Cost Estimation for Configurable Model-Driven SoC Designs Using Machine Learning

Lorenzo Servadei[1,2], Edoardo Mosca[3], Keerthikumara Devarajegowda[4], Michael Werner[3], Wolfgang Ecker[1,3], Robert Wille[2,5]
[1]Infineon Technologies AG: name.surname@infineon.com
[2]Johannes Kepler University Linz: name.surname@jku.at
[3]Technical University Munich: name.surname@tum.de
[4]University of Kaiserslautern
[5]Software Competence Center Hagenberg GmbH (SCCH)

ABSTRACT

The complexity of today's *System on Chips* (SoCs) forces designers to use higher levels of abstractions. Here, early design decisions are conducted on abstract models while different configurations describe how to actually realize the desired SoC. Since those decisions severely affect the final costs of the resulting SoC (in terms of utilized area, power consumption, etc.), a fast and accurate cost estimation is essential at this design stage. Additionally, the resulting costs heavily depend on the adopted logic synthesis algorithms, which optimize the design towards one or more cost objectives. But how to structure a cost estimation method that supports multiple configurations of an SoC, implemented by use of different synthesis strategies, remains an open question. In this work, we address this problem by providing a cost estimation method for a configurable SoC using *Machine Learning* (ML). A key element of the proposed method is a data representation which describes SoC *configurations* in a way that is suited for advanced ML algorithms. Experimental evaluations conducted within an industrial environment confirm the accuracy as well as the efficiency of the proposed method.

CCS CONCEPTS

• **Hardware** → *Hardware-software codesign.*

KEYWORDS

Machine Learning, Design Automation, Hardware-Software co-design

1 INTRODUCTION

The ever-increasing complexity of today's *System on Chips* (SoCs) leads to tremendous challenges for their design and implementation. In order to tackle this complexity, designers rely on abstractions. After the gate level, the *Register Transfer Level* (RTL), and the *Electronic System Level* (ESL) have been established in the past, now also model-driven design flows find application in industrial practice. Here, an abstract specification of the system to be realized is provided and hardware generation frameworks [4, 6, 11] are utilized to create the desired design out of it. This does not only allow for an easier implementation of SoCs, but also creates more potential for design exploration since more design variants can be considered through model-driven design. Following this design flow requires the designer to define, e.g., the number of components to be used, their attributes, as well as their structure and interplay between each other (e.g. bus specifications, internal ports, etc.) —resulting in a *configuration* that describes how to realize the SoC from a model. The corresponding design decisions will have a severe impact on the costs of the resulting SoC (e.g., with respect to area or power consumption). Hence, in order to elaborate on whether a determined configuration helps in satisfying given design objectives (such as staying below a certain power consumption), designers need information on how these decisions will eventually affect the costs. To this end, *cost estimation* is essential. Unfortunately, the complexity of the design as well as the the interdependencies and interplay between its components makes it difficult to accurately estimate the resulting costs. The complexity of the estimation increases even more, when different logic synthesis strategies are applied to the design [10]. Logic synthesis algorithms, in fact, transform an RTL design into a gate level representation, so that one or more costs is optimized. To date, the typical way to determine accurately what costs a configuration will have is to completely generate the code, synthesize it, and simulate the resulting design—a time-consuming process particularly when several different design choices (i.e., configurations) should be explored. As an alternative, *Machine Learning* algorithms have been proposed in the literature for this purpose. For example, in the work of [12], ML algorithms in the form of *Convolutional Neural Networks* (CNNs) are used for estimating the area reduction caused by changes in specific parameters of the SoC. Others focus on problems of distinct hardware components such as GPUs [24] or memory systems [20, 21, 25]. Furthermore, the approach presented in [9] proposes a cost estimation method for accelerators of heterogeneous systems. However, although these works are valuable contributions for approaching the complexity of cost estimation, they either present application-specific methods (such as [12] with its focus on application-dependent parameters) or focus on the configuration of a restricted set of devices (such as GPUs [24], memory systems [20], or accelerators of heterogeneous systems [9]). But SoCs are frequently used and, hence, designed in a broader variety which is not supported by these previous works. Additionally, the related work does not take into account different strategies of synthesis, which clearly contribute to the final cost in terms of area, power, etc. Therefore, a cost estimation that supports the configurability of an SoC and its components together with different synthesis strategies is of key importance for the research. Last, although High Level Synthesis tools are an alternative for area/power/performance estimation, they do work with higher level languages and not with low-level details of the design (i.e. not at RTL level). In this work, we propose a cost estimation method which addresses this drawback. To this end, a data representation for SoC components is proposed which represents SoC *configurations* and is suited for advanced ML algorithms. The proposed approach well-adapts to the SoC components considered during the model-driven design flow and, hence, is applicable to various design configurations implemented by use of different logic synthesis algorithms. Experimental evaluations conducted within an industrial environment confirm the accuracy of the proposed approach (R^2 score of 0.91 on the number of Lookup Tables, of 0.93 on the number of Slice Registers, and of 0.92 on the power consumption). At the same time, the costs can be estimated in negligible run-time,

