electronics

MDPI

Article

Generative Design of the Architecture Platform in Multiprocessor System Design

Luise Müller *, Nico Schumacher, Lukas Steffen and Christian Haubelt *

Applied Microelectronics and Computer Engineering, University of Rostock, 18059 Rostock, Germany; nico.schumacher@uni-rostock.de (N.S.); lukas.steffen@uni-rostock.de (L.S.)
* Correspondence: luise.mueller@uni-rostock.de (L.M.); christian.haubelt@uni-rostock.de (C.H.)

Abstract: When designing a system at the Electronic System Level (ESL), designers are confronted with a very large number of design decisions, each affecting the characteristics of the resulting system. Simultaneously, the demands for the system's performance, reliability, and energy consumption have increased drastically. Design Space Exploration (DSE) aims to facilitate this complex task by automating the system synthesis and traversing the design space autonomously. Previous studies on DSE have mainly considered fixed architectures with a fixed set of hardware components only. In the paper at hand, we overcome this limitation to allow for a higher degree of freedom in the design of a multiprocessor system. Instead of a fixed architecture as input, we are using a resource library containing resource types whose instances can then be arbitrarily placed and connected. More specifically, we enable the exploration of the types, the number, and the positions of required processing-type instances in a grid-based topology template in addition to deciding on the remaining system synthesis tasks, namely, resource allocation, task binding, routing, and scheduling. We provide an extensible framework, based on Answer Set Programming (ASP) modulo Theories (ASPmT), for generating system architectures fulfilling predefined constraints. Our studies show that this higher degree of freedom, originating from fewer restrictions regarding the architecture, leads to an increased complexity of the problem. In extensive experiments, we show scalability trends for a set of parameters, demonstrating the capabilities and limits of our approach.

Keywords: generative design approach; hardware–software co-design; high-level synthesis; design space exploration; answer set programming

Citation: Müller, L.; Schumacher, N.; Steffen, L.; Haubelt, C. Generative Design of the Architecture Platform in Multiprocessor System Design. *Electronics* **2024**, *13*, 1404. https://doi.org/10.3390/electronics13071404

Academic Editor: Alexander Barkalov

Received: 15 February 2024
Revised: 4 April 2024
Accepted: 5 April 2024
Published: 8 April 2024

1. Introduction

Given the rising demands for functionality, performance, and cost, the system synthesis of embedded systems is a critical task. Simultaneously, the complexity of such systems, regarding the number of included components or their heterogeneity, is increasing and thus challenging. In reality, such systems are first designed at a high level of abstraction, i.e., at the Electronic System Level (ESL). Potentially good solutions for the system under consideration will be refined at lower levels of abstraction later on in the design process. In previous work, effective multi-objective Design Space Exploration (DSE) at the ESL, which can identify valuable solutions with desired properties from various design options, has been implemented as a binary ILP [1] problem first and then in ASP [2,3]. In system synthesis, ASP has been observed to be able to beat ILP in terms of runtime performance [4]. Compared to an implementation as a Boolean Satisfiability Problem (SAT), densely connected network structures with multi-hop communication can be encoded more efficiently in ASP [2,3].

However, most traditional approaches are limited because they only consider a fixed *architecture template* with a fixed set of available hardware components [3,5–11]. Alterations in the architecture between design points are only achieved by not allocating idle hardware components. This inflexible setup potentially excludes optimal solutions for

a given application, e.g., due to size constraints or an unsuitable component selection. For this reason, we are targeting a *free-world* approach, enabling the implementation of arbitrary architectures. To overcome the current limitations, in our proposed approach, we replace the architecture with a *resource library*, containing a set of *resource types*, available to generate the computation and communication infrastructure. Techniques like these are referred to as a Generative Design Approach (GDA) in the literature and can be understood as a methodology where the individual design points in the solution space (i.e., actual configurations of a product) are generated by a computer. The role of the designer can then be, for example, to pick from a set of generated configurations [12,13].

Nowadays, the GDA has been successfully applied in various engineering application fields, such as mechanical engineering [14–16]. But to the best of our knowledge, an automated application-specific platform generation at the ESL has rarely been studied before. An example illustrating the potential of application-specific designs can be found in a study of the CryptoManiac processor, designed exclusively for efficient cryptographic cipher execution [17]. Exploiting domain-specific characteristics and applying the respective optimizations to the architecture, the application-specific processor design can significantly outperform a high-end general-purpose design for the given application [17]. Therefore, we aim to exploit the advantages of the GDA in the design process of embedded multiprocessor systems.

However, the assessment of all architectural design options comes at the expense of the complexity of the DSE. To deal with the explosion of complexity, as a first step, we approach the ideal free-world design process by the use of a *topology template*, where only the communication infrastructure is fixed in the specification. Given an application, we select computation types from the resource library and position their instances without restrictions inside this topology template to answer the following questions, among others:

- How many processing units are required for an optimal execution of the given application?
- Which processing types are required for an optimal execution of the given application?
- What is an efficient distribution of tasks from an application onto processing units?
- How do we organize the communication structure to enable an efficient synchronization of the given processes?

Our contribution is a novel system synthesis problem definition based on ASP that does not require a fixed architecture template as input but instead considers a resource library containing resource types and a topology template for the communication infrastructure. We provide a modular and extensible encoding that enables a higher degree of freedom in the design of a multiprocessor system and allows a multi-objective optimization considering solutions to the architecture design as well as system synthesis decisions. The same generality might also be achieved using traditional approaches, but it would require a considerable effort to specify a template that did not restrict the design space. With this approach, we aim to answer the following questions:

- How can we efficiently encode a GDA at the ESL using ASP?
- How well is this problem solvable for various problem sizes?
- What might be challenges on the way to an unrestricted generative design at the ESL?

2. Related Work

An overview of DSE and its concepts can be found in [18]. State-of-the-art DSE approaches can be divided into exact and non-exact (meta-heuristic) design space searching methods [19]. Non-exact methods often do not explore the design space systematically, potentially leading to a duplicate evaluation of individual design points and degraded runtime performance [20]. An example of non-exact DSE at the ESL can be found in [21], where the authors used genetic algorithms to map tasks on a fixed architecture template of eight homogeneous processors connected via a single shared bus. Similarly, Richthammer et al. [6,22] proposed strategies for improving the efficiency of meta-heuristic DSE approaches, again mapping tasks to resources in fixed architecture templates.

In the paper at hand, we focus on ASP, which is an exact method. ASP has some advantages over other exact methods, like ILP and the SAT. Ishebabi et al. [4] concluded in their comparative study that ASP has a greater potential for solving synthesis problems than ILP. They report that an ASP encoding of the synthesis problem was solved up to three orders of magnitude faster while still achieving results of the same quality compared to ILP. ASP, as opposed to the SAT, relies on a closed-world assumption that allows the direct expression of reachability [2,20]. Consequently, denser network structures with multi-hop communication benefit from considerably reduced solving times due to smaller problem descriptions [2,20]. A more general survey of ASP and its usage can be found in [23]. The authors provide an overview of what kinds of problems lend themselves well to being solved with ASP. They also provide a thorough overview of successful applications by listing concrete examples and case studies covering many domains.

Many previous state-of-the-art DSE publications [3,5–11] are based on or are similar in nature to the system model originally presented by Blickle et al. in [24], which uses fixed architecture graphs. They all have in common that they map tasks directly to processing elements in a fixed architecture graph. Though processing units can be omitted in the final platform, the architecture graph is fixed and specified before the DSE is performed.

The problem of generating an application-specific hardware platform at the ESL from a specification of a task graph, to the best of our knowledge, has not been approached with the same generality that we are proposing. There are, however, several works that have covered subsets of our problem statement. These subsets usually assume some parts of the system as given, e.g., the number and kind of processing and/or communication elements. In [25], Todorov et al. propose a solution to a constrained version of the synthesis problem in which the processing elements are given by a floor plan and the goal is to generate the communication infrastructure between them. Communication scenarios are specified by a set of graphs, which the authors call use cases. Their approach is based on a combination of clustering and pathfinding algorithms that are applied successively. Likewise, in [26], Li et al. consider a similar problem but focus specifically on fault tolerance of the communication infrastructure. They use ILP for acquiring valid solutions. In [27], an ILP formulation and a heuristic approach are used to synthesize an optimal crossbar configuration for a Multiprocessor System-On-Chip (MPSoC). Starting with the communication information obtained from a simulation using a full crossbar structure, the architecture is optimized for power efficiency without sacrificing timing requirements.

All previous examples have in common that they focus on generating only a subset of the final hardware platform, i.e., the communication infrastructure. Additionally, studies have investigated the generation of hardware without an architecture template tailored to addressing domain-specific challenges. In [28], the authors presented an evolutionary algorithm for generating feasible architectures for the specific case of network processors. Similarly, in [29], Lieverse et al. used DSE methodologies based on simulation for signal processing systems.

