant 0.10$, the best performing *reference* GA is the one with mutation rate 0.10. In case of low pressure (Fig. 4a), the combined method performs only slightly better than the reference GA with mutation rate 0.10. However, if we increase the pressure, then the distance between the two becomes much larger. For example, in case of medium pressure (Fig. 4c), the combined method GA with mutation rate 0.15 has an average result of 2376 and the reference GA 2410. And with high pressure (Fig. 4e), 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 rates 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 a mutation rate of 0.05 and low pressure (Fig. 4b). 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. 4b) versus 2404 for the combined GA (Fig. 4a). However, for the medium and high pressure cases, the combined GA method always results in a better average result, for both mutation rates 0.10 and 0.15. These results are consistent with those from 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.

Table 1 shows results for a range of different synthetic applications, where the number of tasks (N) in the application is varied as well as the task interconnectivity (a connectivity of 1 implies a fully connected graph). All numbers are percentages, where the combined GA is compared to the reference GA. The column BEST indicates the mean improvement of the best result found (averaged over 120 runs), the column IT the difference of the average search iteration in which the best result has been found, the column CONV the difference in convergence of the GA (where convergence is the mean improvement per GA iteration of found solutions), and the column TIME the difference in wall-clock time



(a) mutation rate≥0.1, low pressure

(b) mutation rate<0.1, low pressure

(c) mutation rate≥0.1, medium pressure

(d) mutation rate<0.1, medium pressure

(e) mutation rate≥0.1, high pressure

(f) mutation rate<0.1, high pressure

Fig. 4. P–Q plots for combined GA vs. reference GA.

Table 1
The combined and reference GAs compared for different applications.

N	BEST	IT	CONV	TIME
Graph connectivity 0.2				
10	−0.07	31.8	92.2	−1.8
14	0.13	16.4	50.9	−0.4
18	1.05	16.9	72.1	−0.2
22	1.90	7.7	−3.6	−0.2
Graph connectivity 0.5				
10	−0.12	28.7	46.3	−1.8
14	0.93	15.1	31.6	−0.6
18	1.72	4.7	19.0	−0.4
22	2.43	6.8	27.1	−0.4

of the search. The combined method shows a better performance for $N \geqslant 14$. The relative performance improvement of the combined GA seems small (between 0.1% and 2.4%), but this is likely due to the fact that both GAs have had the chance to fully converge. Had the GAs been allowed fewer iterations, then BEST would show a much larger difference, since the reference GA would likely not have fully converged. This is supported by IT, showing that the combined GA finds its optimum 5–32% faster than the reference GA. In general, we can observe the following trend in which the combined GA: (1) shows a similar result as the reference GA but using fewer GA iterations, or (2) shows a better result at the cost of more iterations (but still fewer iterations than with the reference GA). Moreover, the TIME column demonstrates that the execution times for the combined GA are only around 1% higher. Interestingly, this small overhead decreases for higher *N*, indicating that the operators for the combined GA are efficient for larger application models.

5. Related work

Current state-of-the-art in system-level DSE often deploys population-based Monte Carlo-like optimization algorithms like simulated annealing, ant colony optimization, or genetic algorithms (e.g., [21,6,17,8]). Several of these efforts also try to optimize the search by tuning the underlying search algorithm. For example, in [21], a fitness function (performance and cost) is defined that adds extra penalties to steer an evolutionary algorithm away from infeasible population individuals, while the work of [17] extends a standard simulated annealing algorithm with automatic parameter selection.

Design space pruning can also be performed via meta-model assisted optimization, which combines simple and approximate models with more expensive simulation techniques [3,14,18,5,1,12]. In [3], 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. In [5], the authors use meta-models as a pre-selection criterion to exclude the less promising configurations from the exploration. In [12], meta-models are used to identify the best set of experiments to be performed to improve the accuracy of the model itself. In [14], an iterative DSE methodology is proposed exploiting the statistical properties of the design space to infer, by means of a correlation-based analytic model, the design points to be analyzed with low-level simulations. The knowledge of a few design points is used to predict the expected improvement of unknown configurations.