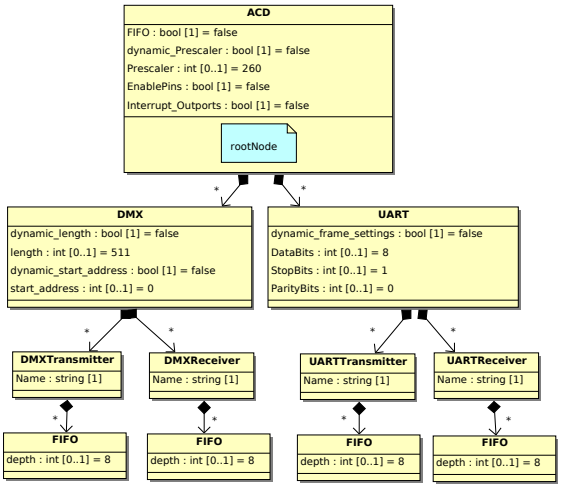


Figure 1: Model for ACD components

i.e., within few seconds only (whereas the current state of the art requires hours). Overall, this yields a fast and accurate cost estimation which is applicable in an industrial contexts for multiple SoC configurations. The remainder of this paper is structured as follows: The next section reviews the considered model-driven design flow and discusses the resulting challenge of cost estimation. Section 3 then presents the general idea of the proposed approach followed by Section 4 describing its implementation. Finally, Section 5 summarizes the obtained results and Section 6 concludes the paper.

2 BACKGROUND AND MOTIVATION

In order to keep the work self-contained, this section first briefly reviews the model-driven industrial design flow for SoCs as considered in this work. Afterwards, we discuss the resulting challenges that originate from this flow.

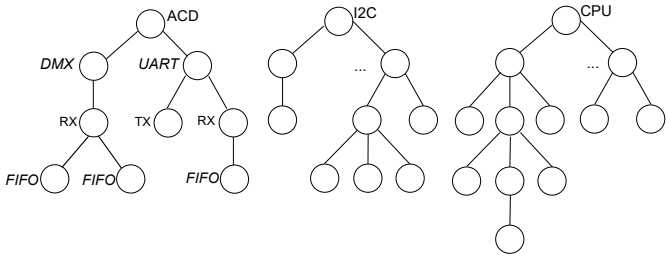
2.1 Model-driven SoC Design

In order to increase the productivity of the design of SoCs, hardware generation frameworks are utilized [1, 6, 22]. They receive an abstract specification of the system to be realized (usually a model provided, e.g., in terms of UML/OCL [19] or SysML [7]), which remains flexible enough so that still different instantiations are possible. In the following, we illustrate this flow by means of the *MetaRTL framework* [6]—a hardware generation framework for configuring and generating hardware designs which follow the RISC-V architecture [23] and, nowadays, is heavily applied in industrial contexts.

Example 1. *Consider the design process of a system composed of*

- *processing units (denoted* CPU*) which allows to execute software as well as*
- *synchronous communication components (denoted* I2C*) and*
- *asynchronous communication devices (denoted* ACD*).*

In the beginning of the design process, meta models are provided that describe the abstract structure of these SoC components and that can be instantiated in various fashions. For example, the meta model for the ACD is shown in Fig. 1 and describes the specification of asynchronous communication peripherals [5, 14]. Here, each node of the model defines the attributes of the component; relations and cardinalities define their structure (e.g., each ACD node may have one or more UART node defining the Universal Asynchronous Receiver/Transmitter*).*



TX: Transmitter, RX: Receiver

Figure 2: Configuration for an SoC

Having a model, it is up to the designer to instantiate it according to his/her respective needs and requirements. To this end, a *configuration* is defined which describes a complete instantiation of the model. The configuration is thereby organized in a hierarchical fashion: First, the designer gets to specify general attributes of the device as well as how many components shall be instantiated in total and how they should be connected (defining the *macro-structure*). Afterwards, the precise instantiation of each component is specified (defining the *micro-structure*, i.e. the attributes characterizing the single components). The resulting configuration is eventually described in a hierarchical and recursive structure, which can be visualized as a tree-like graph.

Example 2. *Consider again the design of the SoC with its ACD component. The designer first instantiates the attributes of the root node as well as the desired number of subcomponents and their structure—defining the macro-structure of the component (e.g., that the ACD component realizes FIFO buffers by utilizing either UART, DMX, or both). Afterwards, he/she will select the attribute of the UART nodes and/or DMX nodes and, recursively, all instantiated subcomponents. Each one of the instantiated child nodes are given specific attribute values describing their properties (e.g., each UART will have a specific number of parity bits, etc.). This procedure is continued until a complete instance of the component is specified. Overall, this results in a configuration which can be described in terms of a tree-like graph as shown in Fig. 2 for an entire SoC. Note that, due to space constraints, Fig. 2 only sketches the macro-structure. Each node in Fig. 2 eventually also includes a micro-structure defining the respective attribute values.*

After choosing the configuration, hardware generation frameworks (here: the MetaRTL framework) can be utilized to generate the desired code (e.g., in VHDL, SystemVerilog, etc.). Once the code is available, the resulting device can be synthesized (e.g., using tools such as *Vivado*[1]) and eventually realized.