Probably the closest study was performed by Ishebabi and Bobda [30], who investigated automated architecture synthesis for multiprocessor systems on an FPGA. By formulating the problem as an ILP problem, they assigned tasks to specific processors, without excluding mappings, and communication tasks to communication topologies. Compared to our approach, they only minimized the overall computation time, whereas we run a multi-objective optimization to be able to compare the trade-offs for our objectives: latency, cost, and power consumption. With their framework, they generated an abstract architecture description, which could be used for hardware synthesis later on, whereas we also produce a feasible schedule with a minimum latency. This enables the designer to compare the latency for different architectures and check if certain latency restrictions can be met.

In general, our approach is similar to the problem of High-Level Synthesis (HLS), which uses a behavioral description of the application to generate a Register-Transfer-Level (RTL) description of the architecture [31]. One major advantage of using automated HLS

compared to traditional hardware modeling is the increased flexibility and productivity in the exploration of multiple designs and design alternatives [31]. Under certain circumstances, the hardware generated by a HLS tool is able to achieve comparable results in some metrics, while outperforming optimized hardware in other metrics [32]. Compared to our approach, HLS is usually applied to problems that are smaller and more specific in nature, e.g., a signal processing algorithm in a programming language like C. This means that in practice, although the goals are similar, the methods employed can differ significantly. Furthermore, HLS focuses on designing all elements directly in hardware, whereas we aim to investigate the trade-offs in hardware–software co-design. Ultimately, the level of complexity for the communication architecture is usually significantly higher in our case, e.g., in packet-based, multi-hop on-chip networks, which our encoding supports.

3. System Model

In our approach, we model a hardware/software system at a high level of abstraction, specifically, the ESL. The system specification $S = (A, L, M)$ consists of an application graph A, a resource library L, and a set of mapping options M. The application $A = (V_A, E_A)$ contains a graph-based high-level description of the system's behavior in which the elements of the vertex set V_A are distinguished between computational tasks T and communication messages C. The set of communication edges $E_A \subseteq (T \times C) \cup (C \times T)$ defines the dependencies among the elements. For further details, an interested reader can refer to [33].

The definition of the resource library L is closely related to our previous work in this field [3,10], where we specified a fixed architecture template containing processors, routers, and links at certain positions, which can be allocated. In contrast, we define the resource library as a set of resource types that are partitioned into processing, communication and interconnection types $Types = Types_{Proc} \cup Types_{Comm} \cup Types_{Link}$, with $Types_{Proc} \cap Types_{Comm} \cap Types_{Link} = \emptyset$. Each resource type $Type \in Types$ is parameterized by the functions $P_{stat} : Type \to \mathbb{N}$ and $area : Type \to \mathbb{N}$ representing the static characteristics, power consumption, and area costs. The final architecture is composed of a selection of instances of the given types. Processing units, as instances of processing types $Types_{Proc}$, are potential targets for the execution of the tasks $t \in T$ from the application graph A.

In the future, further properties could be attached to these processing types, which in turn influence the metrics of the execution of tasks on these processing types. It would be possible to assign each processing type a distinct Instruction Set Architecture (ISA). This would lead to varying average Cycles per Instruction (CPI) or Energy per Instruction (EPI) values between the processing types. In combination with the properties of each task T, distinct values for dynamic characteristics such as execution time or power consumption could be derived for the execution of the respective task on each processing type. Furthermore, we could prohibit the execution of certain tasks by not offering specialized features or instructions like floating point arithmetic. But the flexibility of our model extends even further. One processing type could also be an application-specific hardware block, which is optimized for the execution of exactly one task in the task graph. This versatility allows us to explore the complex relationships involved in the co-design of hardware–software systems.

Communication units as instances of communication types $Types_{Comm}$ are used to form a communication infrastructure to enable the transfer of messages between two processing nodes. For each processing and each communication type, we additionally specify how many connection points it possesses, i.e., to how many units of any type an instance of that specific resource type can be connected to. Finally, instances of interconnection types are used as interconnections between elements of processing and communication types. Each interconnection type $Type_{Link} \in Types_{Link}$ has additional characteristics like the transmission energy consumption $E_{transmission} : Type_{Link} \to \mathbb{N}$ or the transmission delay $\delta_{transmission} : Type_{Link} \to \mathbb{N}$. Referring back to the design of a multiprocessor system, our concept can model well-known communication structures in System-on-Chip (SoC) design like bus-based communication and Network-on-Chips (NoCs) [34]. On the one

hand, for a bus-based communication, we can assign a respective communication type with properties that specify the maximum number of connected devices and the blocking connection arising for the bus-based structure, whereas the respective interconnection types can model the access times and transmission cost for messages. On the other hand, we can model the routers inside a NoC by communication types with a lower maximum number of connections, whilst the transmission delay and cost are again modeled by the interconnection type.

Due to its modular design, we can easily extend the resource library with new types and the existing types with new features. Overall, the definition of the resource library is at a conceptional level of abstraction. Further research would be required to bring this theoretical model closer to real-world applications. Nevertheless, this higher level of abstraction is also beneficial because the proposed framework and concepts might be applicable to problems of other granularity or in other domains, like for the synthesis of an efficient data-center layout for given applications.

A set of mapping options $M \subseteq T \times Types_{Proc}$ assigns each task $t \in T$ with at least one processing type $type_{Proc} \in Types_{Proc}$, representing that the task t can be potentially executed on an instance of the processing type $type_{Proc}$. In contrast to that, in previous work, the authors assigned each task to a specific processor instance [3,10,30]. For each mapping option $m \in M = (t, type_{Proc})$ the function $w : M \to \mathbb{N}$ defines the worst-case execution time of the task t on an instance of the processing type $type_{Proc}$. Likewise, $E_{dyn} : M \to \mathbb{N}$ models the dynamic energy requirement of each mapping option. As stated earlier, these mapping options might be derived from specific properties of the tasks and the processing types. So far, communication messages are not explicitly assigned with mapping options. However, we can imagine this extension, such as $M \subseteq C \times Type_{Comm}$. This enables a specification of varying values for $E_{transmission}$ and $\delta_{transmission}$ for differently sized messages.

The state-of-the-art work [3,5–11] allows the exploration of the architecture up to a certain degree. Given a fixed architecture template, these approaches derive different architectures by allocating only those resources from the specification that are actually used for the task execution and message transmission in the final implementation. In our new approach, we simplify the process of specifying the system's requirements for a user and offer an elegant formulation. Further, our encoding allows us to leave the exact number of instances of each resource type unspecified.

For illustration, we show an example in Figure 1a. Given a task t_1 and two processing types $type_1$ and $type_2$, there are two options for the execution of task t_1. Simultaneously, a given bus-based architecture template offers two slots for the positioning of at most two instances of the given processing types. In Figure 1b, we illustrate the specification of the emerging problem. In the state-of-the-art approach (top), the task requires a mapping option to each instance of each type in each position to avoid eliminating any solution from the feasible solution space. In general, the number of instances from each processing type is set to the maximal required number, i.e., to the number of available positions. In contrast to that, in our proposed approach, we map the task to processing types instead of processing types' instances and explore to which position these will be assigned to.

In order to achieve the same flexibility for the state-of-the-art approaches, a vast specification overhead is expected. Figure 1c compares exemplarily the required number of mapping options for an increasing size of the potentially generated architecture of the two approaches. In this case, we consider a two-dimensional grid-based architecture template as well as a small test instance containing five tasks and four processing types. Regarding the positions on the grid, our proposed approach has a constant specification overhead with a maximum number of $|T| \cdot |Types_{Proc}|$ mapping options, whereas previous approaches would require at most $|T| \cdot |Types_{Proc}| \cdot |GridPositions|$ mapping options. This simple example can clearly show that the specification overhead for a user is massively reduced.