Another class of design space pruning is based on hierarchical DSE (e.g., [10,15,11,6]). In this approach, DSE is first performed using analytical or symbolic models to quickly find the interesting parts in the design space, after which simulation-based DSE is performed to more accurately search for the optimal design points.

We have proposed domain-specific methods to optimize a GA using newly developed techniques that, to the best of our knowledge, have not been used previously in the field of system-level DSE. Our methods for optimization are not only relevant since the mapping-based representation is commonly used, but also because they are highly compatible with many other optimizations (so that different optimizations can be applied simultaneously).

6. Conclusions

In this article, we have addressed system-level design space exploration (DSE) for MPSoCs, and in particular, the exploration of application-to-architecture mappings and methods for optimizing such mapping exploration. To this end, we focused on DSE techniques based on genetic algorithms (GA) and introduced two new extensions to a GA that exploit domain knowledge in order to optimize the search process. One extension aims at reducing the redundancy present in chromosome representations of a GA, while the other extension introduces a new crossover operator based on a mapping distance metric. We have also investigated the combination of the two extensions. In the presented experimental results, the GAs with the proposed extensions perform at least as well, but typically much 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 rates and high selective pressure. We hypothesize that the higher mutation rate keeps population diversity high, 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 latter only seemed to perform well for low mutation rates and low pressure, whereas the extended methods performed better in all other situations. As part of our future work, we will consider whether the presented techniques can be further exploited to optimize GAs for design space exploration. This could, e.g., be done with so-called niching genetic algorithms, which require a measure of distance in the parameter search space to prevent premature convergence and to find optima in diverse areas of the design space. To our knowledge, no such GAs have been used for the purpose of system-level design space exploration, but the distance metric presented here may prove to be suitable for this purpose.