2.2 Resulting Challenge: Cost Estimation

The model-driven design flow as sketched above is a useful methodology in order to tackle the ever increasing complexity of SoC design. However, it also requires the designer to make design decisions early in the process that severely impact the costs of the resulting SoC. In fact, defining, e.g., the number of components and their attributes obviously affects the area and the power consumption of the resulting chip. Moreover, also their structure and interplay between each other have significant impact to the eventual costs. Finally, the different fashions in which synthesis can be conducted (usually encapsulated through several optimization

[1]For further information: https://www.xilinx.com/products/design-tools/vivado.html

procedures in synthesis tools that can be applied in various combinations at different levels of abstraction) play an important role. Overall, the eventual costs of a design obtained from a model as well as a respectively chosen configuration depend on a complex interplay between various factors which are infeasible to explicitly estimate anymore.

Example 3. *Consider again the running example from above together with two different configurations for the ACD component: The first configuration realizes it with a* UART *component with 9* DataBit*s, 1* StopBit*, and 0* ParityBit*s; the second one uses a* UART *component with 6* DataBit*s, 1* StopBit*, and 1* ParityBit*. Since a larger bitstream requires a broader* FIFO*, the first configuration likely yields more total area than the second one. While this is easy to estimate, other aspects are harder to predict. For example, the second configuration requires specific hardware for the control and calculation of the parity checks (not needed in the first configuration, since no parity bits are employed). In a similar fashion, power consumption heavily depends on the size of the bitstream as well as the needed storage capabilities. Considering that, additionally, the structure of the components (and the entire SoC) as well as attribute values and relations of all other components also contribute to the resulting costs, it becomes infeasible to explicitly predict the costs "caused" by choosing a particular configuration.*

To date, the only accurate way to know what a configuration costs is to completely generate the code, synthesize it, and simulate the resulting design[2]—a time-intensive process particularly when several different design choices (i.e., configurations) should be explored. For example, the generation, synthesis, and simulation cycle of an SoC composed of a CPU, an ACD, and an I2C takes approximately 4-5 hours on average in our industrial environment. Considering that usually several configurations have to be estimated until one is met which satisfies certain objectives on costs such as area or power consumption, this quickly becomes infeasible. Hence, a severe challenge within the context of model-driven design is how to efficiently determine proper estimates on the costs of a configuration.

3 GENERAL IDEA OF THE PROPOSED COST ESTIMATION

Determining the costs of an SoC device obtained from a model and a corresponding configuration prior to its synthesis and implementation is a non-trivial task. The eventual costs are caused by a huge number of factors such as the number of components, their attributes and structure, as well as various optimization schemes. This leads to an enormous amount of factors to consider which makes it infeasible to aim for a deterministic cost function. Instead, we propose a method that rests on the principles of Machine Learning. Machine Learning has shown to be able to address the problem of estimation over complex functions in various areas such as detection of objects from images/videos (i.e. object detection, see [28]), extracting the sentiment expressed in user generated text (i.e. sentiment analysis, see [13]), or the translation of sentences into different languages (i.e. machine translation, see [26]). In this section, we describe how these principles can be employed for cost estimation of a configurable SoC. To this end, we first review the main advantages of ML. Afterwards, we sketch the general idea how to utilize that for the problem considered here.

[2]Note that, as discussed in Section 1, approaches proposed in [9, 12, 20, 24] are not applicable since they focus on a dedicated application or a restricted set of devices but not a configurable SoC.