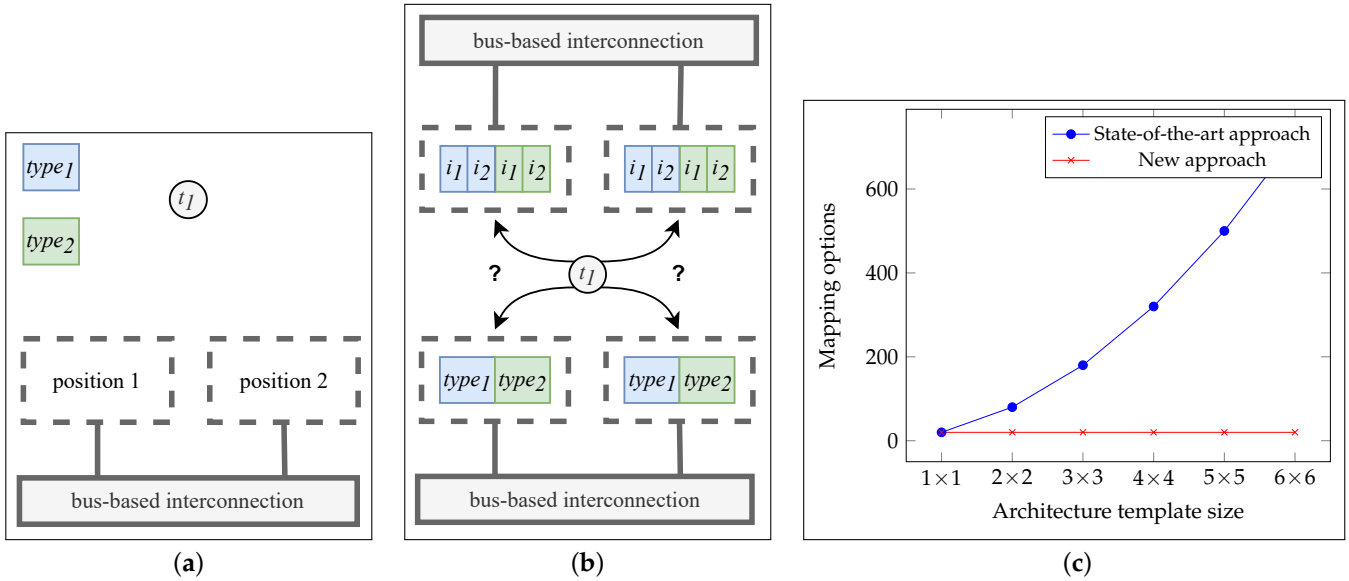


Figure 1. Comparison of the maximal number of specified mapping options for different architectures. (**a**) Given are a task t_1 which shall be executed, two processing types, whose instances can be target for the execution of task t_1, and a bus-based communication infrastructure with two available slots for the potential positioning of at most two instances of the given processing types. (**b**) Problem formulation for the state-of-the-art (top) and our new (bottom) approach on a bus-based architecture template. (**c**) Comparison of the maximal number of specified mapping options for the state-of-the-art (blue) and our new (red) approach on a grid-based architecture template.

In Figure 2, we summarize the system model of our approach. The specification and its elements act as input to the DSE. Simultaneously, we consider user-defined rules and constraints on the architecture, which allows a user to direct the search towards desired architecture topologies, e.g., by determining the allowed communication types or by setting a limit on the architecture size or the costs. That way, our approach can address individual customer requirements. Finally, during the DSE, we explore the solutions for the system implementation, including an application-specific architecture.

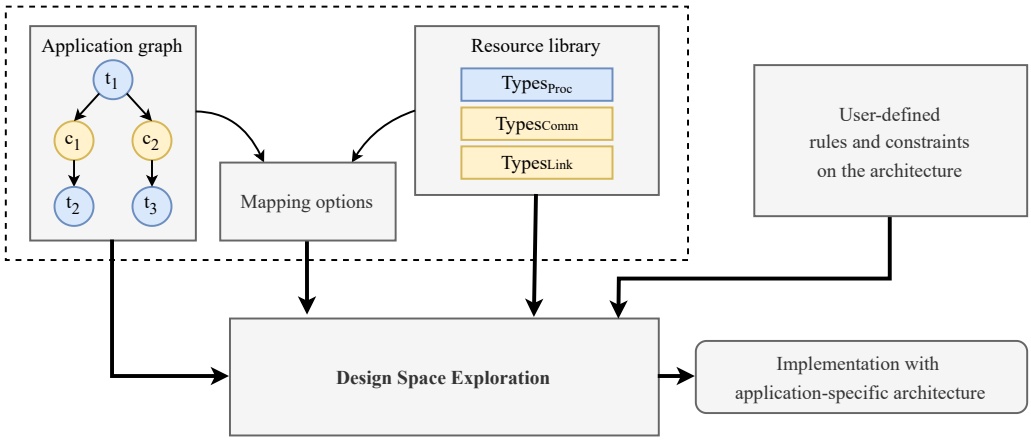


Figure 2. An overview of the approach.

4. Exploration Model

In this section, we present the key aspects of the DSE. This includes the decisions on the system synthesis, the concept of Pareto optimality, and the exploration of an application-specific architecture. In the end, we explain the realization of our approach in ASP.

4.1. System Synthesis

Given a system specification, the DSE searches for a feasible system implementation, i.e., the behavioral description and the properties of the system are translated into a structural description. In order to get a valid solution for the synthesis problem, a valid allocation, binding, routing, and schedule have to be determined. The allocation α is a set of instances of all resource types that are used to realize the desired system functionality. The binding β provides information on the assignment of the functionality to the allocated hardware resources. We distinguish between type binding $\beta_{Type} \subseteq M$ that selects exactly one mapping option, i.e., one processing type per task, and instance binding $\beta_{Instance}$, specifying on which processing-type instance each task is actually executed. Obviously, the selected instance binding has to match the chosen type of binding. The routing γ chooses for each message a cyclic-free path in the communication structure, depending on the instance binding of the sending and receiving tasks, resulting in a subset of the allocated interconnection types. Thereby, each subset can contain instances of different interconnection types. Finally, the schedule τ assigns to each task and each communication message a starting time for its execution on the selected resource.

The underlying encoding of the system synthesis problem is based on [3]. But we have replaced the fixed hardware architecture with a resource library containing resource types. Thus, we additionally explore the topology and size of the architecture besides the system synthesis decisions (for a detailed description of our objective functions, an interested reader can refer to [10] and for our optimization framework to [3]).

Depending on the design decisions that are made, the resulting system-level implementations show different qualities. Typically, to measure the quality of an implementation x, we assess more than a single property. In this approach, we focus on the overall latency $lat(x)$ of the system, its area costs $area(x)$, and its overall energy consumption $E(x)$. Regarding these, we formulate the DSE, without loss of generality, as a multi-objective minimization problem [24]:

$$\begin{aligned} &\text{minimize } f(x) = (lat(x), area(x), E(x)), \\ &\text{subject to:} \\ &\quad x \text{ is a feasible system implementation.} \end{aligned} \tag{1}$$

Due to the conflicting objectives, the system synthesis commonly has a set of Pareto-optimal solutions X_P. These are determined by the dominance relation $\succ$. Considering the n-dimensional quality vectors of two distinct solutions, a solution x dominates another solution y ($x \succ y$), if x is at least as good in every objective as y, and if it outperforms y in at least one objective. We call a solution x Pareto-optimal, if there is no dominant solution y over x. Hence, by definition, Pareto-optimal solutions in the Pareto set X_P for a given problem are mutually non-dominated by each other: $\nexists x, y \in X_P : x \succ y \vee y \succ x$.

4.2. Architecture Exploration

Unlike previous work, this approach also considers design decisions on the target architecture in the DSE. To take advantage of application specificity, we adapt the size, structure, and elements of the architecture to the respective application. In detail, the design decisions include the selection of adequate resource types from the resource library, the selection of a sufficient number of instances of these resource types, as well as their positioning in the final architecture. Simultaneously, the decisions on the system synthesis are made, according to Section 4.1. In the end, the generated platform is appropriate for an efficient, energy-saving execution of the application and the corresponding process synchronization over the communication structure.

Usually, real-world examples require an enormous number of design decisions to be made. With the increasing size of the given application, the number of potential design options grows exponentially, and thus, the search space cannot be explored exhaustively in a feasible time. As an example, the work of [10] addresses this issue in the product

development of embedded systems. Instead of an exhaustive search, the authors propose a heuristic DSE in order to access good solutions early in the search process.

The introduced architecture exploration expands the complexity of the DSE by another dimension. As illustrated in Figure 3, there is a trade-off between the freedom in the design and the introduced complexity to the DSE. Starting with a specific architecture (corresponding to the state of the art), we aim for an approach where, given a blank sheet of paper, a tool automatically selects components of any type from a resource library and generates a valid and optimal architecture along with the determination of a valid and optimal system implementation. We call this approach "free world" in order to differentiate it from others, because it is unique in not limiting the number of feasible solutions. But due to the vast number of design options, we are confronted with an explosion of the solution space. Therefore, we approach our goal step-by-step and investigate the question of how much freedom is actually required to generate a platform that perfectly fits the application's needs.

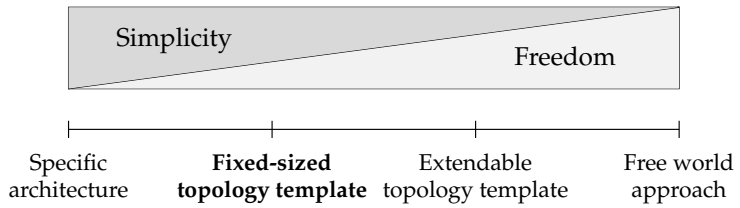


Figure 3. Trade-off between simplicity of the Design Space Exploration (DSE) and freedom of the design decisions.