References

[1] G. Ascia, V. Catania, A.G. Di Nuovo, M. Palesi, D. Patti, Efficient design space exploration for application specific systems-on-a-chip, J. Syst. Architect. 53 (October 2007) 733–750.
[2] David Beasley, David R. Bull, Ralph R. Martin, An overview of genetic algorithms: Part 1, Fundamentals, Univ. Comput. 15 (2) (1993) 58–69.
[3] G. Beltrame, D. Bruschi, D. Sciuto, C. Silvano. Decision-theoretic exploration of multiprocessor platforms, in: Proceedings of the International Conference on Hardware/Software Codesign and System, Synthesis (CODES+ISSS), 2006, pp. 205–210.
[4] Kalyanmoy Deb, Samir Agrawal, Amrit Pratap, T. Meyarivan, A fast and elitist multiobjective genetic algorithm: Nsga-ii, IEEE Trans. Evol. Comput. 6 (2) (2002) 182–197.
[5] M.T.M. Emmerich, K. Giannakoglou, B. Naujoks, Single- and multiobjective evolutionary optimization assisted by gaussian random field metamodels, IEEE Trans. Evol. Comput. 10 (2006) 421–439.
[6] C. Erbas, S. Cerav-Erbas, A.D. Pimentel, Multiobjective optimization and evolutionary algorithms for the application mapping problem in multiprocessor system-on-chip design, IEEE Trans. Evol. Comput. 10 (3) (June 2006) 358–374.
[7] C. Erbas, A.D. Pimentel, M. Thompson, S. Polstra, A framework for system-level modeling and simulation of embedded systems architectures, EURASIP J. Embedded Syst. (2007) 1.
[8] F. Ferrandi, P.L. Lanzi, C. Pilato, D. Sciuto, A. Tumeo, Ant colony heuristic for mapping and scheduling tasks and communications on heterogeneous embedded systems, IEEE Trans. Comp. Aided Des. Integr. Circuits Syst. 29 (6) (june 2010) 911–924.
[9] M. Gries, Methods for evaluating and covering the design space during early design development, Integr. VLSI J. 38 (2) (2004).
[10] Z.J. Jia, A.D. Pimentel, M. Thompson, T. Bautista, A. Núñez, Nasa: A generic infrastructure for system-level MP-SoC design space exploration, in: Proceedings of the IEEE Workshop on Embedded Systems for Real-Time Multimedia (ESTIMedia), 2010, pp. 41–50.
[11] J. Kim, M. Orshansky, Towards formal probabilistic power-performance design space exploration, in: Proceedings of the 16th ACM Great Lakes symposium on VLSI, ACM, 2006, pp. 229–234.
[12] J. Knowles, Parego: A hybrid algorithm with on-line landscape approximation for expensive multiobjective optimization problems, IEEE Trans. Evol. Comput. 10 (1) (2006) 50–66.
[13] H.W. Kuhn, The Hungarian method for the assignment problem, Naval Res. Logist. Q. 2 (1–2) (1955) 83–97.
[14] G. Mariani, A. Brankovic, G. Palermo, J. Jovic, V. Zaccaria, C. Silvano, A correlation-based design space exploration methodology for multi-processor systems-on-chip, in: Proceedings of the Design Automation Conference (DAC), ACM, 2010, pp. 120–125.
[15] S. Mohanty, V.K. Prasanna, S. Neema, J. Davis, Rapid design space exploration of heterogeneous embedded systems using symbolic search and multi-granular simulation, SIGPLAN Notices, 2002, pp. 18–27.
[16] H. Nikolov, M. Thompson, T. Stefanov, A.D. Pimentel, S. Polstra, R. Bose, C. Zissulescu, E. Deprettere, Daedalus: toward composable multimedia MP-SoC design, in: Proceedings of the Design Automation Conference DAC'08, 2008, pp. 574–579.
[17] H. Orsila, E. Salminen, T.D. Hämäläinen, Parameterizing simulated annealing for distributing kahn process networks on multiprocessor SoCs, in: Proceedings of the International Conference on System-On-Chip, 2009, pp. 19–26.
[18] G. Palermo, C. Silvano, V. Zaccaria, Respir: a response surface-based pareto iterative refinement for application-specific design space exploration, IEEE Trans. Comp. Aided Des. Integ. Circuits Syst. 28 (2009) 1816–1829.
[19] A.D. Pimentel, C. Erbas, S. Polstra, A systematic approach to exploring embedded system architectures at multiple abstraction levels, IEEE Trans. Comp. 55 (2) (2006) 99–112.
[20] E-G. Talbi, B. Weinberg, Breaking the search space symmetry in partitioning problems: an application to the graph coloring problem, Theor. Comp. Sci. 378 (1) (2007) 78–86.
[21] J. Teich, T. Blickle, L. Thiele, An evolutionary approach to system-level synthesis, in: Proceedings of the International Workshop on Hardware/Software Co-Design, 1997, pp. 167–171.
[22] M. Thompson, Tools and techniques for efficient system-level design space exploration (PhD thesis), University of Amsterdam, February 2012.
[23] A. Tucker, J. Crampton, S. Swift, RGFGA: an efficient representation and crossover for grouping genetic algorithms, Evol. Comput. 13 (December 2005) 477–499.

Mark Thompson received the MSc (2004) and PhD (2012) degrees in computer science from the University of Amsterdam. His research interests include methods and tools for high-level simulation, modeling and performance analysis of heterogeneous embedded systems and design space exploration. Currently, he is a post-doctoral researcher at Leiden University Medical Center.

Andy D. Pimentel is associate professor in the Computer Systems Architecture group of the Informatics Institute at the University of Amsterdam. He holds the MSc (1993) and PhD (1998) degrees in computer science, both from the University of Amsterdam. He is co-founder of the International Symposium on embedded computer Systems: Architectures, Modeling, and Simulation (SAMOS) and is member of the European Network of Excellence on High-Performance Embedded Architecture and Compilation (HiPEAC). His research interests include computer architecture, computer architecture modeling and simulation, system-level design, design space exploration, performance and power analysis, embedded systems, and parallel computing. He serves on the editorial boards of Elsevier's Simulation Modelling Practice and Theory as well as Springer's Journal of Signal Processing Systems. Moreover, he serves on the organizational committees for a range of leading conferences and workshops, such as IEEE/ACM DAC, DATE, IEEE/ACM CODES+ISSS, IEEE ICCAD, FPL, SAMOS, and IEEE ESTIMedia.