3.1 Machine Learning Approaches

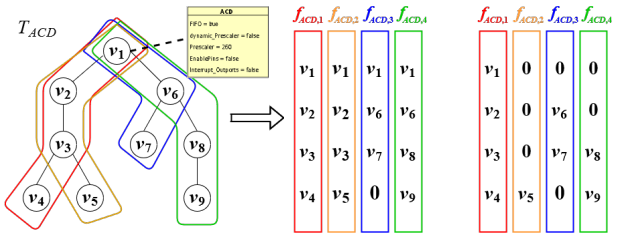
Machine Learning (ML) is a form of *Artificial Intelligence* (AI) that is able to perform a certain task without being specifically programmed for it. This characteristic makes it extremely useful each time the task involves too complex rules for being performed in a deterministic manner. Furthermore, in *parametric* ML algorithms, the algorithm can be executed in a time-efficient fashion [16]. Typically, a ML model learns from previous examples related to the specific task, during a process called training. In this work, we utilize the concept of supervised learning [16], since supervised learning algorithms are trained using labeled examples, i.e., an input where the desired output is known. Here, information from the training data is captured as association between *training samples* (denoted as inputs $\{\mathbf{x}^i\}_{i=1}^N$) and *ground truth values* related to each sample (denoted as labels $\{\mathbf{y}^i\}_{i=1}^N$). This forms a set $\{(\mathbf{x}^1, \mathbf{y}^1), \ldots, (\mathbf{x}^N, \mathbf{y}^N)\}$, where each $\mathbf{x}^i = (x_1^i, x_2^i, \ldots, x_p^i)^T$ is a vector of values of dimension p for each training sample and where N corresponds to the number of samples used for training the ML algorithm [17]. The information contained in the vector of feature measurements (usually called *feature vector*) has therefore a primary role in guiding the association learned by the ML algorithm. This raises the issue, how to structure this vector (i.e. the *data representation*) for a particular ML algorithm. In the past, this has been heavily investigated and led to particular innovations in data representation yielding significant improvements in the performance of ML algorithms. Examples include, e.g., multi-scale and augmented images for object detection [2], character level encoding for sentiment analysis [27], and *Contextualized Word Vectors* (CoVe) for machine translation [15]. Hence, determining a proper data representation (which is in line with a corresponding ML algorithm) is key for a successful utilization of ML for a particular application—especially, when a generic solution is desired. Accordingly, a ML-based solution for cost estimation of SoCs heavily depends on how to represent an SoC configuration in terms of such vectors.

3.2 Data Representation

An SoC may be composed of several components (e.g., a CPU, I2C, or ACD as illustrated in Example 1). Formally, those components can be represented as a set of trees, i.e., a set $SoC = \{T_i\}_{i=1}^r$ assuming that r components can be instantiated inside the SoC. Each tree can be represented as a set of interconnected nodes, i.e., $T_i = \{\mathbf{v}_{i,j}\}_{j=1}^s$ with $s = |T_i|$ being the total number of nodes of the tree T_i. Each node represents a subcomponent, as for instance the subcomponent *UART* or *DMX* are nodes of the *ACD* tree (see, e.g., Fig. 2). As each one of the nodes of the tree can be instantiated with different attribute values, each node is defined by a set of configurable attributes, i.e., $\mathbf{v}_{i,j} = \{c_{i,j,k}\}_{k=1}^d$ with $d = |\mathbf{v}_{i,j}|$. Setting these attributes as well as defining the connections between the nodes will eventually describe a configuration of the SoC.

Example 4. *Consider again the SoC example from above together with the configuration sketched in Fig. 2. This configuration can be represented by a set of trees* $SoC = \{T_{ACD}, T_{I2C}, T_{CPU}\}$*, where, e.g., the ACD tree is represented by a set of connected nodes, so that* $T_{ACD} = \{\mathbf{v}_1, \mathbf{v}_2, .., \mathbf{v}_9\}$ *(as shown in Fig. 3a). Here,* $\mathbf{v}_1$ *corresponds to the rootnode,* $\mathbf{v}_2$ *to a* DMX *node, etc. Further, each node provides values to its corresponding attributes (e.g., a* DMX *node could have the value* c_1 *corresponding to* dynamic_length = true, c_2 *being* dynamic_start_address = false*, etc.).*

However, as discussed above, ML approaches usually process data in terms of feature vectors. Accordingly, the tree representation has to be transformed into a vector representation. In this work, we are proposing doing that by considering each path of the tree from the root node to the maximum depth reached by its leaves.



(a) Configuration tree of an ACD component **(b) Possible feature vectors** **(c) Proposed feature vectors**

Figure 3: ACD component tree and alternative feature vectors

Then, each one of these paths represents a set of nodes at a different hierarchical level, thus preserving the spatial information within the tree. The nodes themselves again also include the respective attribute values. This way, all the information needed for structuring each feature vector used by the ML approach is available.

Example 5. *Consider again the formal representation of a component configuration as shown in Fig. 3a. Alternative vector representations of that are shown in Fig. 3c and Fig. 3b.*

Of course, mapping the given configuration (provided in terms of trees) to the required feature vector representation can be conducted in different fashions (as shown in Fig. 3). Furthermore, also the precise architecture of the ML approach has a severe impact on the accuracy of the eventually obtained results. Hence, the next section describes in more detail how the corresponding data representation is generated and how the ML algorithm is afterwards applied to obtain a fast and accurate cost estimation from it. The resulting solution will eventually be applicable to various configurations and, hence, a configurable SoC design.

4 IMPLEMENTATION

In this section, we describe in more detail how the proposed approach represents each configuration as a set of feature vectors. Based on that, we then describe how we select the ML algorithm and how it is exactly designed.