As a first step in the investigation of the idea of an automatic application-specific architecture generation, we study in this paper the DSE based on structural topology templates. Therefore, we start from a fixed-size topology template in a grid-based layout, similar to distributed switched networks popular in larger MPSoC designs [34]. As shown in Figure 4a, we consider an $(m \times n)$ grid consisting of $m \cdot n$ instances of a communication type *router*. For simplicity, in this paper we only consider two dimensions, although our encoding allows a three-dimensional grid structure. Further, $m \cdot n$ slots are available for the placing of processing-type instances. All components are bidirectionally connected at maximum via $2 \cdot ((m-1) \cdot n + m \cdot (n-1))$ instances of an interconnection type *link*. During the DSE, besides the decisions on the system synthesis, we explore the type and number of required processing-type instances and their positions on the grid. Instances, including the predefined communication- and interconnection-type instances in Figure 4a, are only allocated in the final architecture if they are actually used in the respective system implementation. Thus, at this point, we handle the exploration of the communication infrastructure similarly as in related work [3,10,11]. Nevertheless, the size of the architecture is still subject to exploration. Only the communication topology and the maximum size of the platform is determined by the given template.

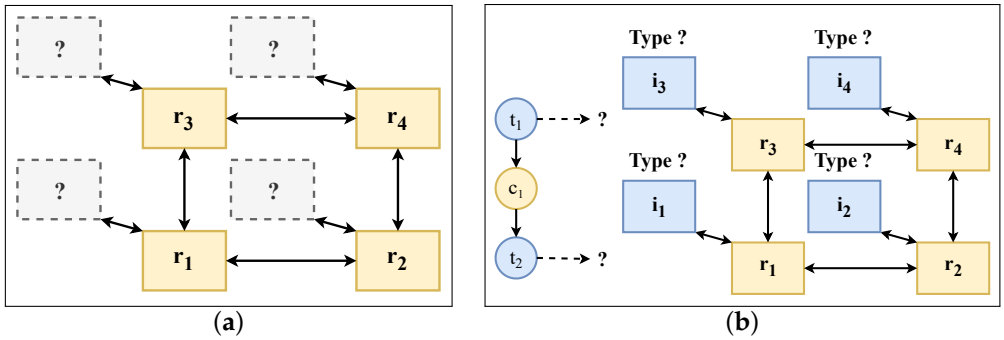


Figure 4. Template of (2×2)-grid-based topology. (**a**) Exploration of selection and positioning of processing-type instances. (**b**) Exploration of instance type and task at each position.

We introduce the topology template in our framework by using "user-defined rules and constraints on the architecture", illustrated in Figure 2. Any other architectural topology template as well as a more or less strict set of constraints can easily be included via this category. Therefore, our framework is already prepared to be utilized for the next steps according to Figure 3. Nevertheless, our approach already introduces more flexibility in the development process of a hardware/software system, and we receive valuable insights on its effect on the efficiency of the DSE and the quality of the feasible design options.

4.3. Encoding in ASP

We propose an implementation of the given design problem in ASP, a declarative programming paradigm tailored towards computationally difficult problems, especially NP-hard combinatorial search problems, and is based on the state-of-the-art API of the ASP tool clingo [35]. In the following, we give a brief overview of the core concepts that are imperative to the exploration of an application-specific architecture.

An ASP encoding is typically separated into a general problem description and a specific problem instance. The initial knowledge of an instance is represented as a set of facts. Listing 1 shows a collection of exemplary elements from a system specification.

Listing 1: Exemplary facts representing aspects of the system's specification.

```
1  % Dimension of an exemplary architecture template
2  dimension(2,2).
3  % Exemplary entry from the resource library
4  processingType(ptype1).
5  resourceCost(ptype1,23).
6  staticPower(ptype1,26).
7  % Exemplary entry the from task graph
8  task(t1).
9  % An exemplary mapping option
10 map(t1,ptype1).
11 dynamicEnergy((t1,ptype1),620).
12 executionTime((t1,ptype1),38).
```

Besides the (x,y)-dimension of a (2×2) grid, Listing 1 defines a processing type `ptype1`, a task $t1$, and a possible mapping option from `t1` onto `ptype1`. Processing type `ptype1` is further specified by its static characteristics. On lines five and six, exemplary values are defined as the resource cost and the static power consumption of processing type `ptype1`, and, respectively, for every instance allocated from this type. Similarly, on lines eleven and twelve, the potential execution of task `t1` on processing type `ptype1` is assigned with exemplary values for its dynamic characteristics.

The problem description is given as a set of rules, each consisting of a head and a body, separated by a colon. The head of a rule is inferred if the respective body holds, i.e., is fulfilled. A fact represents a rule that has an empty body and thus holds unconditionally. In contrast, an empty-headed rule, called an integrity constraint, cannot be inferred. Accordingly, integrity constraints are used to exclude specific assignments from a stable model. In Listing 2, we show an example for a normal rule (on line two) and an integrity constraint (on line three). In this example, we first derive from the given dimension `(NX,NY)` of a grid-based architecture template the maximal available number of slots `N` for placing processing-type instances. On the third line, in the curly brackets, we aggregate a group of items that remains after evaluating the condition inside. In this case, we collect each allocated resource type that is a processing type. Communication or interconnection types do not fulfill this condition. Finally, we compare the number of elements to the so-called "guards", which can optionally be set around the curly brackets, indicating a minimum and maximum limit. The exemplary integrity constraint ensures that considering all given processing types, not more instances than available slots M in the template are allocated.

Listing 2: Exemplary rule and integrity constraint.

```
1 % It is not allowed to allocate more processing type instances
    than the amount of available slots
2 numberProcessorSlots(N) :- dimension(NX,NY), N=NX*NY.
3 :- M+1 { allocated((TYPE,NR)) : processingType(TYPE) },
    numberProcessorSlots(M).
```

Note that the capital letters in Listing 2 represent variables. Before the solving, the ASP encoding is grounded into a variable-free representation, i.e., each variable is replaced by its instances, and all possible combinations are generated. Then, during the solving, a truth assignment satisfying the given set of propositional formulas is inferred. The resulting answer set is a valid implementation of the specified system.

In our approach, we introduce a numbering of the positions in the grid-based architecture template based on the respective coordinates `(X,Y)` according to Listing 3, which simultaneously will be the implicit number `NR` of each potentially placed instance. The resulting numbers are illustrated in Figure 4b.

Listing 3: Introduction of position numbering to identify positions in the grid.

```
1 % Allowed positions in the grid
2 implicitPosition(NR,(X,Y)) :- dimension(NX,NY),
    X=1..NX, Y=1..NY, NR=X+NX*(Y-1).
```

That way, each position has a unique number, regardless of the optional placing of an instance. This eliminates a vast amount of redundancy that would be introduced if we placed equal-typed but differently named instances instead. Hence, the unambiguous implicit instance numbering decreases the size of the problem grounding and of the solution space. Subsequently, in Listing 4, the ASP solving has to make guesses on the following two issues:

- For each task `T` of the application graph, exactly one binding to a place in the hardware grid based on the implicit position numbering `NR` is selected (line two).
- For each implicit position number `NR` in the architecture template, a maximum of one processing type `TYPE` from the resource library is selected (line three).

We illustrate these questions in Figure 4b as well. As shown in Listing 4 on line six, a common type and task binding to one position `NR` leads to an instance binding of the respective task `T` to the respective instance `NR` of the processing type `TYPE`. Finally, on lines four and seven, we explicitly place in the implementation the utilized processing-type instance `(TYPE,NR)` at its respective location and allocate it.

Listing 4: Problem description of the processing type instance selection and positioning as well as of the task binding.

```
1 % Guessing of position binding per task and of processing type
    per position
2 1 { placeBind(T,NR) : implicitPosition(NR,_) } 1 :- task(T).
3 { placeType(TYPE,NR) : processingType(TYPE) } 1 :-
    implicitPosition(NR,_).
4 location((TYPE,NR),(X,Y)) :- placeBind(T,NR),
    placeType(TYPE,NR), implicitPosition(NR,(X,Y)).
5 % Both together result in instance binding of each task
6 bind(T,TYPE,NR) :- placeBind(T,NR), placeType(TYPE,NR).
7 allocated((TYPE,NR)) :- bind(T,TYPE,NR).
```

To guarantee valid solutions, we add the constraints in Listing 5 to the encoding. These ensure, on the one hand, that the task and type binding to a common position have to match the specified mapping option of the task. On the other hand, they guarantee

that no task binding without a respective type at the common location and no type at a location without a respective task binding exists. This eliminates not only invalid but also redundant solutions.

Listing 5: Additional constraints required to ensure valid solutions.

```
% Task is bound, and the processing type must fit the
    respective position
:- placeBind(T,NR), placeType(TYPE,NR), not map(T,TYPE).
% No binding without a respective type at the location and no
    type at a location without a respective binding
:- placeBind(_,NR), not placeType(_,NR).
:- placeType(_,NR), not placeBind(_,NR).
```

The presented encoding represents the problem description of the selection, positioning, and allocation of the processing-type instances, as well as the type and instance binding of each task. Simultaneously, the solver decides on the message routing, the allocation of communication- and interconnection-type instances, and the scheduling.

Note that a detailed description of the ASP solving process and of the ASP grounding and solving tool clingo is out of scope of this paper. An interested reader can find additional information at [35,36]. Further, a detailed explanation of the ASP encoding of the system synthesis steps, including allocation, binding, routing, and scheduling, can be found at [10].

5. Simulation Study

In this section, we present the setup and the outcome of our simulation study. Despite enhancements in the solving process, deciding on a valid implementation of a system remains an NP-complete problem. In the worst case, the runtime of the DSE grows exponentially with the number of decision points in the problem formulation. Since we introduced another order of magnitude of complexity to the system synthesis problem, we wanted to evaluate the feasibility of our approach by testing it with different-sized system specifications. In detail, we varied the size of three parameters to investigate the scalability trend of our approach, namely, the application graph, the resource library, and the topology template. We looked into the structure as well as the quality of the generated application-specific architectures.

5.1. Simulation Setup

For the automatic generation of our benchmark set, we adapted the ASP-based benchmark generator from Neubauer et al. [33] to our new system model. Instead of a complete hardware architecture in [33], we generated only our grid-based topology template, consisting of instances of the communication type router and the interconnection type link, analogously to our explanations in Section 4.2. The resource library was a completely new element in our benchmark set. It contained resource types and their respective characteristics (as defined in Section 3). Similar to the implementation in [33], each processing type offered a set of instruction types. Simultaneously, each task required specific instruction types for its execution. The assignment was set randomly in each case. Tasks had mapping options to all compatible processing types, i.e., to those supporting their required instruction types. At the same time, we added ASP constraints to the benchmark generator to ensure that each task had at least one mapping option.

All in all, the mapping options and additionally, the respective dynamic characteristics, were automatically generated from a given application and resource library. This way, a user only needs to define the workload but does not necessarily need to care for the elements in the resource library. This supports our overall goal of easing the specification process for the user. For comparison, in [33], a set of mapping options to specific processor instances was randomly generated, and its number per task ranged between a user-defined minimum and maximum value. The construction of application graphs, generated as “series parallel

graphs", was taken unchanged from [33]. In total, we took ten application graphs, seven topology templates, and three resource libraries. We present the details in Table 1.

Table 1. Overview of the generated benchmark components.

Component	Instances
Application	Containing 6, 15, 31, 39, 44, 56, 62, 73, 86, or 92 tasks
Topology	Containing a (2×1), (2×2), (2×3), (3×3), (4×4), (5×5), or (6×6) grid
Library	Containing 4, 6, or 8 processing types

Based on this setup, we performed two simulation experiments. For all applications, we first set the library to contain four processing types and examined the influence of the seven topology template sizes. Secondly, we fixed the topology template to the (2×2) grid and tested our approach with three resource libraries, each containing a different number of processing types. Additionally, we examined the impact of the ten differing application graph sizes in both simulations. In total, we conducted $(10 \cdot 7 \cdot 1) + (10 \cdot 1 \cdot 3) = 100$ DSEs, each limited to one hour of time and 100 GByte of RAM usage. To ensure both limitations, we utilized the tool runlim, version 2.0.0rc12 (runlim tool, available online: https://github.com/arminbiere/runlim (accessed on 24 January 2024)). We tested the introduced scenarios for the tool clingo, version 5.7.0, its extension clingo-dl, version 1.4.1, and Python version 3.10.10, executing on the Ubuntu 22.04.3 operating system. The platform itself contained an Intel Xeon Gold 6242R CPU with 20 cores/40 threads and 768 GByte RAM. For each DSE, we executed the run five times in parallel on this machine.

We implemented the proposed approach as a combination of ASP and Python code. For the analysis and the evaluation of the experimental results, we used Python, C++, and Bash scripts. To evaluate the quality of the resulting system implementations, we utilized the ϵ-dominance metric [37]. Therefore, for each application, we constructed a reference front, consisting of the best solutions found up to the timeout from all cases (over all architecture templates or resource libraries, respectively). This reference front was considered as the optimal solution front for that specific instance. That way, we could evaluate the quality of all cases (all architecture templates for simulation one and all resource libraries for simulation two) with respect to the respective reference front.

5.2. Results

In our first simulation, we focused on examining the impact of varying topology template sizes and of differing application graph sizes.

In Figure 5a, we classified all DSE runs according to their outcome, i.e., whether the system synthesis problem was satisfiable (green), unsatisfiable (red), or whether the outcome was unknown (orange) due to a timeout during the grounding step (orange). Further, we distinguished whether it was possible to exhaustively explore the solution space in the given time (dark green) or whether the solving step was interrupted by a timeout (light green). Note that each DSE outcome classified as green contained at least one solution. In Figure 5b, we contrast the size of the given architecture template with the respective grounding size of the system synthesis problem, and that for all application graph sizes. Note that the black boxes indicate the exploration problems that were not groundable within one hour.

Both figures show a clear trend. As the size increases, regardless of whether referring to the topology or the application size, the grounding size, grounding time, and solving time are escalating, causing an hour to be insufficient to at least ground the given problem in some cases.

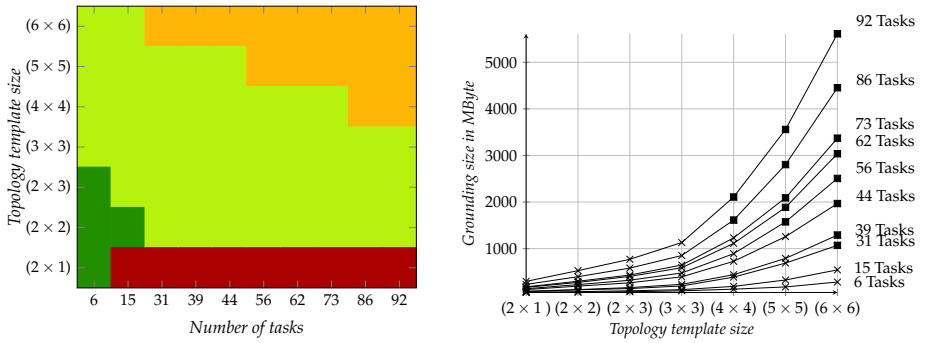


Figure 5. Results for simulation one, regarding an increasing topology template size. (**a**) Distribution of the DSE states after timeout (dark green—satisfiable, no timeout; light green—satisfiable, timeout; orange—unknown, timeout during grounding; red—unsatisfiable). (**b**) Development of the grounding size for differing topology template and application graph sizes.

Notably, the highest grounding size observed during the simulation reached 5611 MByte. This grounding bottleneck is a well-known issue of traditional ASP systems, following the ground-and-solve paradigm, when dealing with large-scale problems as they are typical for industrial design and planning problems [23].

To provide a further analysis, we selected one instance from the benchmark set to evaluate the grounding and solving times as well as the solution quality by means of the ϵ-dominance [37]. We decided on the smallest instance (with six tasks), because the solver successfully tackled all topology template sizes in combination with this instance. The results are shown in Table 2. The values present the average values out of five identical DSE runs (highlighted in gray) and the corresponding deviations.

Table 2. Comparison of the grounding and searching time and the quality of the discovered solutions for the smallest instance in simulation one. For the solving time, we distinguish the time up to the first solution found and the time required to exhaustively explore the solution space (in case of timeout, it is set to 3600 s, i.e, to one hour).

Case	Grounding	First Solution	Overall Search	$\epsilon < 1$	$\epsilon > 1$
(2 × 1)	0.23 s ± 3.02 %	0.00 s ± 0.00 %	0.25 s ± 2.78 %	1.00 ± 0.00 %	1.14 ± 0.00 %
(2 × 2)	1.01 s ± 1.70 %	0.00 s ± 0.00 %	3.96 s ± 1.01 %	1.00 ± 0.00 %	1.00 ± 0.00 %
(2 × 3)	2.39 s ± 5.79 %	0.00 s ± 0.00 %	767.55 s ± 3.68 %	1.00 ± 0.00 %	1.00 ± 0.00 %
(3 × 3)	5.59 s ± 6.36 %	0.07 s ± 0.00 %	3600.00 s ± 0.00 %	1.00 ± 0.00 %	1.00 ± 0.00 %
(4 × 4)	17.97 s ± 2.80 %	5.62 s ± 2.49 %	3600.00 s ± 0.00 %	0.82 ± 0.00 %	2.47 ± 0.00 %
(5 × 5)	44.26 s ± 3.43 %	10.34 s ± 4.72 %	3600.00 s ± 0.00 %	0.86 ± 0.00 %	2.01 ± 0.00 %
(6 × 6)	93.89 s ± 2.51 %	19.76 s ± 13.8 %	3600.00 s ± 0.00 %	0.73 ± 0.00 %	2.34 ± 0.00 %

We consistently observe the same trend as in Figure 5: as the topology template size increases, both grounding and searching times scale up as well. However, despite long grounding times, a first solution can be found quickly. Although some entries in Table 2 show a high deviation, most follow the general trend outlined before. Further, there is no deviation in the category of the ϵ-dominance, i.e., no varying Pareto-optimal design

points were discovered. This should be the case, since the DSE is deterministic, meaning that differences in the number and quality of solutions can only be caused by a variation in the search times.