4.1 Generating the Data Representation

Given a configuration in terms of a tree representation, a translation to feature vectors has to be conducted. As sketched in Section 3.2, this can be done by considering all paths of the tree from the root node to the leaves. However, explicitly generating corresponding vectors can be conducted in different fashions.

For example, one way is to consider each path separately and create a set of vectors where each vector represents *all* nodes of the respective path. This could be easily implemented using a reverse order depth first traversal. However, this would lead to a description in which single nodes occurs in more than one vector—even if it is only instantiated once. This can easily lead to wrong associations in the ML algorithm and, consequently, wrong cost estimations, since it is not uniquely represented anymore whether a node indeed has been instantiated more than once or simply is part of more than one path. In order to avoid this, we employ a modified reverse order depth first traversal in which, (1) each time the traversal backtracks, a new feature vector is instantiated and (2) values of the vectors are initialized with zero up to the currently considered backtrack level. The following example illustrates the respective concepts.

Example 6. *Consider again the SoC configuration represented by the tree shown in Fig. 3a. Following the reverse order depth first traversal and representing each path by its own vector leads to a representation as shown in Fig. 3b. Here, the node $\mathbf{v}_1$ occurs a total of four times in the representation—even if it is only instantiated once. An ML algorithm may likely derive wrong associations and, hence, wrong cost estimations. In contrast, the modified reverse order depth first traversal leads to a representation as shown in Fig. 3c. Here, every node only occurs as often as it is indeed instantiated. Additionally, the structure and the dependencies are still represented in a fashion which is suitable for ML algorithms.*

Following this method, a set of *feature vectors* $\{\mathbf{f}_{i,l}\}_{l=1}^{s'}$ is generated from each SoC configuration represented by a tree T_i, where s' is the number of feature vectors, or equivalently the number of paths, for the component T_i (therefore in general $s' \leq s$ holds). Using Fig. 3c, we can visualize how the algorithm for creating the feature vectors works. Starting from the root node of the tree T_{ACD}, the algorithm iteratively pushes the nodes of each path in a corresponding vector $\mathbf{f}_{ACD,l}$—creating a set of vectors $\{\mathbf{f}_{ACD,l}\}_{l=1}^{4}$ representing the tree. Once the algorithm encounters common nodes in paths, it substitutes the attribute values of the common nodes iteratively with a vector of values zero (denoted by (0)), in order to avoid attribute repetition (and so, additional impact on the SoC cost) in the ML estimation of objectives.

Overall, this method allows to represent SoC configurations in a fashion suitable for ML algorithms. In fact, a faithful representation is guaranteed because each vector preserves the spatial information of its path and, by this, the hierarchy of the nodes. For example, the first attribute values in the feature vectors shown in Fig. 3c correspond to the attribute values of the root node in the SoC configuration shown in Fig. 3a. Similarly, the hierarchy and, hence, spatial information, of all other nodes is preserved. Furthermore, as pointed out in Fig. 3c, the order of each feature vector in the set determines common paths of the feature vectors, representing uniquely the structure of the tree. Hence, when employing the ML algorithm afterwards, this information is properly represented. Last, the linear space and time complexity of the depth first traversal in the number of nodes [3], allows the proposed method to scale easily to components with more feature vectors/hierarchical levels.

4.2 Applying the Machine Learning Algorithm

Having the representation as described above eventually allows to utilize recent ML approaches for the task of cost estimation considered here. To this end, we generated a representative amount of hardware configurations (i.e. the *dataset*). These contain mixed type of variables, where we normalize numerical values. These configurations are our method's *Primary Inputs*. Furthermore, in order to train our network to estimate the hardware objectives under different synthesis strategies (the logic optimization synthesis applied by the synthesis tool, i.e., *Area Optimization*, *Area Optimization*, or *Power Optimization*), we add an *Auxiliary Input* (denoted as *Conditioning*, similarly to [18]) which indicates the desired synthesis strategy. Since we are focusing on area and power consumption, the outputs of the ML estimation are *Lookup Tables* (LUTs) ($\mathbf{y}_1$), *Slice Registers* (SRs) ($\mathbf{y}_2$), and *Power Consumption* ($\mathbf{y}_3$). The labels for the learning process can be easily retrieved from the reports generated during the synthesis. Additionally, if the designer is interested in other metrics/objectives, the method can be easily adjusted, modifying the outputs metrics of the ML estimation. Then, the resulting data set is passed to a multi-output regression *Convolutional Neural Network* (CNN) that uses convolutions and fully connected layers. As shown in [8], convolutions are able to use the spatial information contained in the data (in our case, in the feature vectors). Processing the information by a shared set or parameters (i.e. *filters*) considerably reduce the number of parameters that the neural network has to learn (as pointed out it [8]). In this way, the parameters used for processing each path can be shared by each

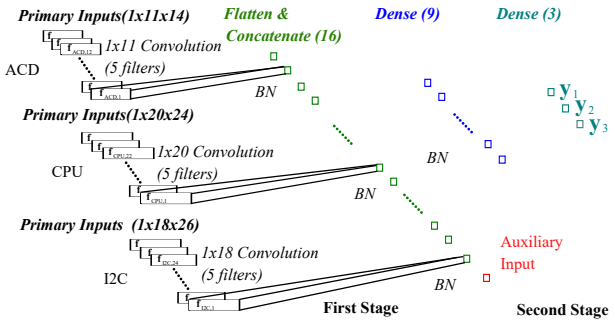


Figure 4: Neural network structure

feature vector. As a consequence, adding new feature vectors to a component (e.g. ACD, CPU, etc.) does not result in an increase of network parameters.