A high-quality solution front, as a result of a DSE, approximates an ϵ-dominance value that equals one, which indicates that at least one Pareto-optimal design point from the reference front is covered [37]. It can be seen in Table 2 that this is the case for the architecture template sizes (2×1), (2×2), (2×3), and (3×3). Interestingly, in the fourth case, the design space was not exhaustively explored. However, there were two kinds of ϵ-dominance, approaching one from different sides. Only if both values equal one has the complete Pareto optimal front been found during the DSE [37], which was not the case, e.g., for the smallest architecture template (2×1). Since the search space was completely searched during this DSE, it means that this architecture was too small to allow all optimal application-specific system implementations. Regarding Figure 5a, this architecture size was also too small for the execution of any other application. Therefore, the design problem for all other applications was unsatisfiable.

At the same time, the architecture template size (2×2) always turned out to be sufficient in our tests. It is likely that the limited diversity of the chosen resource library, which contained four processing types, played a significant role, because one instance of each processing type could potentially get a grid placement. Further, in our simulations, we used applications with at most three completely independent sub-applications. Regarding real-world examples, as applications grow in complexity, and especially as specifications contain more independent sub-processes per application, the availability of additional redundant processing-type instances becomes crucial for parallelized workload execution. Therefore, we assume that for this workload type, larger topologies will lead to Pareto optimal solutions too, but we could not discover this in our simulation.

Analogously to Figure 5, we present in Figure 6 the results of the second simulation, regarding an increasing number of processing types in the resource library. For the grounding size in Figure 6b, we can identify the same trend as indicated before but weaker.

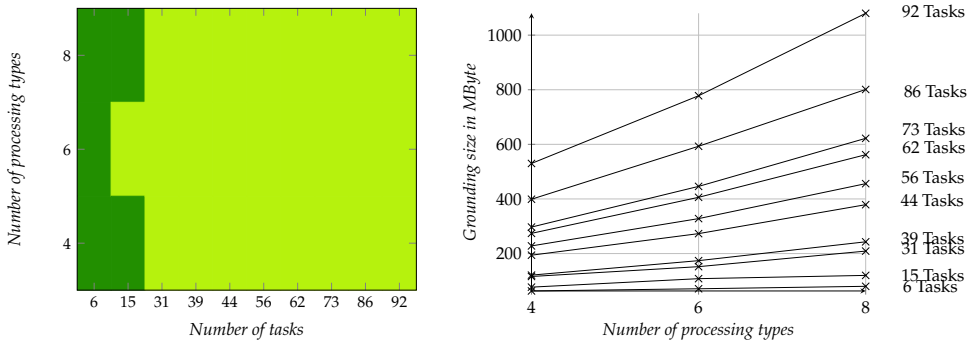


Figure 6. Results for simulation two, regarding an increasing number of processing types in the resource library. (**a**) Distribution of the DSE states after timeout (dark green—satisfiable, no timeout; light green—satisfiable, timeout; orange—timeout during grounding; red—unsatisfiable). (**b**) Development of the grounding size for various resource libraries and application graph sizes.

However, this trend cannot analogously be attributed to the difficulty of the synthesis problem. In Figure 6a, we can see that an increasing number of processing types in the resource library does not indicate that the problem is harder to solve. Instead, the difficulty is random, due to the randomly generated mapping options from tasks in the application to the processing types in each resource library.

In the end, we wanted to take a closer look at the generated application-specific architectures for one selected instance. As in Table 2, we chose the smallest application. We give the respective application graph, containing six tasks and six communication

messages (in black), in Figure 7a. Additionally, we present the mapping options of each task (in red).

Figure 7b compares the static and dynamic characteristics of the processing types from the respective, randomly generated four-type resource library. For each metric, we give the normalized values of all four processing types to the lowest, i.e., best value. Finally, in Figure 8, we visualize the results for the (2×1)-, (2×2)- and (2×3)-grid topology template, because these DSEs were finished before the timeout, meaning that the presented solutions were contained in the Pareto-optimal front for that instance. The generated architectures contained processing-, communication-, and interconnection-type instances, each labeled with a tuple consisting of the respective resource type and the ID. The ID of each instance was based on its implicit position numbering, as we explained in Section 4.3. Both Figures 7a and 8 were generated by utilizing the ASP-based visualization tool clingraph [38].

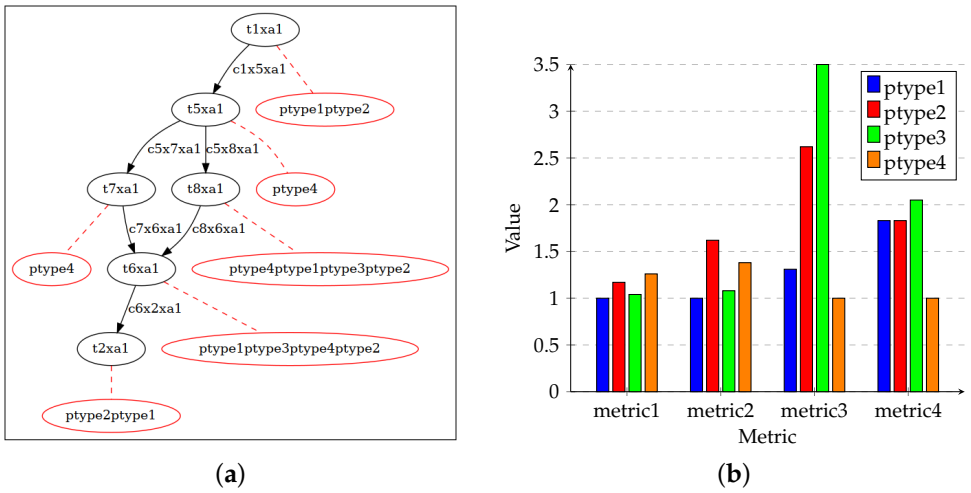


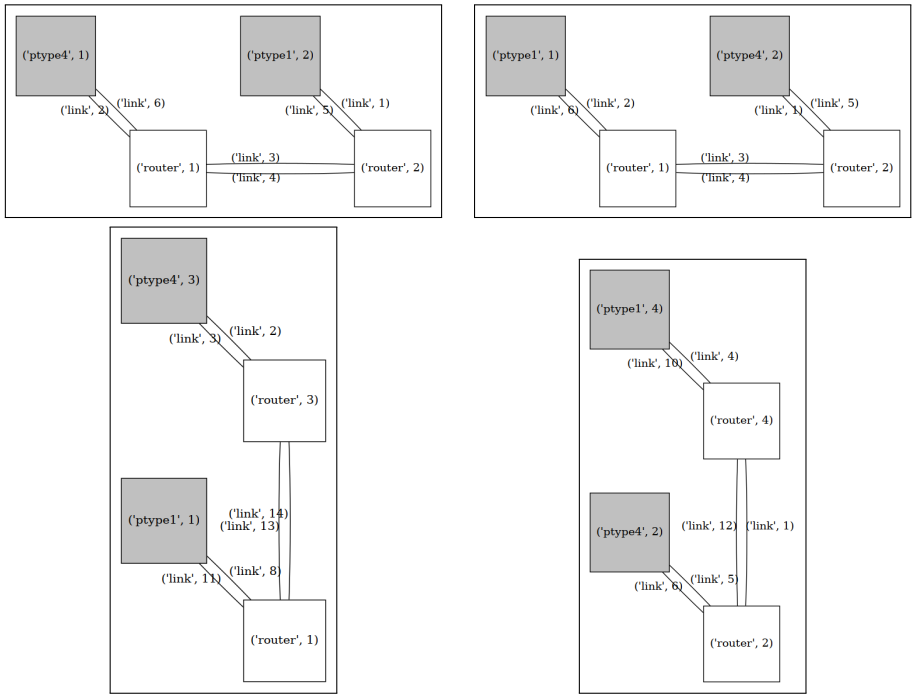
Figure 7. Presentation of aspects of the smallest benchmark instance. (**a**) The application graph, consisting of tasks and communications messages (black), and the mapping options (here, for each task, all processing types are summarized, each in one red node). (**b**) The characteristics of all available processing types normalized to the lowest value for the respective characteristic. The metrics include metric 1 = static resource cost, metric 2 = static energy consumption, metric 3 = average execution time for all tasks of the application, metric 4 = average dynamic energy consumption for all tasks of the application.

In Figure 7a, there are tasks like `t5xa1` or `t7xa1` with a mapping option to exactly one processing type, namely, `ptype4`, indicating that at least one instance of that processing type needs to be allocated in the final system implementation. But at least one other instance of a second processing type is required because the tasks `t1xa1` and `t2xa1` have no mapping option to processing type `ptype4`. As we can see in Figure 8, only the processing types `ptype1` and `ptype4` are selected in all solutions. The reason for this can be found in their properties in Figure 7b. Processing type `ptype3` and especially type `ptype2` turn out to be highly unattractive. Regarding the static characteristics, `ptype1` is the favorable option, whereas for the dynamic characteristics, it is `ptype4`.