Fig. 4 sketches the correspondingly used architecture of the CNN. The algorithm is composed of two stages: In the first stage, the input configurations (i.e., the Primary Inputs) of the SoC components are processed in parallel by 1-dimensional (1D) convolutions with ReLU activation functions in three separated branches that do not interfere with each other. The width of the convolution filter is extended to the whole feature vector, so that each hierarchical level of the vector is processed by different parameters. The outputs of the first stage and the Auxiliary Input (i.e. Conditioning) discussed above are brought to a common format with a flatten layer and concatenated. The whole stack is then forwarded to the second stage and processed by fully connected layers to compute predictions for our set of outputs $\{\mathbf{y}_i\}_{i=1}^3$. As we can see, the three components are processed separately in the first stage because their feature vectors have different sizes and we want to use the spatial information contained in the feature vectors of each single component (e.g. ACD, CPU, etc.) before bringing them to a common format and process them together. Since the input of the second stage is composed of outputs from different branches of the first stage that can have a very different scale, we used *Batch Normalization* (BN) layers to maintain a non-skewed distribution and smoothen the training process [8]. Overall, this yields to a rather fast and efficient cost estimation for a configurable SoC as also confirmed by evaluations summarized in the next section.

5 EXPERIMENTAL EVALUATION

The proposed approach for cost estimation as described above has been implemented within our industrial context.[3] Afterwards, we evaluated the accuracy of the resulting method and compared it to the currently available approach for cost estimation (i.e., generating the code, synthesizing it, and simulating the resulting design as reviewed in Section 2.2). In this section, we summarize the results obtained from this comparison. To this end, we first describe the used environment, discuss the resulting accuracy and, finally, present application results.

5.1 Used Environment

As software development environment, Tensorflow-GPU v1.0.1 and Python v3.6 has been used. Hyperopt has been utilized for retrieving the optimal architecture of the network. For optimizing the training process with an Nvidia GPU, the CUDA Toolkit 9.0 and cuDNN v7.0 has been employed. As hardware, we used the Nvidia Tesla P100 for training the ML algorithm (running on an system with an Intel Core i7-8700K CPU and a DIMM 16GB DDR4-3000 module of RAM). The actual costs of a configuration (w.r.t. the number of LUTs, the number of SRs, and the Power Consumption) have been obtained from reports generated by the Vivado v2015.1 design tool running on three Intel Xeon CPU E5-2690 v2 machines, and applied to an Arty-7 FPGA board from Xilinx.[4]

Using this setup, we generated a total of 7512 different configurations of an SoC composed of a CPU together with ACD and I2C communication devices, taken from a predefined design space. For doing so, variations in the number and attribute values of the above mentioned SoC are considered. Additionally to configurable devices, the SoC is provided with a non-configurable interrupt handler, a *Serial Peripheral Interface* (SPI) and a data bus. Through a data generator, we realized instances of the components in the SoC taken from a space defined by the designer's experience. We then generated code for each configuration and implemented it using logic synthesis algorithms available on Xilinx Vivado. In particular, we applied the *Area High Optimization*, *Area Medium Optimization*, and *Power Optimization* on respectively one third of the dataset, each.

5.2 Resulting Accuracy

Using the environment described above, the accuracy of the proposed cost estimation method has been considered in a first evaluation. To this end, we obtained the labels, i.e., ground truth values, for the given objectives from the above-mentioned reports generated by the Vivado tool. As objectives, we considered the number of LUTs (labeled $\mathbf{y}_1$), the number of SRs (labeled $\mathbf{y}_2$), and the power consumption (labeled $\mathbf{y}_3$). For the evaluation, the data has been split into a training (70%), a validation (15%), and a test set (15%). The amount of 7512 configurations was selected, because adding further SoC configurations did not improve the accuracy of the estimation algorithm, even considering a larger ML model. Furthermore, we did not consider any different data split once we saw that our model was not overfitting the training samples. The selected model was trained for $3h15m$ on the configurations dataset.

Table 1 provides the characteristics of the dataset in terms of mean values ($\{\mu_{\mathbf{y}_i}\}_{i=1}^3$), range for each objective, and accuracy results. The latter clearly shows the high accuracy obtained through the ML estimation. In fact, an R^2 score of 0.91 on the numbers of Lookup Tables, of 0.93 on the number of Slice Registers, and of 0.92 on the power consumption (measured in Watts) are reported. These values show how the spatial information captured by the feature vectors can be efficiently processed by the CNN, leading to very good results for the estimation.