Since conflicting objectives, like cost and latency, were optimized, the resulting hardware platforms varied in size. Nonetheless, due to the symmetrical architecture template, we discovered a lot of redundant solutions. Interestingly, even this small example provided a wide range of symmetry types. In Figure 8a, we can see mirrored and exchanged positions of the instances. This applies to the selected positions for the processing-type instances, as well as to the allocated communication-type instances. In Figure 8b, we could generate one solution by folding/unfolding the respective allocated architecture of the other solution.

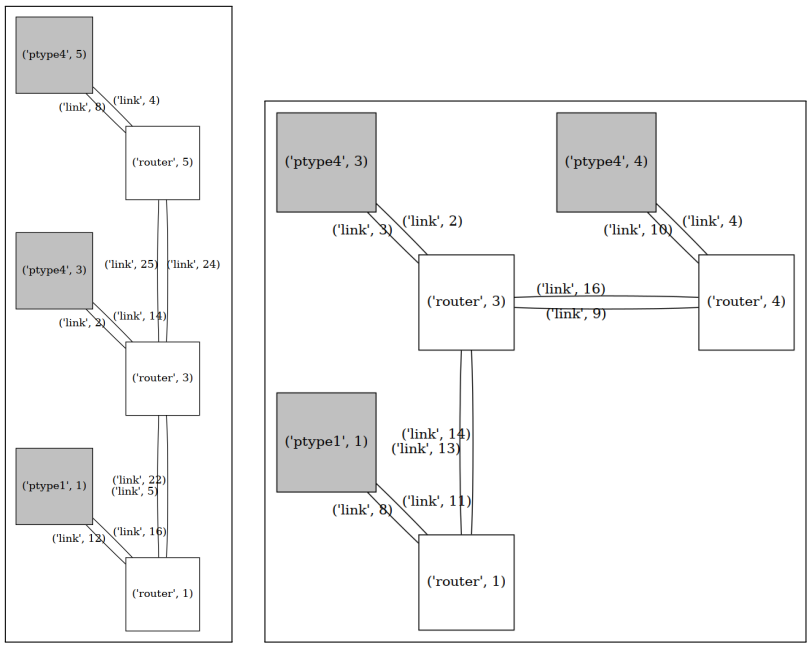
The problem complexity scales with the size of the problem instances, i.e., with the number of tasks, the size of the topology template and of the resource library. Our previous evaluations showed that even some medium-sized instances were not solvable within one

hour. But real-world examples can especially be extremely complex, and thus, the vast design space cannot be exhaustively explored in a reasonable time. In this case, when we stop the search after a certain timeout, we cannot know if the found solution is optimal or how much of the design space remains unexplored. Eliminating redundant solutions, and thus reducing the size of the feasible solution space is therefore essential for the DSE of real-world use cases. In that context, Goens et al. [39] identify redundant solutions for the task mapping problem on a grid-based architecture. They provide a framework for automatically identifying local and global symmetries using the concept of inverse semigroups. Due to the considered similar system model, we expect beneficial results when applying their methods to our approach. However, we have not yet conducted a further investigation in that direction.



(**a**)

Figure 8. *Cont.*



(**b**)

Figure 8. Generated architectures contained in the Pareto-optimal solutions. (**a**) Utilizing two processing-type instances. (**b**) Utilizing three processing-type instances.

6. Discussion and Conclusion

In the paper at hand, we presented a framework for designing multiprocessor systems at the ESL using ASP. In the beginning of the paper, we proposed the following research questions:

1. How can we efficiently encode a GDA at the ESL using ASP?
2. How well is this problem solvable for various problem sizes?
3. What might be challenges on the way to an unrestricted generative design at the ESL?

An answer to the first question was proposed in this paper. Compared to the state of the art, we do not require the hardware platform to be fully specified, thus reducing the specification overhead for a user while simultaneously increasing the flexibility for the DSE. Instead of a fixed architecture template, we introduce a resource library that describes all available processing, communication, and linking types. A designer merely supplies a model of an application to be executed on a final hardware architecture. Based on the GDA, our approach allows a fully autonomous DSE of a wide range of design alternatives for the final platform, thus tailoring the system to the needs of the application. This paper represents a first step on the way to a framework that will enable the fully unrestricted generation of a hardware platform. By providing only a template for the communication topology of the hardware platform, we manage to increase the flexibility of the design process while avoiding a severe explosion of the search space.

Moreover, it has to be noted that the individual implementation steps of allocation, binding, routing, and scheduling are more interwoven as the allocation now makes further topology decisions that influence binding, routing, and scheduling, accordingly. This results in a heavily application-specific system implementation. Overall, the encoding is modular and easily extensible, which provides a lot of flexibility to the end user while enabling the comparison of multiple objectives, in our case, latency, energy consumption, and area cost. Unfortunately, we were not able to compare the results to other work in this area. One reason is the lack of adequate system-level benchmarks, which is probably caused by the novelty of this approach. Furthermore, our specification differs in some cases significantly from other works in this area, which would render a comparison useless.

Regarding the second question, we were able to demonstrate through simulations the capability of our framework to produce optimal results. In detail, we showed the effectiveness and limits of our framework for a wide range of plausible parameters regarding the size of the target topology, the number of tasks, and the number of different available processing types. We also showed that there were limitations in the scalability of our approach. For some problem instances, the ASP-solver was not able to finish the grounding within one hour and in turn did not find any valid solution. This applied even to medium-sized instances, not covering real-world applications yet. Unfortunately, this is a well-known limitation of ASP-based systems [23] and needs to be addressed in future work.

Throughout the paper, we were also able to gain valuable insights regarding our third question. As mentioned before, one of the largest challenges faced during a fully flexible DSE is the scalability. At present, our approach is limited to a grid-based architecture platform. In the future, we want to further explore the possibility of loosening the constraints made during the DSE while still being able to solve the problem in a reasonable amount of time. However, since our approach exhaustively investigates all possible configurations of the system, the complexity is increasing with every additional degree of freedom. By enabling the free placement of resources in our topology template, we showed that symmetries could also pose a significant problem, which might be tackled by detecting redundancies inside our system model, as well as in our problem encoding, and excluding symmetric solutions from the feasible search space of the DSE.

Also, efficient clustering methods on the application side as well as the subdivision of the architecture platform into tiles, potentially containing sub-architectures, could help us cope with the enormous complexity. Furthermore, more emphasis has to be put into developing a more sophisticated resource library, which better represents the versatility of hardware components available for architecture generation. At the same time, we want to better adapt our framework to the customers' needs, e.g., by providing differently sized resource types, so that the exploration of an application-specific architecture for a given floor plan is conceivable. Using these ideas, we aim to enhance the flexibility in the design process of hardware/software systems even further to enable the designer to autonomously explore customized architectures based on the application requirements and thus to reduce the demands on the designer.

Author Contributions: Conceptualization, L.M., N.S., L.S., and C.H.; methodology, L.M., N.S., L.S., and C.H.; software, L.M.; writing—original draft preparation, L.M., N.S., and L.S.; writing—review and editing, L.M., N.S., L.S., and C.H.; visualization, L.M. All authors have read and agreed to the published version of the manuscript.

Funding: This work was funded by the German Science Foundation (DFG) under grant HA 4463/4-2.

Data Availability Statement: The implementation of our proposed approach as well as the benchmark instances and results from the experiments are provided at: https://github.com/lu-hub77/generative-design-space-exploration/releases/tag/v1.0.0 (accessed on 15 February 2024).

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Lukasiewycz, M.; Glass, M.; Haubelt, C.; Teich, J. Efficient Symbolic Multi-Objective Design Space Exploration. In Proceedings of the 2008 Asia and South Pacific Design Automation Conference, Seoul, Republic of Korea, 21–24 March 2008; IEEE: Piscataway, NJ, USA, 2008; pp. 691–696. [CrossRef]
2. Andres, B.; Gebser, M.; Schaub, T.; Haubelt, C.; Reimann, F.; Glaß, M. Symbolic System Synthesis Using Answer Set Programming. In Proceedings of the Logic Programming and Nonmonotonic Reasoning, Corunna, Spain, 15–19 September 2013; pp. 79–91. [CrossRef]
3. Neubauer, K.; Wanko, P.; Schaub, T.; Haubelt, C. Exact Multi-Objective Design Space Exploration Using ASPmT. In Proceedings of the 2018 Design, Automation & Test in Europe Conference & Exhibition (DATE), Dresden, Germany, 19–23 March 2018; IEEE: Piscataway, NJ, USA, 2018; pp. 257–260. [CrossRef]