5.3 Application

Finally, a brief summary on the application perspective is provided. After the evaluation on the accuracy yielded very good results (as discussed above), the resulting cost estimation tool has been applied in the production process. Here, the costs of different configurations for an SoC design including a CPU, an ACD, and I2C as well as a non-configurable interrupt handler, an SPI, and a data bus have been estimated using the proposed method. For reasons of comparison, we repeated the cost estimation process using the previously applied Vivado flow (reviewed in Section 2.2).

Table 2 shows the obtained results. Here, the cost estimations of three different SoC configurations are listed. After generating RTL code for the different configurations, we ran a synthesis on it, using for each configuration a different synthesis strategy. In particular, Config 1 has been realized by use of the *Area High Optimization*, while Config 2 by use of the *Area Medium Optimization* and Config 3 by use of the *Power Optimization* algorithm for logic synthesis. The columns denoted *(Est.) Costs* provide values for the costs (w.r.t. $\mathbf{y}_1$,

[3] Due to the double-blind review process, we omit details about this industrial context. We will, however, add this information in a possibly final version of this paper.

[4] The documentation on Vivado and generated reports can be found on the *Vivado Design Suite User Guide – v2015.1*.

Table 1: Dataset and obtained accuracy

Dataset Mean	$\mu_{y_1} = 15359$ $\mu_{y_2} = 23627$	$\mu_{y_3} = 0.28$ W
Dataset Range	$y_1 \in [4991, 39608]$ $y_2 \in [2736, 86805]$	$y_3 \in [0.17, 0.55]$ W

	Area	**Power Consumption**
R^2 **Score**	$R^2_{y_1} = 0.91$ $R^2_{y_2} = 0.93$	$R^2_{y_3} = 0.92$

The R^2 *Score* provides the explained variance w.r.t. the number of LUTs ($R^2_{y_1}$), SRs ($R^2_{y_2}$), Power Consumption ($R^2_{y_3}$).

Table 2: Results obtained from an application perspective

SoCs		Vivado Approach		Proposed Approach	
		Costs	Time	Est. Costs	Time
Config 1	y_1	9271 LUTs	3h 20m	9572 LUTs	4.1s
	y_2	13042 SRs		15786 SRs	
	y_3	0.290 W		0.277 W	
Config 2	y_1	27467 LUTs	5h 13m	26740 LUTs	6.2s
	y_2	50336 SRs		49175 SRs	
	y_3	0.404 W		0.420 W	
Config 3	y_1	20071 LUTs	4h 46m	23793 LUTs	5.3s
	y_2	38834 SRs		39480 SRs	
	y_3	0.322 W		0.301 W	

y_2, and y_3, i.e., LUTs, SRs, and power consumption measured in Watts) obtained from the Vivado flow and the proposed method. The columns denoted *Time* provide the time needed to obtain these results.

Of course, Vivado provides ground truth values as the respective configurations are indeed completely realized in these cases. But, as can be clearly seen in Table 2, this comes at huge costs w.r.t. runtime. Each determination requires more than three hours—certainly too long for an initial design space exploration. In contrast, the proposed cost estimation flow can determine proper values in just a few seconds. At the same time, the resulting cost values remain very close to the actual ground truth values. Hence, this provides an efficient yet rather accurate alternative for cost estimation.

6 CONCLUSION

In this work, we proposed a cost estimation method for configurable, model-driven SoC design based on Machine Learning. The approach presented is fast, accurate, and well adapts to SoC configurations considered in the design flow and to different synthesis strategies. This is accomplished by a data representation which describes SoC configurations in a way that keeps all spatial information and, at the same time, is suited for ML algorithms. Evaluations confirm the accuracy of the proposed method and application studies demonstrated its benefits: While the state of the art for determining the cost required several hours per configuration, the proposed method provides an accurate cost estimation within seconds. Future work will cover further objectives such as timing and CPU cycles—additionally incorporating the execution of the firmware programs.

7 ACKNOWLEDGEMENT

This work has partially been supported by the LIT Secure and Correct Systems Lab funded by the State of Upper Austria, the German "Bundesministerium für Bildung und Forschung (BMBF)" projects COMPACT grant BMBF FE 01IS17028 and SAVE4I grant BMBF 01IS17032 as well as BMK, BMDW, and the Province of Upper Austria in the frame of the COMET Programme managed by FFG.