4. Ishebabi, H.; Mahr, P.; Bobda, C.; Gebser, M.; Schaub, T. Answer Set versus Integer Linear Programming for Automatic Synthesis of Multiprocessor Systems from Real-Time Parallel Programs. *Int. J. Reconfig. Comput.* **2009**, *2009*, 863630. [CrossRef]
5. Lukasiewycz, M.; Sagstetter, F.; Steinhorst, S. Efficient Design Space Exploration of Embedded Platforms. In Proceedings of the 52nd Annual Design Automation Conference, San Francisco, CA, USA, 7–11 June 2015; pp. 1–6. [CrossRef]
6. Richthammer, V.; Fassnacht, F.; Glaß, M. Search-Space Decomposition for System-level Design Space Exploration of Embedded Systems. *ACM Trans. Des. Autom. Electron. Syst.* **2020**, *25*, 14. [CrossRef]
7. Goens, A.; Menard, C.; Castrillon, J. On the Representation of Mappings to Multicores. In Proceedings of the 2018 IEEE 12th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC), Hanoi, Vietnam, 12–14 September 2018; pp. 184–191. [CrossRef]
8. Weichslgartner, A.; Gangadharan, D.; Wildermann, S.; Glaß, M.; Teich, J. DAARM: Design-time Application Analysis and Run-Time Mapping for Predictable Execution in Many-Core Systems. In Proceedings of the 2014 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS), New Delhi, India, 12–17 October 2014; pp. 1–10. [CrossRef]
9. Schwarzer, T.; Weichslgartner, A.; Glaß, M.; Wildermann, S.; Brand, P.; Teich, J. Symmetry-Eliminating Design Space Exploration for Hybrid Application Mapping on Many-Core Architectures. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* **2018**, *37*, 297–310. [CrossRef]
10. Haubelt, C.; Müller, L.; Neubauer, K.; Schaub, T.; Wanko, P. Evolutionary System Design with Answer Set Programming. *Algorithms* **2023**, *16*, 179. [CrossRef]
11. Lukasiewycz, M.; Streubuhr, M.; Glass, M.; Haubelt, C.; Teich, J. Combined System Synthesis and Communication Architecture Exploration for MPSoCs. In Proceedings of the 2009 Design, Automation & Test in Europe Conference & Exhibition, Nice, France, 20–24 April 2009; IEEE: Piscataway, NJ, USA, 2009; pp. 472–477. [CrossRef]
12. Gradišar, L.; Klinc, R.; Turk, Ž.; Dolenc, M. Generative Design Methodology and Framework Exploiting Designer-Algorithm Synergies. *Buildings* **2022**, *12*, 2194. [CrossRef]
13. Li, H.; Lachmayer, R. Automated Exploration of Design Solution Space Applying the Generative Design Approach. In Proceedings of the Design Society: International Conference on Engineering Design, Delft, The Netherlands, 5–8 August 2019; Volume 1, pp. 1085–1094. [CrossRef]
14. Pioneering Bionic 3D Printing | Airbus. Available online: https://www.airbus.com/en/newsroom/news/2016-03-pioneering-bionic-3d-printing (accessed on 8 February 2024).
15. Generatives Design bei Airbus | Kundenprojekte | Autodesk. Available online: https://www.autodesk.de/customer-stories/airbus (accessed on 23 January 2024).
16. Khan, S.; Gunpinar, E.; Sener, B. GenYacht: An Interactive Generative Design System for Computer-Aided Yacht Hull Design. *Ocean Eng.* **2019**, *191*, 106462. [CrossRef]
17. Weaver, C.; Krishna, R.; Wu, L.; Austin, T. Application Specific Architectures: A Recipe for Fast, Flexible and Power Efficient Designs. In Proceedings of the 2001 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems, Atlanta, GA, USA, 16–17 November 2001; pp. 181–185. [CrossRef]
18. Gries, M. Methods for Evaluating and Covering the Design Space during Early Design Development. *Integration* **2004**, *38*, 131–183. [CrossRef]
19. Pimentel, A.D. Methodologies for Design Space Exploration. In *Handbook of Computer Architecture*; Chattopadhyay, A., Ed.; Springer: Singapore, 2022; pp. 1–31. [CrossRef]
20. Neubauer, K. Model-Based Symbolic Design Space Exploration at the Electronic System Level: A Systematic Approach. Ph.D. Thesis, University of Rostock, Rostock, Germany, 2021. [CrossRef]
21. Thompson, M.; Pimentel, A.D. Exploiting Domain Knowledge in System-Level MPSoC Design Space Exploration. *J. Syst. Archit.* **2013**, *59*, 351–360. [CrossRef]
22. Richthammer, V.; Scheinert, T.; Glaß, M. Data Mining in System-Level Design Space Exploration of Embedded Systems. In *Embedded Computer Systems: Architectures, Modeling, and Simulation*; Orailoglu, A., Jung, M., Reichenbach, M., Eds.; Springer International Publishing: Cham, Switzerland, 2020; Volume 12471, pp. 52–66. [CrossRef]
23. Falkner, A.; Friedrich, G.; Schekotihin, K.; Taupe, R.; Teppan, E.C. Industrial Applications of Answer Set Programming. *KI Künstliche Intell.* **2018**, *32*, 165–176. [CrossRef]
24. Blickle, T.; Teich, J.; Thiele, L. System-Level Synthesis Using Evolutionary Algorithms. *Des. Autom. Embed. Syst.* **1998**, *3*, 23–58. [CrossRef]
25. Todorov, V.; Mueller-Gritschneder, D.; Reinig, H.; Schlichtmann, U. Deterministic Synthesis of Hybrid Application-Specific Network-on-Chip Topologies. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* **2014**, *33*, 1503–1516. [CrossRef]
26. Li, Z.; Huang, J.; Xu, Q.; Chen, S. Integer linear programming based fault-tolerant topology synthesis for application-specific NoC. In Proceedings of the 2017 IEEE 12th International Conference on ASIC (ASICON), Guiyang, China, 25–28 October 2017; pp. 96–99. [CrossRef]
27. Murali, S.; Benini, L.; De Micheli, G. An Application-Specific Design Methodology for On-Chip Crossbar Generation. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* **2007**, *26*, 1283–1296. [CrossRef]
28. Thiele, L.; Chakraborty, S.; Gries, M.; Künzli, S. Design Space Exploration of Network Processor Architectures. *Netw. Process. Des. Issues Pract.* **2002**, *1*, 55–89. [CrossRef]

29. Lieverse, P.; Van Der Wolf, P.; Vissers, K.; Deprettere, E. A Methodology for Architecture Exploration of Heterogeneous Signal Processing Systems. *J. VLSI Signal Process. Syst. Signal Image Video Technol.* **2001**, *29*, 197–207. [CrossRef]
30. Ishebabi, H.; Bobda, C. Automated Architecture Synthesis for Parallel Programs on FPGA Multiprocessor Systems. *Microprocess. Microsyst.* **2009**, *33*, 63–71. [CrossRef]
31. Lahti, S.; Sjovall, P.; Vanne, J.; Hamalainen, T.D. Are We There Yet? A Study on the State of High-Level Synthesis. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* **2019**, *38*, 898–911. [CrossRef]
32. Bansal, S.; Hsiao, H.; Czajkowski, T.; Anderson, J.H. High-Level Synthesis of Software-Customizable Floating-Point Cores. In Proceedings of the 2018 Design, Automation & Test in Europe Conference & Exhibition (DATE), Dresden, Germany, 19–23 March 2018; pp. 37–42. [CrossRef]
33. Neubauer, K.; Haubelt, C.; Wanko, P.; Schaub, T. Systematic Test Case Instance Generation for the Assessment of System-level Design Space Exploration Approaches. In Proceedings of the Workshop Methoden Und Beschreibungssprachen Zur Modellierung Und Verifikation von Schaltungen Und Systemen, 2018; p. 10. [CrossRef]
34. Pasricha, S.; Dutt, N. *On-Chip Communication Architectures: System on Chip Interconnect*; Morgan Kaufmann: Burlington, MA, USA, 2010.
35. Kaminski, R.; Romero, J.; Schaub, T.; Wanko, P. How to Build Your Own ASP-based System?! *Theory Pract. Log. Program.* **2021**, *23*, 299–361. [CrossRef]
36. Lifschitz, V. *Answer Set Programming*; Springer International Publishing: Cham, Switzerland, 2019. [CrossRef]
37. Zitzler, E.; Thiele, L.; Laumanns, M.; Fonseca, C.; da Fonseca, V. Performance Assessment of Multiobjective Optimizers: An Analysis and Review. *IEEE Trans. Evol. Comput.* **2003**, *7*, 117–132. [CrossRef]
38. Hahn, S.; Sabuncu, O.; Schaub, T.; Stolzmann, T. Clingraph: ASP-Based Visualization. In Proceedings of the Logic Programming and Nonmonotonic Reasoning, Genova, Italy, 5–9 September 2022; Springer International Publishing: Cham, Switzerland, 2022; pp. 401–414. [CrossRef]
39. Goens, A.; Siccha, S.; Castrillon, J. Symmetry in Software Synthesis. *ACM Trans. Archit. Code Optim.* **2017**, *14*, 20. [CrossRef]