REFERENCES

[1] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. Chisel: constructing hardware in a scala embedded language. In *DAC Design Automation Conference*, 2012.
[2] Zhaowei Cai, Quanfu Fan, Rogerio S. Feris, and Nuno Vasconcelos. A unified multi-scale deep convolutional neural network for fast object detection. In *European conference on computer vision*. Springer, 2016.
[3] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. 2009.
[4] Frank P Coyle and Mitchell A Thornton. From UML to HDL: a model driven architectural approach to hardware-software co-design. In *Information systems: new generations conference (ISNG)*, volume 1, pages 88–93. Citeseer, 2005.
[5] Clifford E Cummings. Simulation and synthesis techniques for asynchronous FIFO design. In *Synopsys Users Group Conference*, 2002.
[6] Wolfgang Ecker and J. Schreiner. Metamodeling and code generation in the hardware/software interface domain. In *Handbook of Hardware/Software Codesign*, 2017.
[7] Sanford Friedenthal, Alan Moore, and Rick Steiner. *A practical guide to SysML: the systems modeling language*. Morgan Kaufmann, 2014.
[8] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.
[9] Joseph L. Greathouse and Gabriel H. Loh. Machine learning for performance and power modeling of heterogeneous systems. In *Proceedings of the International Conference on Computer-Aided Design*, ICCAD '18. ACM, 2018.
[10] David Money Harris and Sarah L. Harris. Hardware description languages. In *Digital Design and Computer Architecture (Second Edition)*, pages 172 – 237. Morgan Kaufmann, Boston, second edition edition, 2013.
[11] Fernando Herrera, Julio Medina, and Eugenio Villar. *Modeling Hardware/Software Embedded Systems with UML/MARTE: A Single-Source Design Approach"*.
[12] Hsuan Hsiao and Jason H. Anderson. Sensei: An area-reduction advisor for FPGA high-level synthesis. In *Design, Automation and Test in Europe*, 2018.
[13] Anuja P Jain and Padma Dandannavar. Application of machine learning techniques to sentiment analysis. In *2nd International Conference on Applied and Theoretical Computing and Communication Technology*, 2016.
[14] Oscar Luna, Daniel Torres, and RTAC Americas. DMX512 protocol implementation using MC9S08GT60 8-Bit MCU. 2006.
[15] Bryan McCann, James Bradbury, Caiming Xiong, and Richard Socher. Learned in translation: Contextualized word vectors. In *Advances in Neural Information Processing Systems*, 2017.
[16] Kevin P. Murphy. *Machine learning: a probabilistic perspective*. MIT press, 2013.
[17] Nasser M Nasrabadi. Pattern recognition and machine learning. *Journal of electronic imaging*, 16(4), 2007.
[18] Ethan Perez, Florian Strub, Harm De Vries, Vincent Dumoulin, and Aaron Courville. Film: Visual reasoning with a general conditioning layer. In *AAAI Conference on Artificial Intelligence*, 2018.
[19] James Rumbaugh, Ivar Jacobson, and Grady Booch. *Unified Modeling Language reference manual*. Pearson Higher Education, 2004.
[20] Lorenzo Servadei, Edoardo Mosca, Elena Zennaro, Keerthikumara Devarajegowda, Michael Werner, Wolfgang Ecker, and Robert Wille. Accurate cost estimation of memory systems utilizing machine learning and solutions from computer vision for design automation. *IEEE Transactions on Computers*, 69(6), 2020.
[21] Lorenzo Servadei, Elena Zennaro, Keerthikumara Devarajegowda, Martin Manzinger, Wolfgang Ecker, and Robert Wille. Accurate cost estimation of memory systems inspired by machine learning for computer vision. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2019.
[22] Shinya Takamaeda-Yamazaki. Pyverilog: A python-based hardware design processing toolkit for Verilog Hdl. In *International Symposium on Applied Reconfigurable Computing*, 2015.
[23] Andrew Waterman, Yunsup Lee, Rimas Avizienis, David A. Patterson, and Krste Asanovic. The RISC-V instruction set manual volume 2: Privileged architecture version 1.7. Technical report, University of California at Berkeley Berkeley United States, 2015.
[24] Gene Wu, Joseph L Greathouse, Alexander Lyashevsky, Nuwan Jayasena, and Derek Chiou. GPGPU performance and power estimation using machine learning. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture*, 2015.
[25] Elena Zennaro, Lorenzo Servadei, Keerthikumara Devarajegowda, and Wolfgang Ecker. A machine learning approach for area prediction of hardware designs from abstract specifications. In *2018 21st Euromicro Conference on Digital System Design (DSD)*. IEEE, 2018.
[26] Jiajun Zhang and Chengqing Zong. Deep neural networks in machine translation: An overview. *IEEE Intelligent Systems*, 2015.
[27] Xiang Zhang, Junbo Zhao, and Yann LeCun. Character-level convolutional networks for text classification. In *Proceedings of the 28th International Conference on Neural Information Processing Systems-Volume 1*. MIT Press, 2015.
[28] Zhong-Qiu Zhao, Peng Zheng, Shou-tao Xu, and Xindong Wu. Object detection with deep learning: A review. *IEEE transactions on neural networks and learning systems*, 2019.